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(){

var List<News>requests:=Nil; News news:=None;

requests :=np.getRequests();

while requests ==Nil do requests :=np.getRequests() od

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

List<News> getRequests(){put requests}}

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.” In fact, 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 created. In either case, the following sequence of calls can occur, which constitute an unbounded loop creating suspended method calls.

Service.produce calls Proxy.publish

Proxy.publish calls Service.produce

Several asynchronous calls are made along the way which only contribute to the quantity of suspended (unfinished) calls, but they can increase without bound either way. We call such sequences async-loops (better name?). In this paper we present an algorithm to statically identify programs that contain async-loops. This approach is conservative in that if it reports that a program is free from async-loops then it is indeed free of such loops, however, it may report async-loops that are in fact bounded by program logic, not amenable to static analysis.

We first create a control flow graph for each method where the nodes are either method calls or future get calls. All other statements are ignored. (May need to include put eventually.) 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 through the graph to see if there is direct or indirect recursion. Here we use recursion loosely in that if the same method is call but for a different object we still consider that recursive. That is if there are two instances class C, call them a and b, 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().

We also use data flow analysis to track the number of futures that have been requested (via asynchronous calls) and the number of futures that have been retrieved (via get or await).

If there is a recursive chain where the number of futures requested exceeds the number of futures retrieved, then there is an async-loop.

Probably delete stuff below. A false start following too closely along the lines of the deadlock detection algorithm.

The possible calls in the original version are:

Service.produce calls Proxy.publish (trivial)

Service.produce calls Producer.detectNews (future)

Service.subscribe calls Proxy.add (blocking)

Proxy.add calls Proxy.add (blocking – younger)

Proxy.publish calls get (the key seems to be that this get precedes any of the other calls)

Proxy.publish calls Client.signal (trivial)

Proxy.publish calls Service.produce (trivial)

Proxy.publish calls Proxy.publish (trivial)

Producer.detectNews calls NewsProducer.getRequests (blocking)

Producer.detectNews calls NewsProducer.getNews (blocking)

Producer.detectNews calls put

NewsProducer.getNews calls put

NewsProducer.getRequests calls put

The possible calls in the modified version are the same with the exception of one addition:

Client.signal calls get

and one deletion

Proxy.publish no longer calls get.

The chains in the original are