CreoleJava is a package that facilitates programming using the Creole synchronization model in Java. The Creole synchronization model is …

Although Java’s java.util.concurrent package provides support for Futures, those futures are not consistent with the concurrency model of Creol because it lacks the concept of an active object with a single process. 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. (Needs work just wanted to capture this thought. Seems necessary to explain why I didn’t just use the Future objects from the Java API.)

The current implementation is done in pure Java, not requiring any form of preprocessing. The addition of a preprocessor would eliminate some of the limitations (see below).

To create an active (Creole Style) object a class need only extend the class CreoleObject. The functions provided by a CreoleJava 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)creoleAwait(fut);

// voluntary suspend that may resume immediately

creoleSuspend();

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

creoleAwait();

Here is a complete example implementing a classic bounded buffer.

import creole.\*;

public class BoundedBuffer **extends CreoleObject** {

private int head, tail, cnt;

private Item[] buf;

BoundedBuffer(int size) {

buf = new Item[size];

}

public void insert(Item item) {

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

buf[head] = item;

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

cnt++;

}

public Item remove() {

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

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 CreoleObject** {

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

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

...

}

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

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

To have a call wait for some local condition to become true we combine a loop to test the condition, with the creoleAwait() 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 creoleSuspend() (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 creole.\*;

public class Consumer extends CreoleObject {

public Item consumeOne(BoundedBuffer buf) {

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

}

public void startConsuming(BoundedBuffer buf) {

Item item;

while (true) {

item = (Item)creoleAwait(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 non-blocking synchronous self calls, it is more efficient to use the normal Java method invocation syntax. So in this example, the line

item = (Item)creoleAwait(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 creole objects. In the future we propose to provide a static checker that will generate an error if any non-self calls are made on creole object methods using conventional Java method calling syntax.

Here is a producer for the bounded buffer.

import creole.\*;

public class Producer extends CreoleObject {

int num;

public void produceOne(BoundedBuffer buf) {

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

}

public void startProducing(BoundedBuffer buf) {

while(true) {

creoleAwait(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 creole 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 creole style invocation would be:

creoleAwait(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 creole.\*;

class BBMain extends CreoleObject{

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 creole 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 CreoleObject methods must be public and CreoleObject classes must also be public. The arises because the invocation using obj.invoke(…) is being made across package boundaries (from the package creole to the application’s package).

The result of CreoleObject method calls is a future, represented by and instance of the class creole.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 CreoleObject 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 CreoleObject 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 CreoleObject methods calls on this, but no others.

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

In summery, the features of a JavaCreole static check should include:

* Perform the usual signature checking for methods invoked using the CreoleObject 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 CreoleObject 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-CreoleObjects by CreoleObject methods.

OTHER THOUGHTS somehow related

Creole 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 CreoleObject 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 creole 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 creole 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 clocking get, and asynchronous followed by non-blocking get.