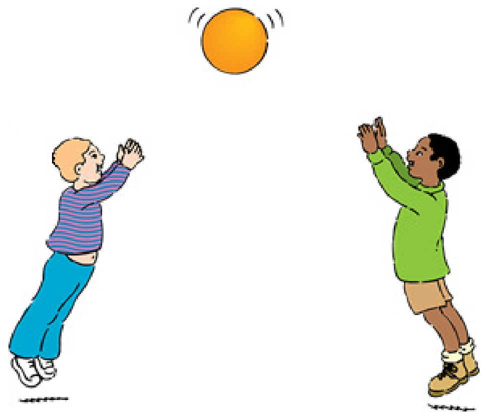
ChuckleBrothers

Introduction

This document describes an ECOA® to-and-fro communications example, given the name "Chuckle Brothers".

"Chuckle Brothers" demonstrates two entities passing something between each other: one will send it, and the other will pass it back. Like children throwing a ball backwards and forwards.

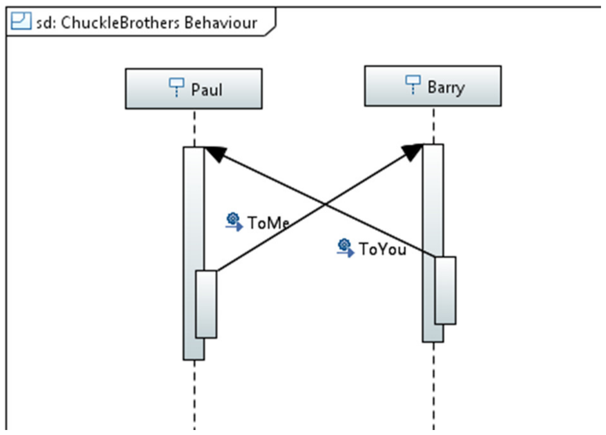
Figure 1 "To You" – "To Me"



In computing terms, the children (let's call them Barry and Paul) "become" two programs, the ball becomes a data item or message, and throwing the ball becomes the invocation of an operation to send the message either one way (let's call that operation "To You") or the other ("To Me").

In UML, the behaviour might be shown as a sequence diagram like Figure 2, where the "Barry" program invokes "ToYou" sending a message to the "Paul" program, which replies by invoking the "ToMe" operation, which triggers "Barry" to invoke "ToYou", and so on:

Figure 2 "*ChuckleBrothers*" Behaviour as a UML Sequence Diagram



This document presents the principal user generated artefacts required to create the two "Chuckle Brothers" programs using the ECOA. It is assumed that the reader is 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.

Aims

This ECOA "*Chuckle Brothers*" example is intended to demonstrate a minimum effort ECOA software system, comprising two ECOA ASCs (components) (ref.[1]), built as two executables, and with a single ECOA Service**.**

ECOA Features Exhibited

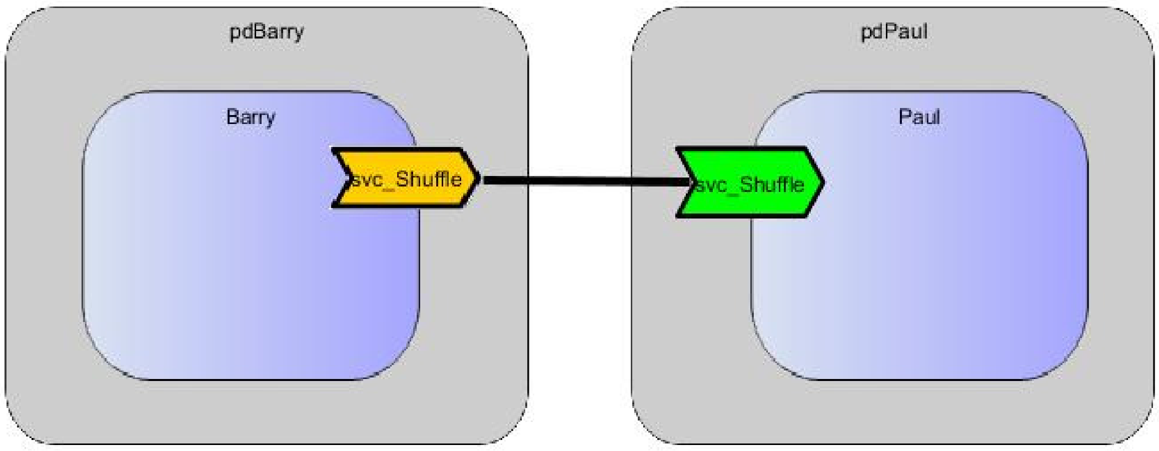
- Composition of an ECOA Assembly of multiple ECOA ASCs (components).
- Multiple cooperating ECPA Protection Domains.
- ECOA Logical Interface (ELI) between ECOA Protection Domains.
- Use of the ECOA runtime logging API.

Design and Definition

ECOA Assembly Design and Definition

This ECOA "*Chuckle Brothers*" example Assembly comprises two ECOA ASCs, named "*Barry*" and "*Paul*". Each ASC will be deployed into a separate ECOA Protection Domain. The ASCs will communicate through an ECOA Service, "*svc_Shuffle*", which *Paul provides*, and *Barry references*.

Figure 3 ECOA "*Chuckle Brothers*" 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 "*Chuckle Brothers*" Assembly is as follows (file *CB.impl.composite*):

```
<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="CB" targetNamespace="http://www.ecoa.technology">
  <csa:component name="Paul">
    <ecoa-sca:instance componentType="Paul">
      <ecoa-sca:implementation name="Paul"/>
    </ecoa-sca:instance>
    <csa:service name="svc_Shuffle"/>
  </csa:component>
  <csa:component name="Barry">
    <ecoa-sca:instance componentType="Barry">
      <ecoa-sca:implementation name="Barry"/>
    </ecoa-sca:instance>
    <csa:reference name="svc_Shuffle"/>
  </csa:component>
  <!-- System Wiring...  -->
  <csa:wire source="Barry/svc_Shuffle" target="Paul/svc_Shuffle"/>
</csa:composite>
```

Looking through this XML it will be seen how it is a mapping of the Assembly diagram, with each of the ASCs (Components) represented, and the Service link ("wire") connecting them.

The *Paul* ASC is defined in XML as follows (file *Paul.componentType*):

```
<componentType xmlns="http://docs.oasis-open.org/ns/opencsa/sca/200912"
      xmlns:xs="http://www.w3.org/2001/XMLSchema" xmlns:ecoa-
      sca="http://www.ecoa.technology/sca-extenion-2.0">
  <service name="svc_Shuffle">
    <ecoa-sca:interface syntax="svc_Shuffle"/>
  </service>
</componentType>
```

i.e. the ASC definition (the *<componentType>* XML element) declares the ECOA Service (using the *<service>* XML tag).

The *Barry* ASC is defined in XML as follows (file *Barry.componentType*):

```
<componentType xmlns="http://docs.oasis-open.org/ns/opencsa/sca/200912"
      xmlns:xs="http://www.w3.org/2001/XMLSchema" xmlns:ecoa-
      sca="http://www.ecoa.technology/sca-extenion-2.0">
  <reference name="svc_Shuffle">
    <ecoa-sca:interface syntax="svc_Shuffle"/>
  </reference>
</componentType>
```

i.e. almost identically to *Paul*, but with the Service *referenced* instead of declared.

ECOA Service and Types Definition

The *svc_Shuffle* Service, which is provided by the *Paul* ASC and referenced by the *Barry* ASC, is defined in a XML file (*svc_Shuffle.interface.xml*):

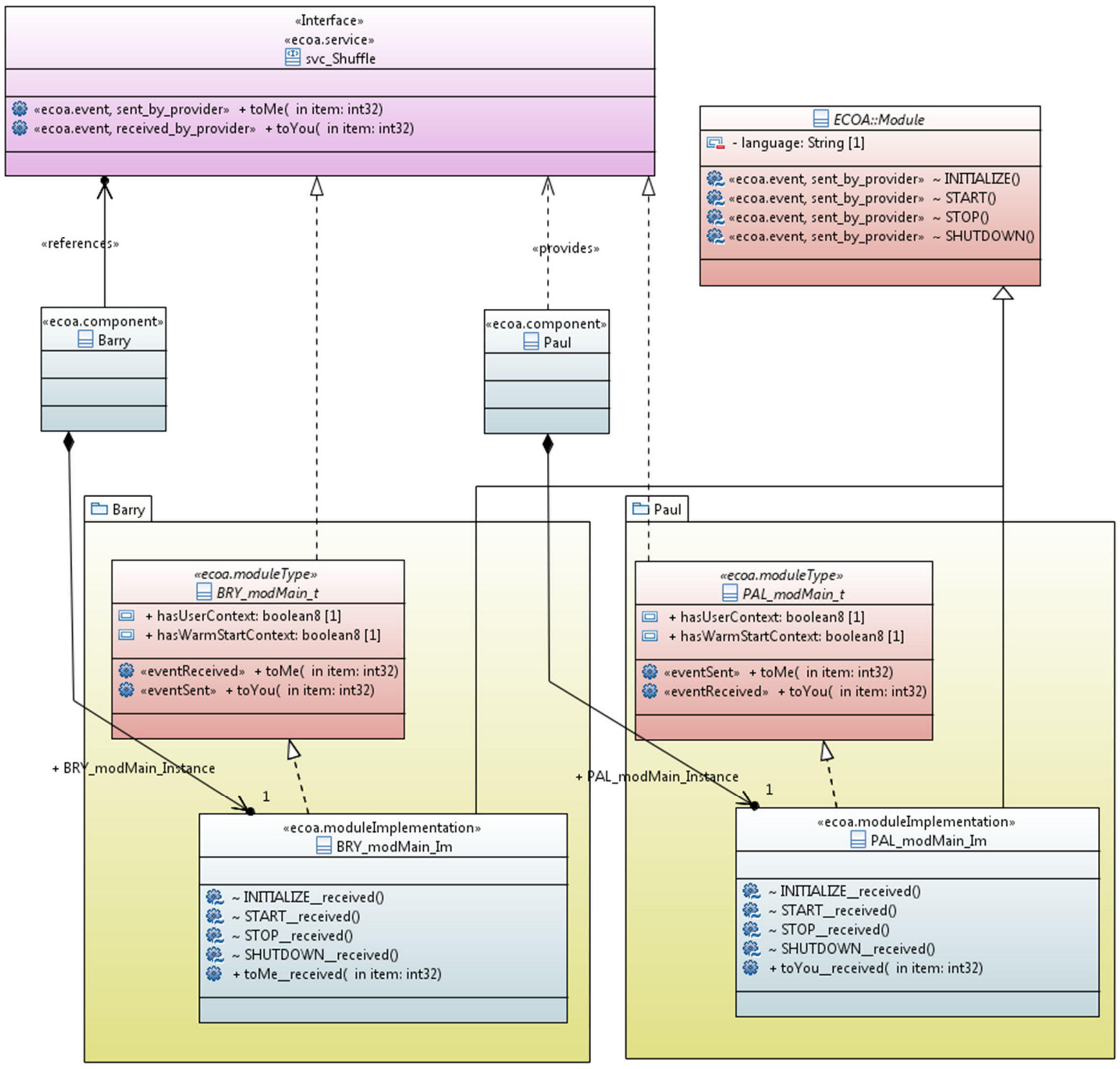
```
<serviceDefinition xmlns="http://www.ecoa.technology/interface-2.0">
  <operations>
    <event name="toMe" direction="SENT_BY_PROVIDER">
      <input name="item" type="int32" />
    </event>
    <event name="toYou" direction="RECEIVED_BY_PROVIDER">
      <input name="item" type="int32" />
    </event>
  </operations>
</serviceDefinition>
```

The Service comprises a pair of ECOA Event Operations called "*toMe*" and "*toYou*". "*toMe*" is sent by the Service provider, and "*toYou*" is *received* by the provider – that is sent by the referencer (client). Each Operation takes a single (integer value) parameter – the message content.

ECOA Module Design and Definition

The *Paul* and *Barry* ASCs are each composed of a single ECOA Module each (Module Implementations *PAL_modMain_Im* and *BRY_modMain_Im* of Module Types *PAL_modMain_t* and *BRY_modMain_t* respectively) as illustrated in UML in Figure 4. As always in the ECOA, the Module Implementation implements the Module Lifecycle operations defined by the ECOA (as represented in UML by the abstract class *ECOA::Module*), depicted in the diagram as an *inheritance* relationship.

Figure 4 ECOA "*Chuckle Brothers*" Module Design (as UML Class Diagram)


«Interface»
«ecoa.service»
svc_Shuffle
«ecoa.event, sent_by_provider» + toMe(in item: int32)
«ecoa.event, received_by_provider» + toYou(in item: int32)
ECOA::Module
- language: String [1]
«ecoa.event, sent_by_provider» ~ INITIALIZE()
«ecoa.event, sent_by_provider» ~ START()
«ecoa.event, sent_by_provider» ~ STOP()
«ecoa.event, sent_by_provider» ~ SHUTDOWN()
«references»
«provides»
«ecoa.component» Barry
«ecoa.component» Paul
Barry
Paul
«ecoa.moduleType» BRY_modMain_t
+ hasUserContext: boolean8 [1]
+ hasWarmStartContext: boolean8 [1]
«eventReceived» + toMe(in item: int32)
«eventSent» + toYou(in item: int32)
«ecoa.moduleType» PAL_modMain_t
+ hasUserContext: boolean8 [1]
+ hasWarmStartContext: boolean8 [1]
«eventSent» + toMe(in item: int32)
«eventReceived» + toYou(in item: int32)
+ BRY_modMain_Instance
1
+ PAL_modMain_Instance
1
«ecoa.moduleImplementation» BRY_modMain_Im
~ INITIALIZE__received()
~ START__received()
~ STOP__received()
~ SHUTDOWN__received()
+ toMe__received(in item: int32)
«ecoa.moduleImplementation» PAL_modMain_Im
~ INITIALIZE__received()
~ START__received()
~ STOP__received()
~ SHUTDOWN__received()
+ toYou__received(in item: int32)

The Paul ASC (ECOA) Implementation

The *Paul* ASC is declared in XML as follows, in what the ECOA calls the "Component Implementation" XML (file *PAL.impl.xml*):

```
<componentImplementation
                 xmlns="http://www.ecoa.technology/implementation-2.0"
                 componentDefinition="Paul">
  <use library="ECOA" />
  <!-- module PAL_modMain_t type definition -->
  <moduleType name="PAL_modMain_t" hasUserContext="true" hasWarmStartContext="false">
    <operations>
      <eventReceived name="toYou">
        <input name="item" type="int32" />
      </eventReceived>
```

```
          <eventSent name="toMe">
            <input name="item" type="int32" />
          </eventSent>
        </operations>
      </moduleType>
      <!-- module implementation definition -->
      <moduleImplementation    name="PAL_modMain_Im"  moduleType="PAL_modMain_t"
                              language="C" />
      <!-- module instances -->
      <moduleInstance name="PAL_modMain_Instance" implementationName="PAL_modMain_Im"
                              relativePriority="1">
      </moduleInstance>
      <!-- Definition of module operation links -->
      <eventLink>
        <senders>
          <moduleInstance instanceName="PAL_modMain_Instance" operationName="toMe"/>
        </senders>
        <receivers>
          <service instanceName="svc_Shuffle" operationName="toMe"/>
        </receivers>
      </eventLink>
      <eventLink>
        <senders>
          <service instanceName="svc_Shuffle" operationName="toYou"/>
        </senders>
        <receivers>
          <moduleInstance instanceName="PAL_modMain_Instance" operationName="toYou"/>
        </receivers>
      </eventLink>
    </componentImplementation>
```

Note that a naming convention has been adopted in this example whereby the Module is conceived as being declared within a code package named "*PAL*" (for "*Paul*"). This packaging is illustrated explicitly in the UML class diagram, and expressed in the XML (and subsequently in actual code) using the prefix "*PAL_*".

So, a Module Type (*PAL_modMain_t*) is declared which an *eventReceived* operation "*toYou*" and an *eventSent* operation "*toMe*", declarations inherited from the ECOA Service. This Module Type is implemented by a concrete Module Implementation *PAL_modMain_Im* (depicted in the UML expanded in the form of the code class produced by the code generation process), which in turn is instantiated at runtime as the Module Instance *PAL_modMain_Instance*.

From these definitions and declarations, a single functional code unit will be produced by the code generation process, implementing in code the single concrete class on the UML diagram (*PAL_modMain_Im*), and named "*PAL_modMain_Im.c*" (assuming the Module Implementation declaration has set the *language* property to "C").

The Barry ASC (ECOA) Implementation

The *Barry* ASC is declared in XML as follows, in what the ECOA calls the "Component Implementation" XML (file *BRY.impl.xml*):

```
<componentImplementation
    xmlns="http://www.ecoa.technology/implementation-2.0"
    componentDefinition="Barry">
    <use library="ECOA" />
  <!-- module BRY_modMain_t type definition -->
    <moduleType name="BRY_modMain_t" hasUserContext="false"
                hasWarmStartContext="false">
    <operations>
      <eventReceived name="toMe">
        <input name="item" type="int32" />
      </eventReceived>
      <eventSent name="toYou">
        <input name="item" type="int32" />
      </eventSent>
    </operations>
  </moduleType>
  <!-- module implementation definition -->
  <moduleImplementation  name="BRY_modMain_Im"
                moduleType="BRY_modMain_t" language="C" />
  <!-- module instances -->
    <moduleInstance name="BRY_modMain_Instance"
                implementationName="BRY_modMain_Im"
                relativePriority="1">
    </moduleInstance>
  <!-- Definition of module operation links -->
  <eventLink>
    <senders>
      <reference instanceName="svc_Shuffle" operationName="toMe"/>
    </senders>
    <receivers>
      <moduleInstance instanceName="BRY_modMain_Instance" operationName="toMe"/>
    </receivers>
  </eventLink>
  <eventLink>
    <senders>
      <moduleInstance instanceName="BRY_modMain_Instance" operationName="toYou"/>
    </senders>
    <receivers>
      <reference instanceName="svc_Shuffle" operationName="toYou"/>
    </receivers>
  </eventLink>
</componentImplementation>
```

You will note how it is similar to the *Paul* Component Implementation except where the name prefix has changed (from *PAL* to *BRY*), and where Module Operation directions have reversed.

So, a Module Type (*BRY_modMain_t*) is declared which an *eventSent*operation "*toYou*" and an *eventTeceived* operation "*toMe*", declarations inherited from the ECOA Service. This Module Type is implemented by a concrete Module Implementation *BRY_modMain_Im* (likewise depicted in the UML expanded in the form of the code class produced by the code generation process), which in turn is instantiated at runtime as the Module Instance *BRY_modMain_Instance*.

From these definitions and declarations, a single functional code unit will be produced by the code generation process, implementing in code the single concrete class on the UML diagram (*BRY_modMain_Im*), and named "*BRY_modMain_Im.c*" (assuming the Module Implementation declaration has set the *language* property to "C").

ECOA Deployment Definition

The ECOA "*Chuckle Brothers*" Assembly is deployed (that is, the declared Module Instances are allocated to ECOA Protection Domains, which are themselves allocated to computing nodes) by the following XML (file *CB.deployment.xml*):

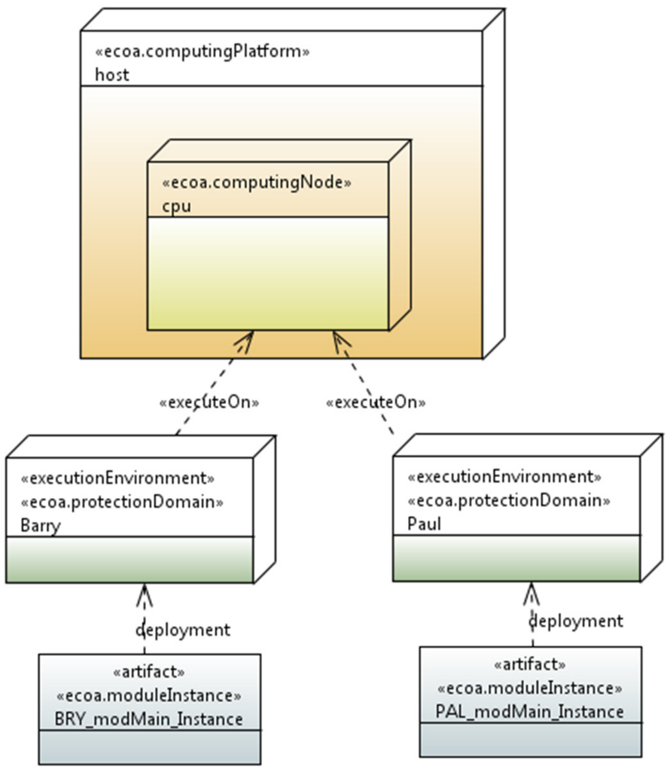
```
<deployment xmlns="http://www.ecoa.technology/deployment-2.0" finalAssembly="CB"
                 logicalSystem="hostbased">
  <protectionDomain name="pdBarry">
    <executeOn computingNode="cpu" computingPlatform="host"/>
    <deployedModuleInstance componentName="Barry"
                 moduleInstanceName="BRY_modMain_Instance" modulePriority="50"/>
  </protectionDomain>
  <protectionDomain name="pdPaul">
    <executeOn computingNode="cpu" computingPlatform="host"/>
    <deployedModuleInstance componentName="Paul"
                 moduleInstanceName="PAL_modMain_Instance" modulePriority="50"/>
  </protectionDomain>
   <platformConfiguration
                 faultHandlerNotificationMaxNumber="8"
                 computingPlatform="host" />
</deployment>
```

Thus in this case, the two Module Instances (*PAL_modMain_Instance* and *BRY_modMain_Instance*), which are the runtime manifestation of the *Paul* and *Barry* ASCs, are deployed into an ECOA Protection Domain each, *pdPaul* and *pdBarry* respectively, both executing on the Computing Node *cpu*, which is part of the (possibly multi-Node) ECOA Computing Platform *host*.

An alternate deployment might have the two Protection Domains hosted on separate ECOA Computing Platforms, for example "*myPC*" and "*yourPC*":

```
<deployment xmlns="http://www.ecoa.technology/deployment-2.0" finalAssembly="CB"
                 logicalSystem="hostbased">
  <protectionDomain name="pdBarry">
    <executeOn computingNode="cpu" computingPlatform="myPC"/>
    <deployedModuleInstance componentName="Barry"
                 moduleInstanceName="BRY_modMain_Instance" modulePriority="50"/>
  </protectionDomain>
  <protectionDomain name="pdPaul">
    <executeOn computingNode="cpu" computingPlatform="yourPC"/>
    <deployedModuleInstance componentName="Paul"
                 moduleInstanceName="PAL_modMain_Instance" modulePriority="50"/>
  </protectionDomain>
   <platformConfiguration
                 faultHandlerNotificationMaxNumber="8"
                 computingPlatform="host" />
</deployment>
```

Figure 5 ECOA "*Chuckle Brothers*" Single-host Deployment



Service Availability Considerations

Since the *Paul* ASC *provides* an ECOA Service (*svc_Shuffle*) it might be useful that the Service be declared (at runtime) as "available". Clients of the Service (e.g. *Barry*) could then check and take alternate action if the Service is not currently being provided. In the present simple example, availability of the *svc_Shuffle* Service has not been implemented – it is just assumed to be available once the Protection Domains (executables) have both started.

Implementation

The ECOA "*Chuckle Brothers*" example is sufficiently simple that the only software code that needs to be added to the code generated frameworks is to send and print out the actual messages.

As shown in Figure 2, the behaviour we want is that when the *Barry* ASC receives the "*ToMe*" message it sends back the "*ToYou*" message. And when the *Paul* ASC receives the "*ToYou*" message it sends back the "*ToMe*" message. The Component Implementation XML for the *Barry* ASC specifies that when the *toMe* ECOA Event (message) is received the *toMe* Module Operation (of Module Type *BRY_modMain_t*) is invoked. That Operation can therefore be coded (in the (C) code unit *BRY_modMain_Im.c*) as:

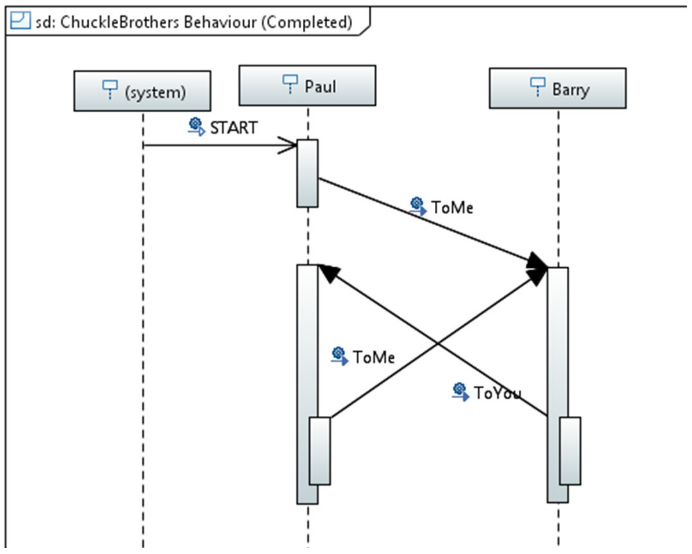
```
void BRY_modMain_Im__toMe__received
   (BRY_modMain_Im__context* context,
    const ECOA__int32 item)
{
      ECOA__log msg;
      /*
       * Wait a moment, then send toYou */
      nanosleep( &ONESEC, NULL );
      BRY_modMain_Im_container__toYou__send( context, item );
      /**/
      msg.current_size = sprintf( msg.data, "\n\tTo You" );
      BRY_modMain_Im_container__log_info( context, msg );
}
```

Likewise, when the *Paul* ASC receives the *toYou* ECOA Event (message), the *toYou* Module Operation (of Module Type *PAL_modMain_t*) is invoked, which can therefore be coded (in the (C) code unit *PAL_modMain_Im.c*) as:

```
void PAL_modMain_Im__toYou__received
   (PAL_modMain_Im__context* context,
    const ECOA__int32 item)
{
      ECOA__log msg;
      //
      context->user.BarryIsAnswering = !0;
      nanosleep( &ONESEC, NULL );
      context->user.SequenceNumber += 1;
      PAL_modMain_Im_container__toMe__send( context, context->user.SequenceNumber );
      //
      msg.current_size = sprintf( msg.data, "\n\tTo Me" );
      PAL_modMain_Im_container__log_info( context, msg );
}
```

In order to kick-start this to-and-fro process though, we need to send one of these two messages on start-up, independently of having not received the other message, adapting the behaviour as in :

Figure 6 Modified "*ChuckleBrothers*" Behaviour



In accordance with the ECOA definition, and as illustrated in Figure 4 by the UML *Generalization* (inheritance) of the *ECOA::Module* abstract class, each Module Implementation is code generated with four stub Module Operation functions, one each for the four ECOA Module Lifecycle operations (*INITIALIZE*, *START*, *STOP*, and *SHUTDOWN*). For the ECOA "*Chuckle Brothers*" example, we want a message to be sent when the whole Assembly starts operating, which is to say, when one of the *START* operations is called. As the *Paul* ASC is the Service provider, it's *START* operation (in Module Implementation *PAL_modMain_Im*), implemented by the (C) code function *PAL_modMain_Im-__START_received* in the (C) code unit *PAL_modMain_Im.c*, becomes:

```
void PAL_modMain_Im__START__received
   (PAL_modMain_Im__context* context)
{
      ECOA__log msg;
      msg.current_size = sprintf( msg.data, "\n\tPaul is ready" );
      PAL_modMain_Im_container__log_info( context, msg );
      //
      // Don't (MUSN'T) be too quick to start sending...
      nanosleep( &ONESEC, NULL );
      PAL_modMain_Im_container__toMe__send( context, context->user.SequenceNumber );
      //
      msg.current_size = sprintf( msg.data, "\n\tTo Me" );
      PAL_modMain_Im_container__log_info( context, msg );
}
```

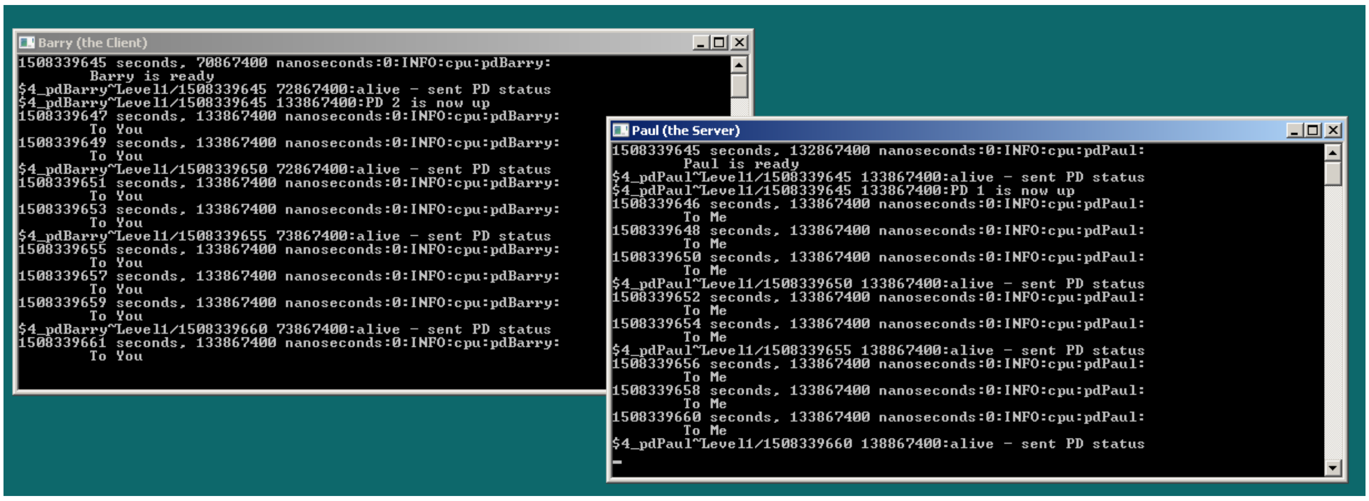
The functions listed all uses the ECOA logging API (*log_info*) to output a message to the user. The message is constructed (as variable *msg* of type *ECOA__log*) by using *sprintf* to copy the message text ("*Hello ECOA World*") into the *msg* variable's *data* field, and set its *current_length* field. *msg* is then passed to the ECOA *log_info* API.

The *Paul* ASC implementation utilizes the User Context capability provided for ECOA Modules (see ref.[1]). The *PAL_modMain_Im* Module Implementation maintains a *SequenceNumber* which is incremented each time the *toYou* message (ECOA Event) is sent to the *Barry* ASC. This *SequenceNumber* is held in the User Context so that the value persists between one invocation of the *toMe* operation and the next.

Program Output

When the ECOA "*Chuckle Brothers*" Assembly is built and run (as two Protection Domains (executables), an output similar to Figure 7 should be achieved. The *Log_info* information messages (mentioned earlier) are output to each programs console window, along with ECOA Platform logging messages (such as the 10 second periodic "alive" message):

Figure 7 ECOA "*Chuckle Brothers*" in Execution



```
1508339645 seconds, 70867400 nanoseconds:0:INFO:cpu:pdBarry:
        Barry is ready
$4_pdBarry~Level1/1508339645 72867400:alive - sent PD status
$4_pdBarry~Level1/1508339645 133867400:PD 2 is now up
1508339647 seconds, 133867400 nanoseconds:0:INFO:cpu:pdBarry:
        To You
1508339649 seconds, 133867400 nanoseconds:0:INFO:cpu:pdBarry:
        To You
$4_pdBarry~Level1/1508339650 72867400:alive - sent PD status
1508339651 seconds, 133867400 nanoseconds:0:INFO:cpu:pdBarry:
        To You
1508339653 seconds, 133867400 nanoseconds:0:INFO:cpu:pdBarry:
        To You
$4_pdBarry~Level1/1508339655 73867400:alive - sent PD status
1508339655 seconds, 133867400 nanoseconds:0:INFO:cpu:pdBarry:
        To You
1508339657 seconds, 133867400 nanoseconds:0:INFO:cpu:pdBarry:
        To You
1508339659 seconds, 133867400 nanoseconds:0:INFO:cpu:pdBarry:
        To You
$4_pdBarry~Level1/1508339660 73867400:alive - sent PD status
1508339661 seconds, 133867400 nanoseconds:0:INFO:cpu:pdBarry:
        To You
```

```
1508339645 seconds, 132867400 nanoseconds:0:INFO:cpu:pdPaul:
        Paul is ready
$4_pdPaul~Level1/1508339645 133867400:alive - sent PD status
$4_pdPaul~Level1/1508339645 133867400:PD 1 is now up
1508339646 seconds, 133867400 nanoseconds:0:INFO:cpu:pdPaul:
        To Me
1508339648 seconds, 133867400 nanoseconds:0:INFO:cpu:pdPaul:
        To Me
1508339650 seconds, 133867400 nanoseconds:0:INFO:cpu:pdPaul:
        To Me
$4_pdPaul~Level1/1508339650 133867400:alive - sent PD status
1508339652 seconds, 133867400 nanoseconds:0:INFO:cpu:pdPaul:
        To Me
1508339654 seconds, 133867400 nanoseconds:0:INFO:cpu:pdPaul:
        To Me
$4_pdPaul~Level1/1508339655 138867400:alive - sent PD status
1508339656 seconds, 133867400 nanoseconds:0:INFO:cpu:pdPaul:
        To Me
1508339658 seconds, 133867400 nanoseconds:0:INFO:cpu:pdPaul:
        To Me
1508339660 seconds, 133867400 nanoseconds:0:INFO:cpu:pdPaul:
        To Me
$4_pdPaul~Level1/1508339660 138867400:alive - sent PD status
_
```

References

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