Concurrent Objects in Java

Although Java currently provides classes to support concurrent objects, asynchronous method calls, and futures, it requires a very heavy weight framework to use, discouraging experimentation with the programming model. Other languages that provide more direct support for this programming model are being developed and in limited use.

We are developing a very lightweight (from the programmer’s perspective) approach to programming in this model using Java. In this paper we will describe our approach to programming concurrent objects in Java.

Introduction

It is widely acknowledged that concurrency will play a role in an increasing percentage of programs written in the future as we hit the limits of Moore’s Law. Multi-core processors are the norm, and the ever-growing demand for greater computation speeds can only be met with parallel computing. The Actor model has been proposed as a natural way to program concurrent systems. In the object-oriented world this can take the form of active concurrent objects that interact via asynchronous method calls. The model supports modular reasoning about the concurrency with relatively simple semantics.

Creol is one programming language developed with this in mind. In order to encourage broader experimentation and development with the style of programming supported by Creol, we created a lightweight approach to “Creol-style” programming using standard Java. We call it lightweight because there is just one package containing two classes, one interface, and five methods that the user needs to know about.

Although Java’s java.util.concurrent package provides support for Futures, those futures are not consistent with the active object concurrency model of Creol because the standard Java approach lacks the concept of an active object executing one process at a time. Each Java Future is connected to a separate runnable thread that will compute the future, but there is no implicit synchronization among runnable threads for the same object. Java also has support for asynchronous messages but only in the context of a complex framework of remote objects (cite xxx).

The current implementation uses pure Java, not requiring any form of preprocessing. This simplicity does have a downside as discussed below. The addition of a preprocessor would eliminate some of these disadvantages as is planned as future work.

Creol

The key elements of Creol that we emulate in Java are futures, asynchronous method calls, active objects (executing at most one process/call at a time), and the ability to release control of an object via an await statement.

Future<T> f; declares f to be a future that will eventually contain a value of type T. Execution in an object is initiated with an asynchronous method call which takes the form f = o!m(e); where o is the object, m is the method, e is a parameter list, and f will hold the future, that will eventually be filled in by the method. The statement x := get f; will block until the value for f has been produced. This blocks the entire object (which is only allowed to execute a single process at a time). To allow for greater concurrency, Creol also provides two forms of await that releases control of an object’s processor. The statement await x := get f; which suspends the current process (method execution) allowing the object to continue with some other method call or previously suspended call. The await statement can also be used with a Boolean expression as in await b; which will suspend the current process. A process suspended via either form of the await will only be allowed to continue once it is enabled (the future produced or the Boolean expression becomes true) and the object’s processor is released either at the completion of another method or when the active process suspends via an await.

Figure x shows a complete Creol program

To create an active (Creol Style) object a class need only extend the class CreolObject. The functions provided by a CreolJava are exemplified here:

// asynchronous call – param list may be empty

Future fut = obj.invoke(“methodName”, param1, param2, …);

// blocks waiting for the future value to be available

MyType x = (MyType)fut.get();

// non-blocking wait for future value to be available

MyType x = (MyType)CreolAwait(fut);

// voluntary suspend that may resume immediately

CreolSuspend();

// voluntary suspend that will not resume until some other action takes place in the object

CreolAwait();

Here is a complete example implementing a classic bounded buffer.

import Creol.\*;

public class BoundedBuffer **extends CreolObject** {

private int head, tail, cnt;

private Item[] buf;

BoundedBuffer(int size) {

buf = new Item[size];

}

public void insert(Item item) {

**while (cnt >= buf.length) CreolAwait();**

buf[head] = item;

head = (head+1)%buf.length;

cnt++;

}

public Item remove() {

**while (cnt <= 0) CreolAwait();**

Item result = buf[tail];

tail = (tail+1)%buf.length;

cnt--;

return result;

}

}

Here we discuss the key lines from this class.

**public** class BoundedBuffer **extends CreolObject** {

Each Creol object class simply extends CreolObject, which contains the majority of the functionality needed to handle Creol style asynchronous calls and active objects. The class must be public so that it is accessible by the code in the Creol package.

**public** void insert(Item item) {

...

}

Similarly, methods in Creol objects that will be “invoked” using the Creol style invocations must be public, even for self calls. This is necessary because they are technically being invoked by name from the Creol package.

**while** (cnt >= buf.length) **CreolAwait();**

To have a call wait for some local condition to become true we combine a loop to test the condition, with the CreolAwait() call. This call will block the caller (freeing the object to continue processing other calls) until “something” happens in the object. Specifically it will wait until another call is at least partially processed. At that time, the suspended call will be put in the queue with other calls that have invoked CreolSuspend() (see xxx) and be continued at some point in the future. There is of course no guarantee that the specified condition will be true, hence the while statement.

Here is a consumer for our bounded buffer above.

import Creol.\*;

public class Consumer extends CreolObject {

public Item consumeOne(BoundedBuffer buf) {

return (Item)buf.invoke("remove").get();

}

public void startConsuming(BoundedBuffer buf) {

Item item;

while (true) {

item = (Item)CreolAwait(this.invoke("consumeOne", buf));

System.out.println(item);

}

}

}

The consumer needs to wait for the actual value from the buffer before continuing so it makes a blocking synchronous call. That is, it makes the invocation of the remove method but immediately tries to get the value of the Future that is returned. The line

return (Item)buf.invoke("remove").get();

is a combination of two steps into one. Here we show the equivalent code that makes the presence of a future with an asynchronous call more visible.

Future fut = buf.invoke("remove");

// Execution can reach this point BEFORE the item is actually

// removed from the buffer.

return (Item) fut.get();

Here is a producer for the bounded buffer. When making synchronous self calls, it is more efficient to use the normal Java method invocation syntax. So in this example, the line

item = (Item)CreolAwait(this.invoke("consumeOne", buf));

can be written simply as

item = consumeOne(buf);

It would not be appropriate to make this substitution when making synchronous calls on other Creol objects. In the future we propose to provide a static checker that will generate an error if any non-self calls are made on Creol object methods using conventional Java method calling syntax.

Here is a producer for the bounded buffer.

import Creol.\*;

public class Producer extends CreolObject {

int num;

public void produceOne(BoundedBuffer buf) {

buf.invoke("insert",new Item(++num)).get();

}

public void startProducing(BoundedBuffer buf) {

while(true) {

CreolAwait(this.invoke("produceOne", buf));

}

}

}

Looking at a few lines from this we have

buf.invoke("insert",new Item(++num)).get();

This line makes a synchronous Creol style call to the insert method of the bounded buffer class. That is, it waits for the insert operation to complete before continuing. It does this by attempting to examine (calling get()) the Future that is returned.

And as was the case in the Consumer, we have semantically equivalent choices for making the synchronous self call. The more explicit (less efficient form) that makes it clear it is a Creol style invocation would be:

CreolAwait(this.invoke("produceOne", buf));

However, this is semantically equivalent to the simpler call:

this.produceOne(buf);

To complete the example, here is a Java main() that puts it all together.

import Creol.\*;

class BBMain extends CreolObject{

public static void main(String[] args) {

new BBMain().test();

}

void test() {

BoundedBuffer buf = new BoundedBuffer(3);

Producer prod = new Producer();

Consumer cons = new Consumer();

prod.invoke("startProducing",buf);

cons.invoke("startConsuming",buf);

}

}

There are a number of disadvantages to this pure Java approach to Creol style asynchronous method calls.

As described above, asynchronous method invocation is made with the following syntax:

obj.invoke(“methodName”[, parameter list])

As a consequence there are none of the usual compile time checks for proper spelling of the method name, or that there exists a method with the given name that accepts the offered parameter types. It would be possible to create a static analyzer that performs these checks. In the current implementation, any errors in the method name or parameter list will result in a NoSuchMethod exception at runtime. Another aspect of the method invocation process is that CreolObject methods must be public and CreolObject classes must also be public. This arises because the invocation using obj.invoke(…) is being made across package boundaries (from the package Creol to the application’s package).

The result of CreolObject method calls is a future, represented by an instance of the class Creol.Future. When extracting the contents of the future with get(), a cast is required.

Future fut = obj.invoke(“methodName”[, parameter list]);

SomeType val = (SomeType)fut.get();

The Java generic mechanism cannot be used here because different methods in the same CreolObject can return futures containing different types of values. Here also, a static analyzer could be built that would be able to automatically insert these casts.

This implementation also does not prevent a programmer from using the standard Java syntax for invoking CreolObject methods. Although synchronous self calls are permitted using the standard Java syntax. The symantics of

this.invoke(“methodName”[,parameter list]).get()

is the same as

this.methodName([parameter list])

and for obvious reasons the latter is preferred. As with the previous limitations, this too can be checked with a static checker. The checker would allow standard syntax for CreolObject methods calls on this, but no others.

In Creol, the equivalent of non-Creol objects are always passed by value. In Java, these other non-Creol objects are passed by reference, resulting in multiple active CreolObjects potentially modifying the same data value. To properly embrace the Creol programming model, write access to shared objects (non-Creol objects) should not be allowed. This too can be detected using a static analyzer.

In summery, the features of a JavaCreol static checker should include:

* Perform the usual signature checking for methods invoked using the CreolObject invoke() method.
* Eliminate the need for the explicit type casts and insure type safety for assignments involving get() to extract the contents of a future.
* Prevent the programmer from directly invoking CreolObject methods using standard Java method calling, except for self calls, which are then equivalent to synchronous self calls.
* Generate an error or warning for any shared write access to non-CreolObjects by CreolObject methods.

OTHER THOUGHTS somehow related

Creol objects are essentially like remote objects in Java (or should act like them). This eliminates the need for the Remote non-remote object distinction. Does this simplify the value/reference problem I noted in the Formalising Java RMI paper? Semantically treat every CreolObject like a remote object. Let the system figure out what to do if it doesn’t happen to be local. If that is a hard problem, then tackle that. Get the programmer out of the business. What is the latest in the parallel programming community about programmer controlling or caring about what is local and what is not?

What if objects could easily be identified as immutable, possibly explicitly by the programmer (and verified by the compiler)? This would make the programs clearer and simplify some RMI type issues with respect to value vs ref semantics which are the same for immutable objects.

Would a move to Creol style objects for all (except pass by value data objects or immutable pass by reference objects) result in simpler code and “good enough” performance with the Creol objects distributed across different systems (as opposed to being all local as in my current implementation)?

We have four variations of calls: blocking synchronous, non-blocking synchronous, asynchronous followed by blocking get, and asynchronous followed by non-blocking get.