Creol (and others) have suggested the use of concurrent objects communicating via asynchronous method calls and futures, as a pathway to better reasoning about concurrent systems. The communication and synchronization model of Creol simplifies deadlock detection, allows for …

These advantages do not come without a price. Although not unique to programs created using Creol style synchronization, this programming style does make it quite easy to create programs that are semantically correct but that fail due to over eager creation of suspended method calls. Consider this example of the publish/subscribe model taken from xxx.

data News=E1|E2|E3|E4|E5|None;

interface ServiceI{

Void subscribe(ClientIcl);

Void produce()}

interface ProxyI{

ProxyIadd(ClientIcl);

Void publish(Fut<News>fut)}

interface ProducerI{

NewsdetectNews()}

interface NewsProducerI{

Void add(Newsns);

NewsgetNews();

List<News>getRequests()}

interface ClientI{

Void signal(Newsns)}

class Service(Intlimit,NewsProducerInp) implements ServiceI{

ProducerI prod;ProxyIproxy;ProxyIlastProxy;

{ prod := new Producer(np);

proxy:= new Proxy(limit,this);lastProxy:=proxy;this!produce()}

Void subscribe(ClientIcl){lastProxy:=lastProxy.add(cl)}

Void produce(){var Fut<News>fut:=prod!detectNews();proxy!publish(fut)}}

class Proxy(Intlimit,ServiceIs) implements ProxyI{

List<ClientI> myClients:=Nil;ProxyInextProxy;

ProxyI add(ClientIcl){

var ProxyI lastProxy=this;

if length(myClients)<limit then myClients:=appendright(myClients,cl)

else if nextProxy==null then nextProxy:= new Proxy(limit,s) fi;

lastProxy:=nextProxy.add(cl) fi;

put lastProxy}

Void publish(Fut<News>fut){

var News ns=None;

ns =fut.get; myClients!signal(ns);

if nextProxy==null then s!produce() else nextProxy!publish(fut) fi}}

class Producer(NewsProducerI np) implements ProducerI{

News detectNews(){

News news:=None;

news:=np.getNews(); put news}}

class NewsProducer implements NewsProducerI{

List<News>requests:=Nil;

Void add(News ns){requests:=appendright(requests,ns)}

News getNews(){

var News firstNews:=None; await requests /= Nil;

firstNews := head(requests);requests:=tail(requests); put firstNews}

}

class Client implements ClientI{

Newsnews:=None;

Void signal(News ns){news:=ns}}

Modifying Client and Proxy as shown below, results in a program that will swamp the system with suspended calls. The changes are shown in boldface. The change is to shift requiring the actual news to have arrived from the Proxy (ns =fut.get; myClients!signal(ns);) to the Client (news:=fut.get).

class Proxy(Intlimit,ServiceIs) implements ProxyI{

List<ClientI> myClients:=Nil;ProxyInextProxy;

ProxyIadd(ClientIcl){

var ProxyIlastProxy=this;

if length(myClients)<limit then myClients:=appendright(myClients,cl)

else if nextProxy==null then nextProxy:= new Proxy(limit,s) fi;

lastProxy:=nextProxy.add(cl) fi; put lastProxy}

Void publish(Fut<News>fut){

**myClients!signal(fut);**

if nextProxy==null then s!produce() else nextProxy!publish(fut) fi}}

class Client implements ClientI{

News news:=None;

Void signal(Fut<News> fut){**news:=fut.get**}}

This seemingly minor change, and one that would even seem to make sense in the interest of maximizing concurrency, is in fact “too much.” We might naively take it even one step further and have the client instead do news:=await(fut) which has the additional advantage of allowing the Client to process the news items as they become available, rather then in the order that the futures were received. In either case, the following sequence of calls can occur, which constitute an unbounded loop.

Service.produce calls Proxy.publish

Proxy.publish calls Service.produce

Each pass around this “loop” also spawns an asynchronous call to Producer.detectNews. Depending upon the speed of execution of the code along the path of the loop, such a loop can create an unbounded number of suspended calls to Producer.detectNews. There is no synchronization where the loop needs to wait for any news to actually be detected.

We call such sequences racing-loops (better name?). In this paper we present an algorithm to statically identify programs that contain racing-loops. This approach is conservative in that if it reports that a program is free from racing-loops then it is indeed free of such loops, however, it may report racing-loops that are in fact bounded by program logic, not amenable to static analysis. In will also report racing-loops that do not in practice produce an increasing number of unprocessed calls due to the execution speed of the racing-loop.

A racing-loop must always be racing against one or more specific asynchronous calls, either trivial calls (no waiting on any future from the call) or calls where the resulting future is not read in the loop, although the future may be passed to a separate asynchronous call where it is read.

Our first version of the program has a racing-loop (Service.produce …) that is racing against the Client.signal calls generated in Proxy.publish. This will not cause a problem, provided the Client objects are able to process these signal calls at least as fast as they are being generated. Our algorithm will alert the programmer to this situation, and the programmer can determine if there is a real problem here, possibly with the aid of some additional program instrumentation.

The second version of the program has a racing-loop that is racing against both the Client.signal calls as above, but also against the Producer.detectNews calls. This is because rather than wait in Proxy.publish for the news to actually be produced as in the first version (ns =fut.get;), Proxy.publish instead simply passes the future out to another asynchronous call (myClients!signal(fut);), eliminating any progress coordination between the racing-loop and the Client.signal calls. This second racing loop is more likely to be a problem because it is not dependent simply on execution speed of some code but is dependent upon the arrival rate of news items and in practice will always result in the number of unprocessed calls quickly growing to system limits.

The Algorithm

We first create a control flow graph for each method where the nodes are either method calls, future get calls, or puts (implicit at the end of every method). All other statements are ignored. In addition, the nodes are of two types, blocking, or non-blocking. Blocking nodes are calls to get on a future, await() calls, or blocking Creol calls (probably should use the Creol notation for each here). (I maybe need to find another word since blocking is being overloaded – e.g. await is considered non-blocking in the Creol sense.)

Next we perform a variation of data flow analysis where we are flowing calls and futures through the graph to see if there is direct or indirect recursion. Here we use recursion loosely in that if the same method is called but for a different object we still consider that recursive. That is if there are two instances of class C, call them a and b, and if C contains a method m, and if a.m() results directly or indirectly in a call to b.m() then we say there is a recursive call on m(). (Reword to indicate that a and b can be the same object but need not be.)

If there is a recursive chain that creates futures (including implicit Void futures for trivial calls) that are not read in the chain, then there is a racing-loop with respect to the call that produced the unread future.

Flow Analysis

1. Build the individual control flow graphs for each method including just calls, get/await, and put.
2. Add call edges between the method cfgs.
3. Use flow analysis to compute the put edges from puts to get/await. (What is going to happen with more complex examples where there are multiple puts that might reach a given get?)
4. Use flow analysis to identify any recursive call loops.
5. Use flow analysis to identify any futures created by a loop but not read by the loop

After step 4 we will have a graph with start nodes, call nodes (blocking or non-blocking), get/await nodes, and put/return nodes. We will have flow edges (within a method), call edges (between methods), and put edges from put/return to get/await.

Step 4 analysis proceeds from the entry point start node.

For each (in some order? Breadth First? Does it matter?) flow node Ni with input edge sets Pi, the output set Oi is computed as:

if the node is an async call to C.m Oi = Union(Pi)+C.m

else if the node is a start node, Oi = Union(Pi)-self

otherwise Oi = Union(Pi)

Repeat until there are no further changes in any Oi.

A recursive call loop exists if Union(Pi) for some start node Ni contains Ni.

Step 5 analysis then proceeds by picking a method start node that is part of a recursive call loop.

For each (in some order? Breadth First? Does it matter?) flow node Ni with input edge sets Pi, the output set Oi is computed as:

if the node is an async call to C.m Oi = Cancel(Union(Pi)+C.m)

else if the node is a get, Oi = Union(Pi)

else if the node is a put, Oi = {-C.m} where C.m is the start of the cfg ending in the put

else if the node is a start node, Oi = Union(Pi)-self

otherwise Oi = Cancel(Union(Pi))

Repeat until there are no further changes in any Oi.

Cancel(S) is S minus matching C.m and –C.m entries.

There is a racing-loop if there is a loop such that the intersection of Oi for all Ni in the loop is non-empty. Any call in that intersection is racing with respect to the loop. Figure x shows the state of the graph and the output sets after the completion of step 5 for the first version of the producer consumer problem above.

