BishBashBosh

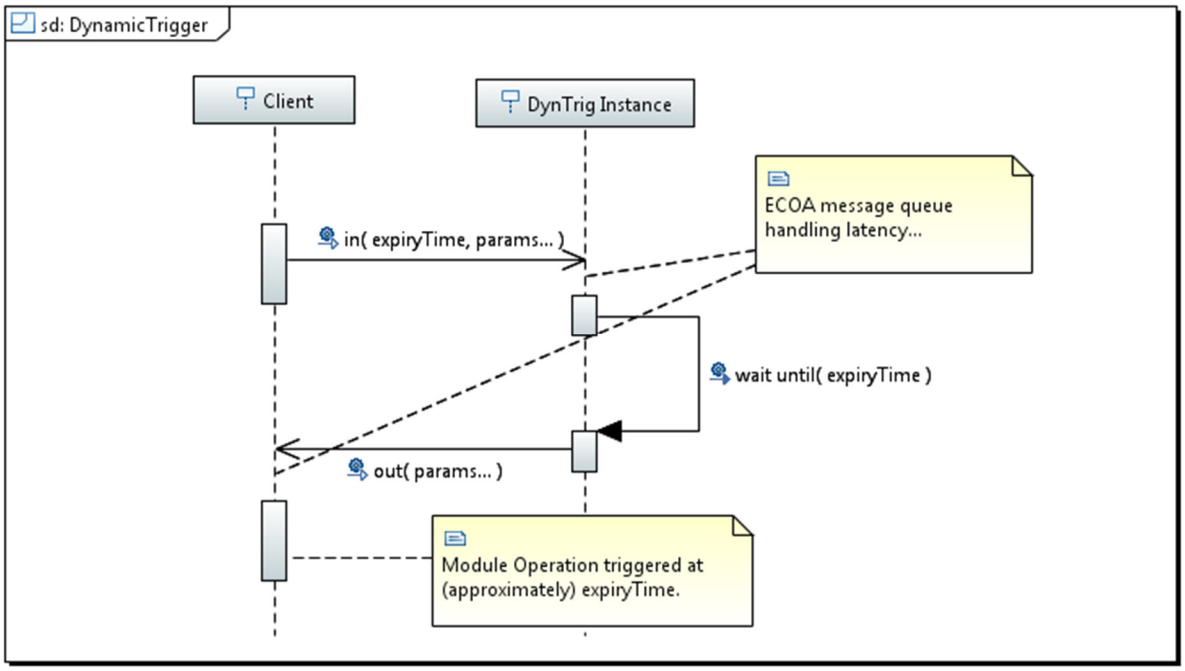
Introduction

This document describes an ECOA® example of using the ECOA Dynamic Trigger Instance mechanism.

This document presents outline information about principal user generated artefacts required to create a "*BishBashBosh*" program using the ECOA. It is assumed that the reader is *thoroughly* conversant with the ECOA Architecture Specification (ref.[1]) and the process of defining and declaring ECOA Assemblies, ASCs (components), Modules, and deployments in XML, and then using code generation to produce Module framework (stub) code units and ECOA Container and Platform code. If not, then let me suggest working through some of the other examples/samples provided, starting with "*Hello World*" and working your way up to "*Pub Sub*".

An ECOA Dynamic Trigger Instance allows software to schedule a single asynchronous ECOA Event at some future time. The client "sets" the trigger (using the "*in*" ECOA Event Operation) to expire at a given time and (possibly) passes values for one or more parameters. When that future time arrives, the trigger sends an "out" ECOA Event, carrying the parameter values if applicable, and the Module Operation linked to that Event is called. This is depicted in Figure 1. As in all ECOA Operation invocations there will be a small intervals during the time the ECOA Operation is queued (see ref.[1] for an explanation of ECOA Operation queuing).

Figure 1 ECOA Dynamic Trigger Behaviour



An ECOA Dynamic Trigger can, within the context of the ECOA Inversion-of-Control principle, therefore be used where a "*delay until*" statement might be used in Ada, or a "*sleep()*" function in C[1]. Note however that unlike these two, the Module Operation that "sets" the trigger (using the "*in*" operation) does not resume when the trigger expires. A different Module Operation, or even a Service Operation, is invoked.

ECOA Dynamic Trigger Instances are not "busy waits". The invoking Module thread is not blocked.

Aims

This ECOA "*BishBashBosh*" example is intended to provide a simple example of how ECOA Dynamic Trigger Instances can be used.

ECOA Features Exhibited

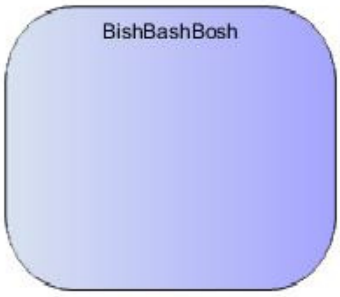
- ECOA Dynamic Trigger Instances.

Design and Definition

ECOA Assembly Design and Definition

As with many of these examples, "*BishBashBosh*" comprises a single ECOA ASC (Component) with no *provided* or *referenced* Services, as in Figure 2.

Figure 2 ECOA "*BishBashBosh*" Assembly Diagram



This ECOA Assembly is defined in an Initial Assembly XML file, and declared in a Final Assembly (or Implementation) XML file (which is practically identical). The Final Assembly XML for the ECOA "*BishBashBosh*" Assembly is as follows (file *bbb.impl.composite*):

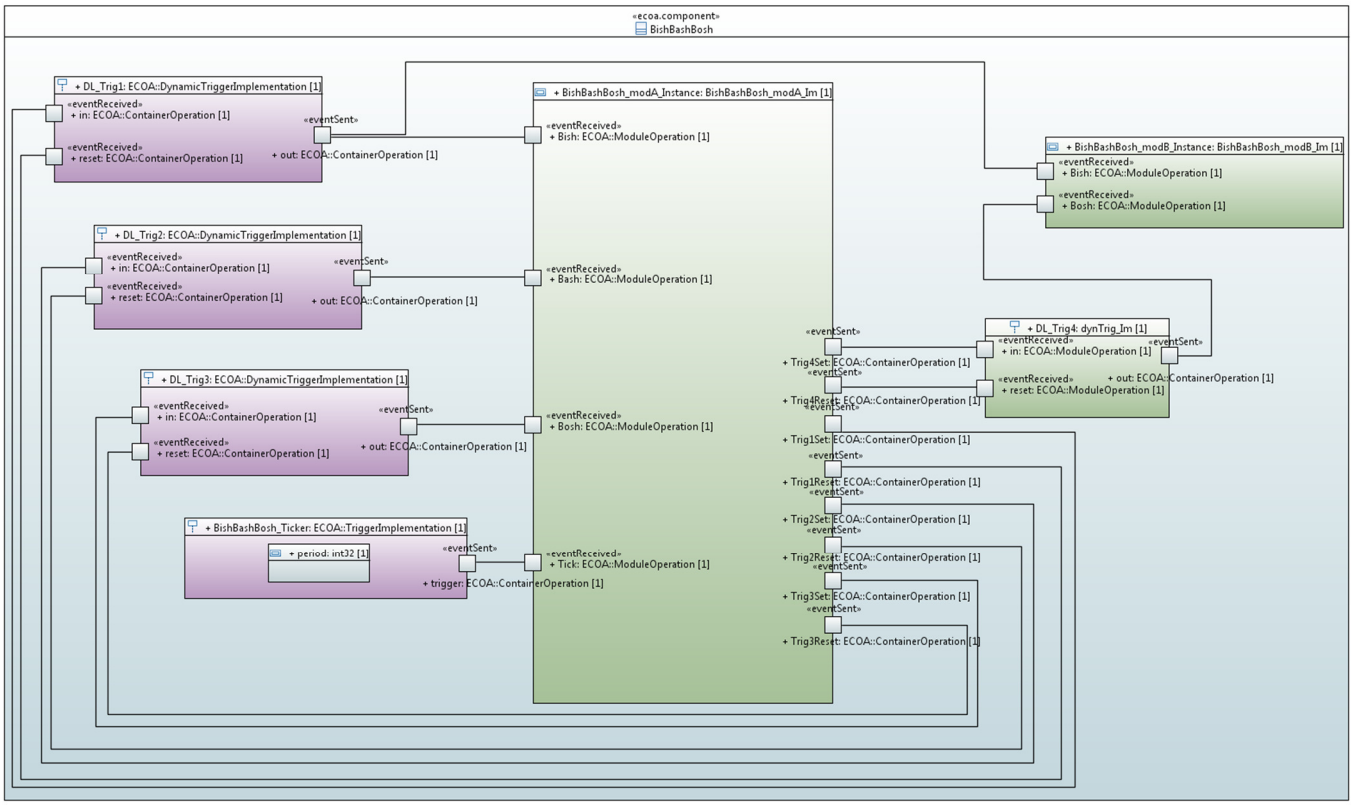
[1] Note however that the ECOA Dynamic Trigger and the Ada "*delay until*" statement use an absolute time reference. "*sleep()*" uses a relative time reference – a delay period.

```
<csa:composite xmlns:csa="http://docs.oasis-open.org/ns/opencsa/sca/200912"
               xmlns:ecoa-sca="http://www.ecoa.technology/sca-extension-2.0"
               name="bbb" targetNamespace="http://www.ecoa.technology">
  <csa:component name="BishBashBosh">
    <ecoa-sca:instance componentType="BishBashBosh">
      <ecoa-sca:implementation name="BishBashBosh"/>
    </ecoa-sca:instance>
  </csa:component>
</csa:composite>
```

ECOA Module Design and Definition

The *BishBashBosh* ASC is composed of three ECOA Modules defined (as Module Types) and declared (as Module Implementations and Instances) following the normal ECOA principals. The "*BishBashBosh*" ECOA example was originally created as a development test vehicle for ECOA Dynamic Trigger Instances, and therefore includes an ECOA Module Implementation simulating the Dynamic Trigger Instance behaviour. This Module has been left in for interest (Module Implementation *dynTrig_Im*). The internal interactions of the *BishBashBosh* ASC are depicted in UML in Figure 3.

Figure 3 The ECOA "*BishBashBosh*" ASC Internals as an UML Composite Structure



Module *BishBashBosh_modA* receives four ECOA Event Module Operations: *Bish*, *Bash*, *Bosh*, and *Tick*. *Tick* is invoked periodically by a "normal" ECOA Trigger Instance (*BishBashBosh_Trigger*). The other three operations are invoked when the Dynamic Trigger Instances (*DL_Trig1*, *DL_Trig2*,

and *DLTrig3*) fire. *DL_Trig1* invokes Module Operation *Bish* in both Modules *BishBashBosh_modA* and *BishBashBosh_modB*.

Module Instance *DL_Trig4* is the simulated, application level implementation, of the Dynamic Trigger Instance behaviour. When it fires, Module Operation *Bosh* of *BishBashBosh_modA* is invoked.

All four Dynamic Trigger Instances (i.e. the three "real" ones and the simulated one) are "set" (re-programmed), using their *in* operation, by Module *BishBashBosh_modA*. Periodically *DL_Trig2* is cancelled using its *reset* operation (see ref.[1] for an explanation).

The Module Type definition *BishBashBosh_modA_t*, taken from the ECOA Component Implementation XML (file *BishBashBosh.impl.xml*) is (for illustration):

```
<moduleType name="BishBashBosh_modA_t" hasUserContext="true"
                hasWarmStartContext="false">
    <operations>
      <eventReceived name="Tick" />
      <eventReceived name="Bish">
         <input name="a" type="int32" />
         <input name="b" type="bbb:LogItem" />
      </eventReceived>
      <eventReceived name="Bash" />
      <eventReceived name="Bosh">
         <input name="a" type="int32" />
         <input name="b" type="bbb:LogItem" />
      </eventReceived>
      <eventSent name="Trig1Set">
         <input name="expiryTime" type="ECOA:global_time" />
         <input name="a" type="int32" />
         <input name="b" type="bbb:LogItem" />
      </eventSent>
      <eventSent name="Trig1Reset"/>
      <eventSent name="Trig2Set">
         <input name="expiryTime" type="ECOA:global_time" />
      </eventSent>
      <eventSent name="Trig2Reset"/>
      <eventSent name="Trig3Set">
         <input name="expiryTime" type="ECOA:global_time" />
         <input name="a" type="int32" />
         <input name="b" type="bbb:LogItem" />
      </eventSent>
      <eventSent name="Trig3Reset"/>
      <eventSent name="Trig4Set">
         <input name="expiryTime" type="ECOA:global_time" />
         <input name="a" type="int32" />
         <input name="b" type="bbb:LogItem" />
      </eventSent>
      <eventSent name="Trig4Reset"/>
    </operations>
</moduleType>
```

Note how the *Bish* and *Bosh* Module Operations receive the additional (application) parameters (*a* and *b*) posted by the Dynamic Trigger Instance, which are set by the *TrigNSet* operations of the relevant Dynamic Trigger Instance.

The rest of the ECOA Component Implementation XML is not repeated here. If you have worked through (some of) the other examples, you should be familiar enough with what it contains…

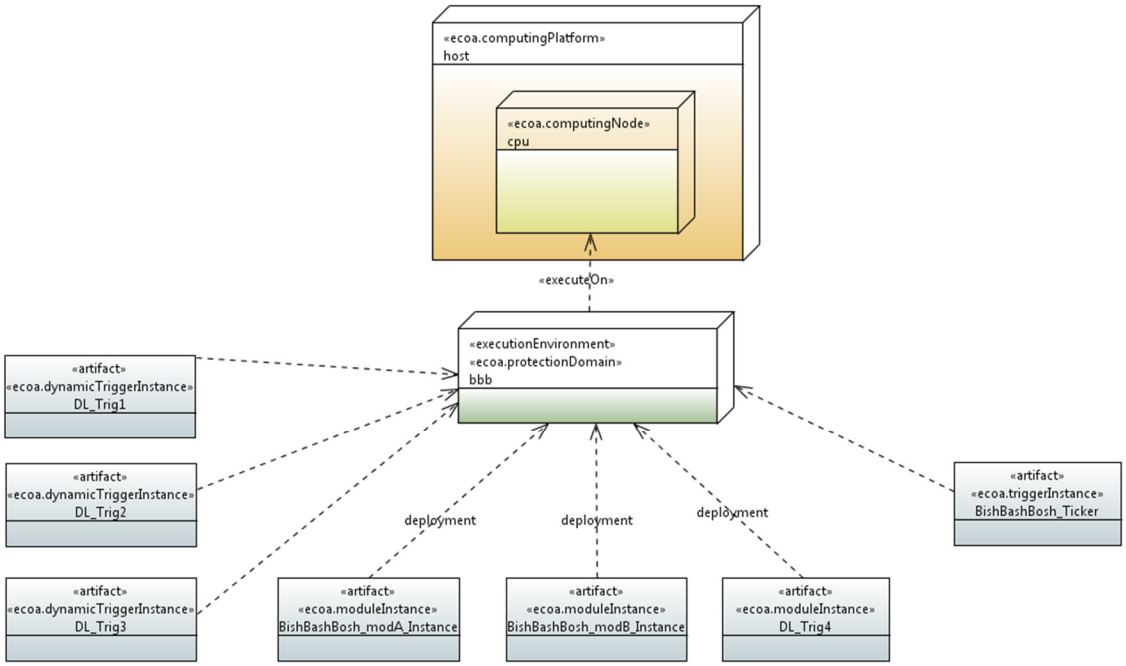
ECOA Deployment Definition

The ECOA "*BishBashBosh*" Assembly is deployed (that is, the declared Module, Trigger, and Dynamic Trigger Instances are allocated to an ECOA Protection Domain, which is itself allocated to a computing node) by the following XML (file *bbb.deployment.xml*):

```
<deployment xmlns="http://www.ecoa.technology/deployment-2.0"
                                finalAssembly="bbb" logicalSystem="hostbased">
    <protectionDomain name="bbb">
        <executeOn computingNode="cpu" computingPlatform="host"/>
        <deployedModuleInstance  componentName="BishBashBosh"
                                moduleInstanceName="BishBashBosh_modA_Instance"
                                modulePriority="50"/>
        <deployedModuleInstance  componentName="BishBashBosh"
                                moduleInstanceName="BishBashBosh_modB_Instance"
                                modulePriority="50"/>

        <deployedTriggerInstance componentName="BishBashBosh"
                                triggerInstanceName="BishBashBosh_Ticker"
                                triggerPriority="51"/>

        <deployedTriggerInstance componentName="BishBashBosh"
                                triggerInstanceName="DL_Trig1" triggerPriority="52"/>
        <deployedTriggerInstance componentName="BishBashBosh"
                                triggerInstanceName="DL_Trig2" triggerPriority="52"/>
        <deployedTriggerInstance componentName="BishBashBosh"
                                triggerInstanceName="DL_Trig3" triggerPriority="52"/>

        <deployedModuleInstance  componentName="BishBashBosh"
                                moduleInstanceName="DL_Trig4"  modulePriority="52"/>
    </protectionDomain>
    <platformConfiguration faultHandlerNotificationMaxNumber="8"
                                computingPlatform="host"/>
</deployment>
```

Thus in this case, all Module, Trigger, and Dynamic Trigger Instances are deployed into a single ECOA Protection Domain *bbb* executing on the Computing Node *cpu* which is part of the ECOA Computing Platform *host*:

Figure 4 ECOA " *BishBashBosh* " Deployment

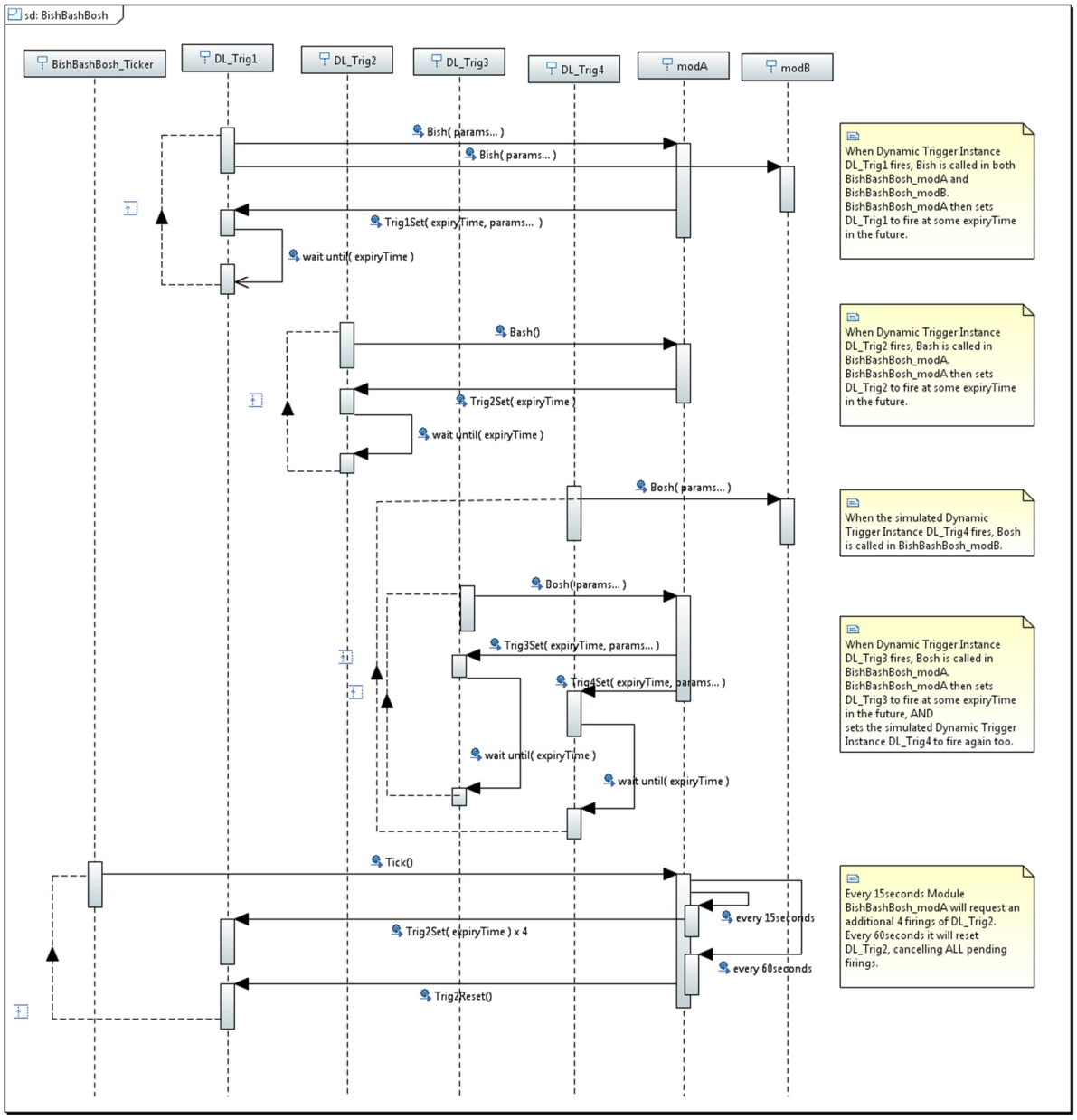


Implementation

Figure 5 shows the overall behaviour of "*BishBashBosh*" as a UML sequence diagram. The comments at the side describe the various behaviours depicted. Note that initialisation of the Dynamic Trigger Instances is omitted for clarity. The periodic behaviours invoked by the (normal) ECOA Trigger Instance *BishBashBosh_Ticker* are included:

a) To demonstrate multiple, queued, invocations of a Dynamic Trigger Instance (*DL_Trig2* in this case);
b) Cancelling queued invocations (also of *DL_Trig2*).

Figure 5 "*BishBashBosh*" Overall Behaviour



"*BishBashBosh*" Functional Code

The C code implementation of the *Bish* and *Bosh* Module Operations of Module Implementation *BishBashBosh_modA_Im* are, for illustration and comparison with the behaviour illustrated above:

```
void BishBashBosh_modA_Im__Bish__received(BishBashBosh_modA_Im__context *context,
                          const ECOA__int32 a,
                          const bbb__LogItem *b)
     {
          ECOA__global_time next_time;
          ECOA__log msg;
          //
          BishBashBosh_modA_Im_container__get_UTC_time( context, &next_time );
```

```
    context->user.TDEL.seconds = ( context->user.TDEL.seconds + 1 ) % 10;
    msg.current_size = sprintf( msg.data, "Bish! (DL_Trig1) a = %u; b->timeStamp =
                       {%u,%09u}; next in %u.%09u seconds...",
                       a, b->timeStamp.secs, b->timeStamp.nsecs,
                       context->user.TDEL.seconds,
                       context->user.TDEL.nanoseconds );
    BishBashBosh_modA_Im_container__log_info( context, msg );
    //
    next_time.seconds     += context->user.TDEL.seconds;
    next_time.nanoseconds += context->user.TDEL.nanoseconds;
    BishBashBosh_modA_Im_container__Trig1Set__send( context, &next_time, 1234, &LOGITEM );
}

void BishBashBosh_modA_Im__Bosh__received(BishBashBosh_modA_Im__context *context,
                       const ECOA__int32 a,
                       const bbb__LogItem *b)
{
    ECOA__global_time next_time;
    ECOA__log msg;
    //
    BishBashBosh_modA_Im_container__get_UTC_time( context, &next_time );
    context->user.TDEL.seconds = ( context->user.TDEL.seconds + 3 ) % 30;
    msg.current_size = sprintf( msg.data, "Bosh! (DL_Trig3) a = %d; b->timeStamp =
                       {%u,%09u}; next in %u.%09u seconds...",
                       a, b->timeStamp.secs, b->timeStamp.nsecs,
                       context->user.TDEL.seconds,
                       context->user.TDEL.nanoseconds );
    BishBashBosh_modA_Im_container__log_info( context, msg );
    //
    next_time.seconds     += context->user.TDEL.seconds;
    next_time.nanoseconds += context->user.TDEL.nanoseconds;
    BishBashBosh_modA_Im_container__Trig3Set__send( context, &next_time, 1234, &LOGITEM );
    BishBashBosh_modA_Im_container__Trig4Set__send( context, &next_time, 1234, &LOGITEM );
}
```

The Module Operation implementations in Module *BishBashBosh_modB_Im* are trivial, simply reporting that they have been invoked:

```
void BishBashBosh_modB_Im__Bish__received(BishBashBosh_modB_Im__context *context,
                       const ECOA__int32 a,
                       const bbb__LogItem *b)
{
    ECOA__log msg;
    msg.current_size = sprintf( msg.data, "Beep! (DL_Trig1) a = %u; b->timeStamp =
                       {%u,%09u}...", a, b->timeStamp.secs, b->timeStamp.nsecs );
    BishBashBosh_modB_Im_container__log_info( context, msg );
}

void BishBashBosh_modB_Im__Bosh__received(BishBashBosh_modB_Im__context *context,
                       const ECOA__int32 a,
                       const bbb__LogItem *b)
{
    ECOA__log msg;
    msg.current_size = sprintf( msg.data, "Bang! (DL_Trig4) a = %u; b->timeStamp =
                       {%u,%09u}...", a, b->timeStamp.secs, b->timeStamp.nsecs );
    BishBashBosh_modB_Im_container__log_info( context, msg );
}
```

Simulated ECOA Dynamic Trigger Instance Code

In order to simulate the Dynamic Trigger Instance behaviour the Module Implementation has to be very non-ECOA. The key Module Operation is *in* which, in C code has the function signature:

```
void dynTrig_im__in__received(dynTrig_im__context *context,
                        const ECOA__global_time *expiryTime,
                        const ECOA__int32 a,
                        const bbb__LogItem *b);
```

The function must:

a) Program a timer to expire when *expiryTime* is reached;
b) Invoke the *out* Operation when that timer expires;
c) Hold the values of the parameters a and b until the timer expires and then pass those values with the *out* operation;
d) Be able to do this for multiple invocations of *in* while previous invocations are still pending timer expiry.

The simple implementation used here is:

```
void dynTrig_im__in__received(dynTrig_im__context *context,
                        const ECOA__global_time *expiryTime,
                        const ECOA__int32 a,
                        const bbb__LogItem *b)
{
    ECOA__global_time now;
    ECOA__duration    delayTime;
    thrdInfo         *thrdInf = NULL;
    Thread_Attr_Type  thrdAttr;
    Create_Thread_Status_Type ctStat;
    int               thrdId;
    //
    dynTrig_im_container__get_UTC_time( context, &now );
    delayTime.seconds = expiryTime->seconds - now.seconds;
    delayTime.nanoseconds = expiryTime->nanoseconds - now.nanoseconds;
    //
    if( delayTime.seconds >= 0 && delayTime.nanoseconds > 0 ){
        thrdInf = (thrdInfo*)malloc( sizeof( thrdInfo ));
        thrdInf->context = context;
        thrdInf->usec = (delayTime.seconds * 1000000 ) + (delayTime.nanoseconds / 1000 );
        thrdInf->a = a;
        memcpy( &(thrdInf->b), b, sizeof(bbb__LogItem));
        //
        thrdAttr = (Thread_Attr_Type){ Sched_FIFO, 99, 1000000 };
        Create_Thread( &thrdAttr, &waitFunc, thrdInf, &thrdId, &ctStat );
    }
}
```

The function calculates how long until *expiryTime* occurs (in this case in microseconds) and stores this, along with the parameter values, into memory (at *thrdInf*). The function then creates a new thread, passing *thrdInf* to the thread function (*waitFunc*). This function therefore breaks the ECOA's Inversion-of-Control principle by creating that independent thread.

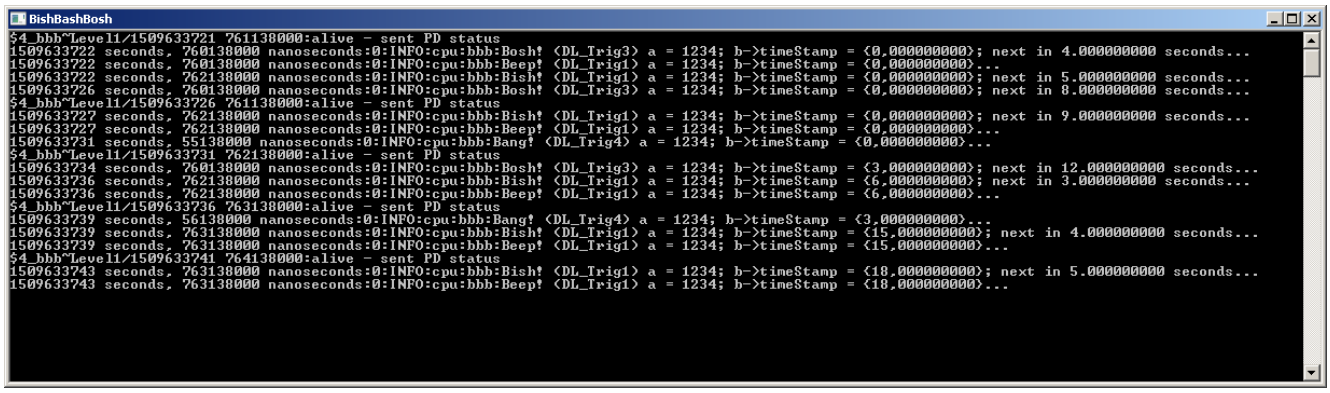
The thread function (*waitFunc*) simply waits for the passed number of microseconds, and then invokes the *out* operation with the passed parameter values:

```
static void waitFunc(void* p)
{
    thrdInfo *thrdInf = (thrdInfo *)p;
    usleep( thrdInf->usec );
    dynTrig_im_container__out__send( thrdInf->context, thrdInf->a, &(thrdInf->b) );
    free( p );
}
```

Program Output

When the ECOA "*BishBashBosh*" Assembly is built and run, an output similar to Figure 6 should be achieved. Text messages are output to the system console indicating when each Dynamic Trigger Instance fires and the values of the parameters passed, interleaved with other ECOA Platform logging messages (such as the 5 second periodic "alive" message in the example shown):

Figure 6 ECOA "*BishBashBosh*" in Execution



References

1	European Component Oriented Architecture (ECOA®) Collaboration Programme: Architecture Specification (Parts 1 to 11) "ECOA" is a registered trade mark.