BAE SYSTEMS
INSPIRED WORK

Dining Philosophers

Introduction

This document describes an ECOA® implementation example of the famous "Dining Philosophers" problem (ref.[2]).

"Dining Philosophers" is an often used example in concurrent programming design, addressing resource contention and synchronization issues.

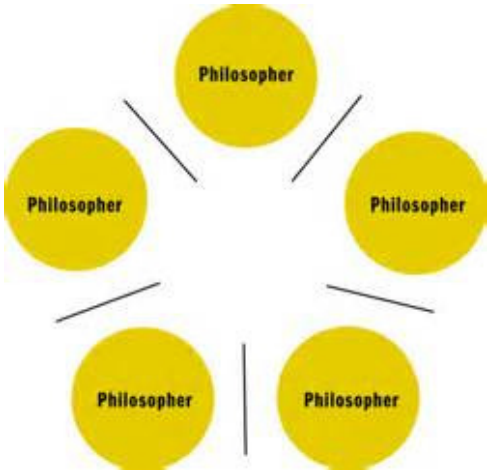
Five silent[1] philosophers sit at a round table with bowls of noodles. A chopstick is placed between each pair of adjacent philosophers.

Each philosopher must alternately think and eat. However, a philosopher can only eat noodles when he has both "left" and "right" chopsticks. Each chopstick can be held by only one philosopher and so a philosopher can use the chopstick only if it is not being used by another philosopher. After he finishes eating, he needs to put down both chopsticks so they become available to others. A philosopher can take the chopstick on his right or the one on his left as they become available, but cannot start eating before getting both of them. Nor can he take a chopstick that is not immediately on his left or right.

Eating is not limited by the remaining amounts of noodle or stomach space; an infinite supply is assumed!

The problem is how to design a discipline of behaviour (a *concurrent algorithm*) such that no philosopher will starve; i.e. each can forever continue to alternate between eating and thinking, assuming that no philosopher can know when others may want to eat or think.

Figure 1 The Dining Philosophers



[1] The philosophers cannot communicate with each other.

This document presents the principal user generated artefacts required to create a "Dining Philosophers" example 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 "*Dining Philosophers*" example is intended to demonstrate how ECOA concepts of concurrency and inversion of control (see ref.[1]) ease and facilitate the design and implementation of multi-threaded, multi-processing, applications.

ECOA Features Exhibited

- Composition of an ECOA Assembly of multiple ECOA ASCs (components).
- Contention-free resource sharing within an ECOA Assembly.
- Multiple cooperating ECOA Protection Domains.
- Service Availability.
- Module Lifecycle Management.
- Use of the ECOA runtime logging API.

Design and Definition

Resource Hierarchy Solution

The method of solving the resource contention issue in the Dining Philosophers problem, i.e. how to ensure that no philosopher starves because he cannot get both chopsticks at once, is Dijkstra's original "resource hierarchy" solution.

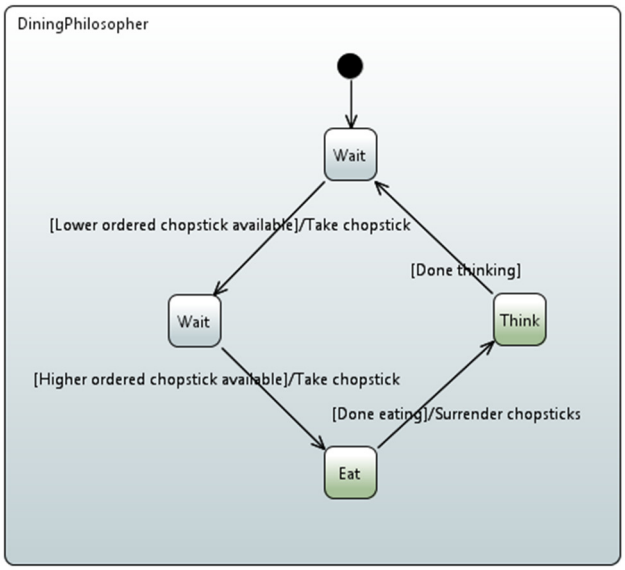
Each chopstick is assigned a "partial order" value ("0" to "4") (with no duplication) so each philosopher has a "lower ordered" chopstick on one side and a "higher ordered" chopstick on the other side. The solution derives from imposing the rule that the lower ordered chopstick must be picked up first.

So after a philosopher has finished thinking, rather than pick up the first chopstick to become available (of those on his left or right), he must wait until the lower ordered of those chopsticks is available, pick that up, then possibly wait again until the higher ordered chopstick becomes available.

Once he has both chopsticks, he eats, and when finished, surrenders each chopstick. He then thinks for a while, before getting hungry and starting over.

This solution is depicted (for one philosopher) in the UML Activity Diagram of Figure 2.

Figure 2 Resource Hierarchy Solution Applied to a Dining Philosopher

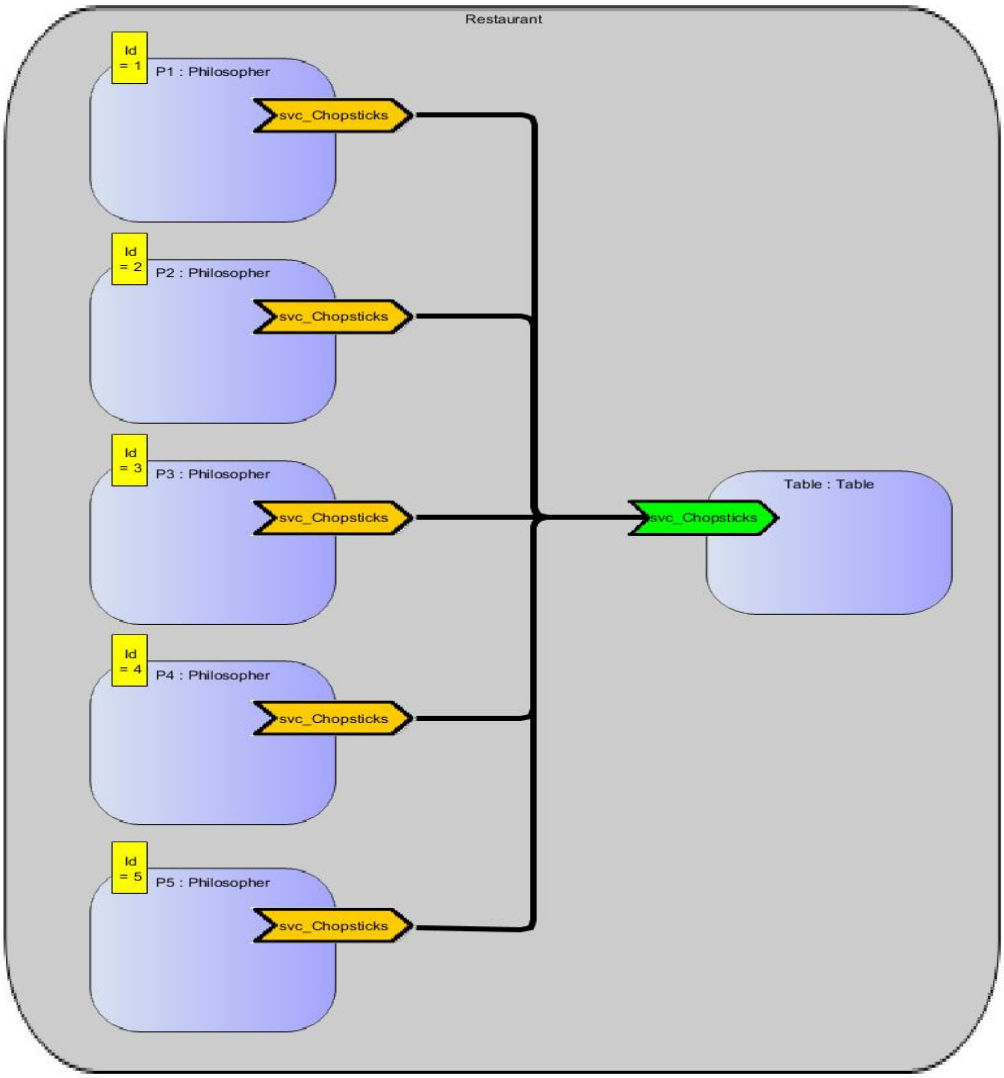


ECOA Assembly Design and Definition

This ECOA "*Dining Philosophers*" example is realized as an ECOA Assembly named "*Restaurant*" comprising five ECOA ASCs named "*P1*" to "*P5*" of the ASC type "*Philosopher*", and one ASC named "*Table*" of ASC type "*Table*". The "*Table*" ASC provides a "*svc_Chopsticks*" ECOA Service, which is referenced by the "*Philosopher*" ASCs, and by which each "*Philosopher*" can take and surrender chopsticks.

The ECOA "*Dining Philosophers*" (*Restaurant)* Assembly is depicted in Figure 3.

Figure 3 ECOA "*Dining Philosophers*" 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 "*Dining Philosophers*" (*Restaurant)* Assembly is as follows (file *Restuarant_impl.composite*), reflecting the Assembly diagram above:

```
<csa:composite xmlns:csa="http://docs.oasis-open.org/ns/opencsa/sca/200912"
      xmlns:ecoa-sca="http://www.ecoa.technology/sca" name="Restaurant_impl"
      targetNamespace="http://www.ecoa.technology/sca">
      <csa:component name="Table">
            <ecoa-sca:instance componentType="Table">
                  <ecoa-sca:implementation name="Table"/>
            </ecoa-sca:instance>
            <csa:service name="svc_Chopsticks"/>
      </csa:component>
```

```
        <!-- -->
        <csa:component name="P1">
              <ecoa-sca:instance componentType="Philosopher">
                    <ecoa-sca:implementation name="Philosopher"/>
              </ecoa-sca:instance>
              <csa:reference name="svc_Chopsticks"/>
              <csa:property name="Id"><csa:value>1</csa:value></csa:property>
        </csa:component>
          :
          : components P2 to P4 repeat
          :
        <csa:component name="P5">
              <ecoa-sca:instance componentType="Philosopher">
                    <ecoa-sca:implementation name="Philosopher"/>
              </ecoa-sca:instance>
              <csa:reference name="svc_Chopsticks"/>
              <csa:property name="Id"><csa:value>5</csa:value></csa:property>
        </csa:component>
        <!-- -->
        <!-- System Wiring...  -->
        <csa:wire source="P1/svc_Chopsticks" target="Table/svc_Chopsticks" ecoa-
                                                                  sca:rank="1"/>
        <csa:wire source="P2/svc_Chopsticks" target="Table/svc_Chopsticks" ecoa-
                                                                  sca:rank="1"/>
        <csa:wire source="P3/svc_Chopsticks" target="Table/svc_Chopsticks" ecoa-
                                                                  sca:rank="1"/>
        <csa:wire source="P4/svc_Chopsticks" target="Table/svc_Chopsticks" ecoa-
                                                                  sca:rank="1"/>
        <csa:wire source="P5/svc_Chopsticks" target="Table/svc_Chopsticks" ecoa-
                                                                  sca:rank="1"/>
  </csa:composite>
```

The *Table* ASC type is defined in XML as follows (file *Table.componcompoententType*):

```
<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">
      <service name="svc_Chopsticks">
            <ecoa-sca:interface syntax="svc_Chopsticks"/>
      </service>
</componentType>
```

That is, the ASC provides a single Service (*svc_Chopsticks*).

The *Philosopher* ASC type is defined in XML as follows (file *Philosopher.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">
      <reference name="svc_Chopsticks">
            <ecoa-sca:interface syntax="svc_Chopsticks"/>
      </reference>
      <property name="Id" type="xs:string" ecoa-sca:type="int32"/>
</componentType>
```

That is, in addition to declaring a *reference* to the *svc_Chopsticks* Service, the ASC defines an ECOA Property (*Id*) . Note that the Property *value* is given for each instance of the ASC in the *Restaurant* Assembly declaration (above), not here in the ASC type definition.

ECOA Service Definition

The *svc_Chopsticks* Service, which is provided by the *Table* ASC and referenced by the *Philosophers* ASCs, is defined in a XML file (*svc_Chopsticks.interface.xml*):

```
<serviceDefinition xmlns="http://www.ecoa.technology/interface-1.0"><!--
      name="svc_Chopsticks" -->
      <operations>
            <requestresponse name="take">
                  <input name="which" type="int32"/>
                  <input name="who" type="int32"/>
                  <output name="taken" type="boolean8"/>
            </requestresponse>
            <requestresponse name="surrender">
                  <input name="which" type="int32"/>
                  <input name="who" type="int32"/>
            </requestresponse>
      </operations>
</serviceDefinition>
```

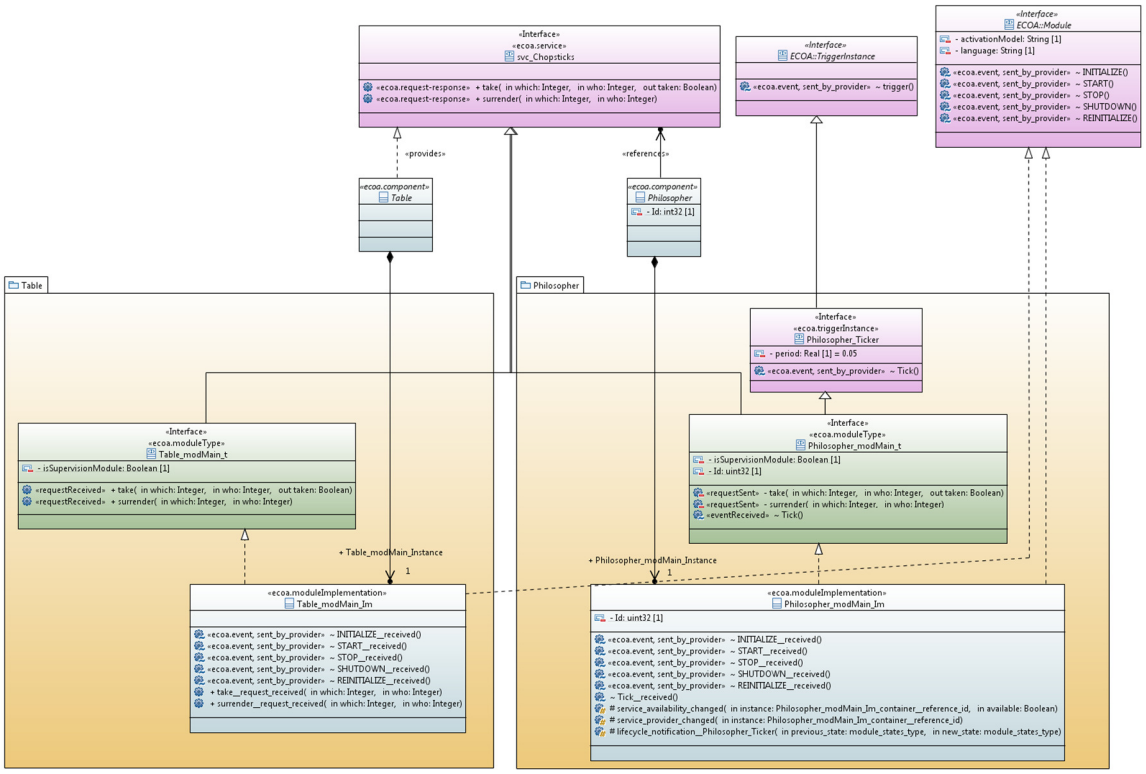
The Service comprise two ECOA Request-Response Operations, *take* and *surrender*, each of which takes two input parameters (*which* and *who*), whilst *take* also has a return parameter (*taken*). The parameter *which* specifies which chopstick the request is for, and is the partial order number for the chopstick ("0" to "4"). The *who* parameter specifies the philosopher's identity ("1" to "5") as given by its *Id* ECOA Property. The *taken* parameter will be *true* if the requested chopstick is available, or *false* if not.

Note that there is no mention or imposition at this declarative stage of the Resource Hierarchy algorithm, except to note that the chopsticks are identified by their numerical order value.

ECOA Module Design and Definition

The *Table* and *Philosopher* ASC (component) types are composed of a single ECOA Module each (Module Implementations *Table_modMain_Im* and *Philosopher_modMain_Im* of Module Types *Table_modMain_t* and *Philosopher_modMain_t* respectively) as illustrated in UML in Figure 4. Here is depicted in UML the *Table* ASC (component) ***providing*** the *svc_Chopsticks* ECOA Service, whilst the *Philosopher* ASC ***references*** the Service, and possesses the *Id* ECOA Property. As always in the ECOA, the Module Implementations implement the Module Lifecycle operations defined by the ECOA (as represented in UML by the interface class *ECOA::Module*).

Figure 4 ECOA "*Dining Philosophers*" Module Design (as UML Class Diagram)



The Table ASC

The *Table* ASC is declared in XML as follows (file *Table.impl.xml*):

```
<componentImplementation xmlns=http://www.ecoa.technology/implementation-1.0
      componentDefinition="Table">
      <use library="ECOA" />
      <!-- -->
      <moduleType name="Table_modMain_t" isSupervisionModule="true">
            <operations>
                  <requestReceived name="take">
                        <input name="which" type="int32" />
                              <input name="who" type="int32"/>
                        <output name="taken" type="boolean8"/>
                  </requestReceived>
                  <requestReceived name="surrender">
                        <input name="which" type="int32" />
                              <input name="who" type="int32"/>
                  </requestReceived>
            </operations>
      </moduleType>
      <!-- -->
      <moduleImplementation  name="Table_modMain_Im" moduleType="Table_modMain_t"
      activationModel="reactive" language="C" />
      <!-- -->
      <moduleInstance name="Table_modMain_Instance"
            implementationName="Table_modMain_Im" relativePriority="1">
      </moduleInstance>
```

```
            <!-- -->
            <requestLink>
                  <clients>
                        <service instanceName="svc_Chopsticks" operationName="take"/>
                  </clients>
                  <server>
                        <moduleInstance instanceName="Table_modMain_Instance"
                                                operationName="take"/>
                  </server>
            </requestLink>
            <!-- -->
            <requestLink>
                  <clients>
                        <service instanceName="svc_Chopsticks" operationName="surrender"/>
                  </clients>
                  <server>
                        <moduleInstance instanceName="Table_modMain_Instance"
                                                operationName="surrender"/>
                  </server>
            </requestLink>
      /componentImplementation>
```

That is, a Module Type (*Table_modMain_t*) is declared which has two *requestReceived* operations, "*take*" and "*surrender*", inherited from the *svc_Chopsticks* ECOA Service (depicted by the UML *generalization* association). This Module Type is implemented by a concrete Module Implementation *Table_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 *Table_modMain_Instance*.

The *<requestLink>* XML segments logically associate the specific concrete operations of the runtime Module Instance with the abstract Service operations.

A single functional code unit will be produced by the code generation process, implementing in code the concrete *Table_modMain_Im* class, and named "*Table_modMain_Im.c*" (assuming the Module Implementation declaration has set the *language* property to "C").

The Philosopher ASC

The *Philosopher* ASC is declared in XML as follows (file *Philosopher.impl.xml*):

```
<componentImplementation xmlns="http://www.ecoa.technology/implementation-
      1.0"componentDefinition="Philosopher">
      <use library="ECOA" />
      <moduleType name="Philosopher_modMain_t" isSupervisionModule="true">
            <properties>
                  <property name="Id" type="uint32"/>
            </properties>
            <operations>
                  <eventReceived name="Tick" />
                  <requestSent name="take" isSynchronous="true" timeout="-1.0">
                        <input name="which" type="int32" />
                        <input name="who" type="int32"/>
                        <output name="taken" type="boolean8"/>
                  </requestSent>
```

```
                <requestSent name="surrender" isSynchronous="true"
                                                        timeout="-1.0">
                    <input name="which" type="int32" />
                    <input name="who" type="int32"/>
                </requestSent>
            </operations>
        </moduleType>
        <!-- -->
        <moduleImplementation  name="Philosopher_modMain_Im"
                               moduleType="Philosopher_modMain_t"
                               activationModel="reactive"
                               language="C" />
        <!-- -->
        <moduleInstance name="Philosopher_modMain_Instance"
                        implementationName="Philosopher_modMain_Im"
                        relativePriority="1">
            <propertyValues>
                <propertyValue name="Id">$Id</propertyValue>
            </propertyValues>
        </moduleInstance>
        <!-- -->
        <triggerInstance name="Philosopher_Ticker" />
        <!-- -->
        <requestLink>
            <clients>
                <moduleInstance instanceName="Philosopher_modMain_Instance"
                                                    operationName="take"/>
            </clients>
            <server>
                <reference instanceName="svc_Chopsticks" operationName="take"/>
            </server>
        </requestLink>
        <requestLink>
            <clients>
                <moduleInstance instanceName="Philosopher_modMain_Instance"
                                                    operationName="surrender"/>
            </clients>
            <server>
                <reference instanceName="svc_Chopsticks" operationName="surrender"/>
            </server>
        </requestLink>
        <eventLink>
            <senders>
                <trigger instanceName="Philosopher_Ticker" period="0.05" />
            </senders>
            <receivers>
                <moduleInstance instanceName="Philosopher_modMain_Instance"
                                                    operationName="Tick"/>
            </receivers>
        </eventLink>
    </componentImplementation>
```

That is, a Module Type (*Philosopher_modMain_t*) is declared which has two *requestSent* operations, "*take*" and "*surrender*", inherited from the *svc_Chopsticks* ECOA Service (depicted by the UML *generalization* association), and an *eventReceived* operation named "*Tick*". This Module Type is implemented by a concrete Module Implementation *Philosopher_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 *Philosopher_modMain_Instance*.

The *Philosopher_Ticker* Trigger Instance is introduced because a periodic iterative polling behaviour is required to implement the philosopher implementation state machine (of which more later). Once every period (0.05 seconds as set in the *<eventLink>* XML[2]) the Trigger will fire and the Module Operation *Tick* will be invoked.

The Service Link (*<requestLink>* and *<eventLink>*) XML segments logically associate the specific concrete operations of the runtime Module Instance with the abstract Service operations, or in the case of the "Tick" operation, associates the concrete Module Operation to the Trigger Instance operation.

A single functional code unit will be produced by the code generation process, implementing in code the concrete *Philosopher_modMain_Im* class, and named "*Philosopher_modMain_Im.c*" (assuming the Module Implementation declaration has set the *language* property to "C").

ECOA Deployment Definition

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

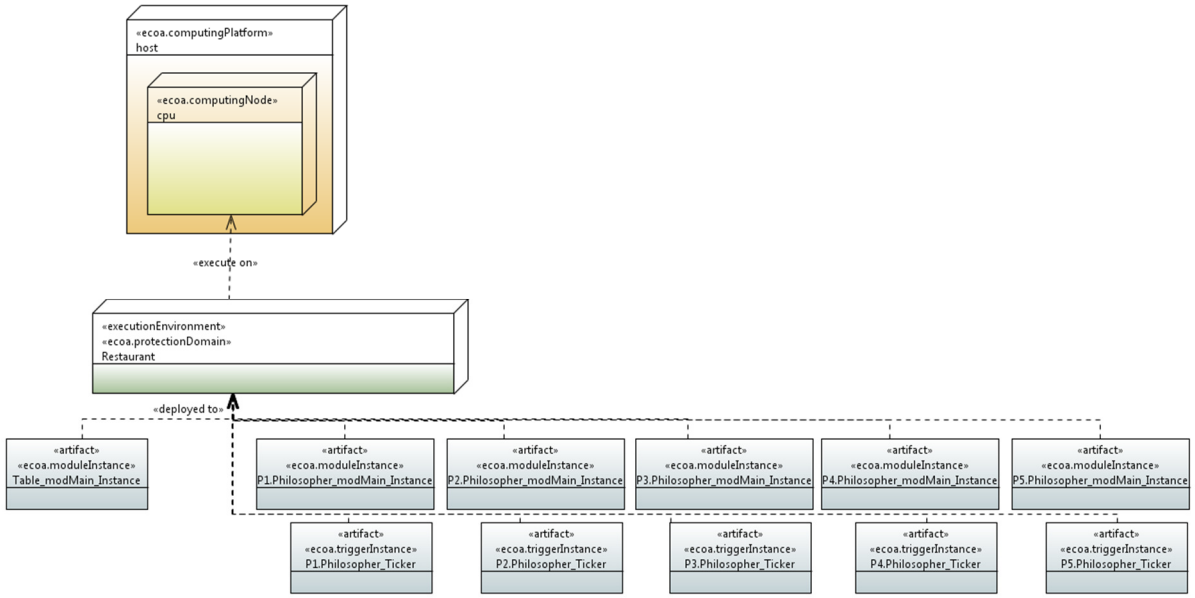
```
<deployment xmlns="http://www.ecoa.technology/deployment-1.0"
                              finalAssembly="Restaurant_impl"
                              logicalSystem="hostbased_logical_system">
      <protectionDomain name="Restaurant">
            <executeOn computingNode="cpu" computingPlatform="host"/>
            <deployedModuleInstance componentName="Table"
                              moduleInstanceName="Table_modMain_Instance"
                              modulePriority="52"/>
            <!-- -->
            <deployedModuleInstance componentName="P1"
                              moduleInstanceName="Philosopher_modMain_Instance"
                              modulePriority="50"/>
            <deployedTriggerInstance componentName="P1"
                              triggerInstanceName="Philosopher_Ticker"
                              triggerPriority="51"/>
            <!-- -->
            <deployedModuleInstance componentName="P2"
                              moduleInstanceName="Philosopher_modMain_Instance"
                              modulePriority="50"/>
            <deployedTriggerInstance componentName="P2"
                              triggerInstanceName="Philosopher_Ticker"
                              triggerPriority="51"/>
            <!-- -->
```

[2] The UML does not explicitly depict Service Links. The period attribute is therefore depicted as a UML property of the *«ecoa.triggerInstance»* UML interface class.

```
            <deployedModuleInstance componentName="P3"
                                    moduleInstanceName="Philosopher_modMain_Instance"
                                    modulePriority="50"/>
            <deployedTriggerInstance componentName="P3"
                                    triggerInstanceName="Philosopher_Ticker"
                                    triggerPriority="51"/>
            <!-- -->
            <deployedModuleInstance componentName="P4"
                                    moduleInstanceName="Philosopher_modMain_Instance"
                                    modulePriority="50"/>
            <deployedTriggerInstance componentName="P4"
                                    triggerInstanceName="Philosopher_Ticker"
                                    triggerPriority="51"/>
            <!-- -->
            <deployedModuleInstance componentName="P5"
                                    moduleInstanceName="Philosopher_modMain_Instance"
                                    modulePriority="50"/>
            <deployedTriggerInstance componentName="P5"
                                    triggerInstanceName="Philosopher_Ticker"
                                    triggerPriority="51"/>
        </protectionDomain>
        <platformConfiguration notificationMaxNumber="8"
                                    computingPlatform="host" />
    </deployment>
```

Thus in this case, a single ECOA Protection Domain is declared (*Restaurant*) executing on ECOA Computing Node *cpu*, in ECOA Computing Platform *host* (as represented as a UML Deployment Diagram in Figure 5).

Figure 5 ECOA "*Dining Philosophers*" (*Restaurant*) Assembly Deployment



A dummy UDP Transport Binding (file *udpbinding.xml*) is provided to satisfy the code generation process. It simply states the UDP local loop-back IP address…:

```
<ecoa:UDPBinding xmlns:ecoa="http://www.ecoa.technology/udpbinding-1.0">
      <ecoa:platform name="host"
            receivingMulticastAddress="127.0.0.1"
            receivingPort="60428"
            platformId="1"/>
</ecoa:UDPBinding>
```

ECOA Lifecycle and Service Availability Considerations

Since we have introduced a Trigger Instance into the *Philosopher* ASC, it is necessary that the Supervision Module (see ref.[1]) of that ASC (i.e. the Module Implementation *Philosopher_modMain_Im*) manage its lifecycle.

The management is generic and comprises:

- invoking an **INITIALIZE** ECOA Event Operation to the Trigger Instance to bring its Module Lifecycle state from **IDLE** to **READY**;
- invoking a **START** ECOA Event Operation to the Trigger Instance once its Module Lifecycle state is **READY.**

No other actions (such as invoking **STOP** or **SHUTDOWN**) are required in the present simple example.

Since the *Table* ASC *provides* an ECOA Service (*svc_Chopsticks*) it is necessary that the Service be declared (at runtime) as "available". For the present simple example, this will be done when the *Table* ASC's Supervision Module (namely the Module Implementation *Table_modMain_Im*) receives a **START** Event Operation. No error conditions are defined, so once set, the *svc_Chopsticks* Service will always be "available".

Implementation

In the implementation, philosophers are numerically identified "1" to "5". Chopsticks are numerically identified "0" to "4" – the modulus-to-base-5 of which is the partial ordering required for the Resource Hierarchy Solution, i.e.:

"0" < "1" < "2" < "3" < "4" < "0"

The Table ASC

The *Table* ASC provides the *svc_Chopsticks* Service using a trivial chopstick allocation algorithm:

```
if the requestedChopstick is free
      allocate requestedChopstick to requestingPhilospher
      return taken Boolean set true
otherwise
      return taken Boolean set false
```

There is no attempt to police whether a *requestingPhilosopher* "may" or "may not" request a particular chopstick. An error is raised if a philosopher tries to surrender a chopstick that is not allocated to him, including a chopstick that is already free.

All chopsticks are initially free, initialized by the *INITIALIZE* operation – implemented by the(C) code function *Table_modMain_Im__INITIALIZE__received* **in the code unit** *Table_modMain_Im.c*:

```
void Table_modMain_Im__INITIALIZE__received(
                                        Table_modMain_Im__context* context)
{
      int i;
      for( i = 0; i < 5; i++ ){
            context->user.Stick[i] = FREE;
      }
}
```

On invocation of the *START* operation, the *Table* ASC will set the Service Availability for the *svc_Chopsticks* ECOA Service, and announce its presence. That *START* operation is implemented by the (C) code function *Table_modMain_Im__START__received* in the code unit *Table_modMain_Im.c*:

```
void Table_modMain_Im__START__received(
                                        Table_modMain_Im__context* context)
{
      ECOA__log msg;
      ECOA__return_status erc;
      //
      erc = Table_modMain_Im_container__set_service_availability( context,
                  Table_modMain_Im_container__service_id__svc_Chopsticks,
                  ECOA__TRUE );
      //
      msg.current_size = sprintf( msg.data,
                  "\n\tThe Table is laid, the chopsticks are available...\n" );
      Table_modMain_Im_container__log_info( context, msg );
}
```

The "*take*" Service Operation is handled by the code function *Table_modMain_Im__take__request_received*. The function checks if the requested chopstick (*which*) is free, and if so allocates it to the requesting philosopher (*who*). The taken output parameter is set *true* (not zero) or *false* (zero) respectively. The code function completes the request-response transaction by invoking the *Table_modMain_Im_container__take__response_send* API function:

```
void Table_modMain_Im__take__request_received(
                                        Table_modMain_Im__context* context,
                                        const ECOA__uint32 ID,
                                        const ECOA__int32 which,
                                        const ECOA__int32 who)
{
      ECOA__boolean8 taken = 0;
      //
      if( context->user.Stick[which] == FREE ){
            taken = !0;
            context->user.Stick[which] = who;
      }else{
            taken = 0;
      }
      Table_modMain_Im_container__take__response_send( context, ID, taken );
}
```

The "surrender" Service Operation is handled by the code function *Table_modMain_Im__surrender__request_received*. The function checks if the nominated chopstick (*which*) is allocated to the philosopher (*who*). If it is, then the chopstick becomes free. Otherwise an error is signalled.

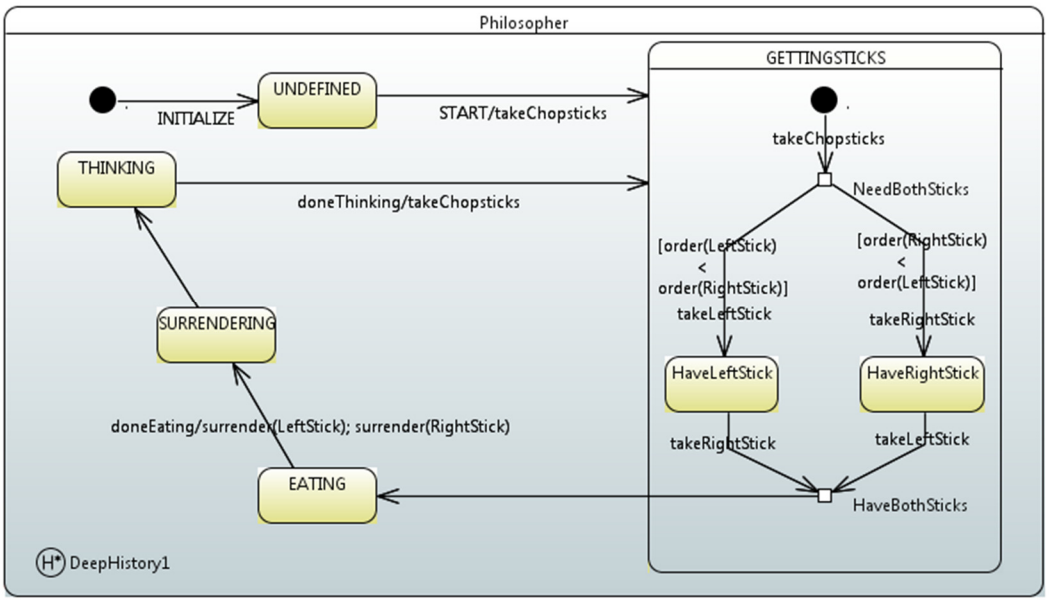
```
void Table_modMain_Im__surrender__request_received( Table_modMain_Im__context*
                                        context,
                                        const ECOA__uint32 ID,
                                        const ECOA__int32 which,
                                        const ECOA__int32 who)
{
      if( context->user.Stick[which] != who ){
            errno = EPERM; perror( "surrender" );
            fprintf( stderr,
                  "%d is trying to surrender chopstick %d held by %d...\n",
                  who, which, context->user.Stick[which] );
            exit(4);
      }else{
            context->user.Stick[which] = FREE;
      }
      Table_modMain_Im_container__surrender__response_send( context, ID );
}
```

The request-response transaction is completed by the code function by invoking the *Table_modMain_Im_container__surrender__response_send* API function. Note that the request-response transaction must be completed even though, in this case, there is no response data.

The Philosopher ASC

The implementation behaviour State Machine of the *Philosopher* ASC, in order to meet the Resource Hierarchy Solution, is depicted in Figure 6.

Figure 6 ECOA "*Dining Philosophers*" Implementation State Machine



On initialization (receipt of the ECOA Module Lifecycle *INITIALIZE* operation) the philosopher's state is "*UNDEFINED*". When the ECOA Module Lifecycle *START* operation is received, the philosopher's state is set to "*GETTINGSTICKS*". From there on, the state machine is free-running:

- when in the "*GETTINGSTICKS*" state, the *LeftStick* and *RightStick* are taken;
- the philosopher then enters the "*EATING*" state;
- when the philosopher has finished eating[3] the chopsticks are surrendered;
- the philosopher then enters the "*THINKING*" state;
- when the philosopher has finished thinking[4], he enters the "*GETTINGSTICKS*" state again.

The *INITIALIZE* Lifecycle operation is implemented by the code function *Philosopher_modMain_Im__INITIALIZE__received* of the (C) code unit *Philosopher_-modMain_Im.c*:, and simply initializes philosopher the state variables:

```
void Philosopher_modMain_Im__INITIALIZE__received
   (Philosopher_modMain_Im__context* context)
{
      context->user.PhiloState = philosopher__State_UNDEFINED;
      context->user.EatUntil   =
            context->user.ThinkUntil =
                  (ECOA__hr_time){ 0, 0 };
      context->user.HaveLeftStick  =
            context->user.HaveRightStick =
                  ECOA__FALSE;
}
```

[3] In this implementation the philosopher has finished eating after 7 seconds – he is VERY hungry!
[4] In this implementation the philosopher has finished thinking after just 11 seconds.

On invocation of the *START* operation, the *Philospher* ASC will initialize its *Philosopher_Ticker* Trigger Instance, as required by the ECOA (ref.[1]). That *START* operation is implemented by the code function *Philosopher_modMain_Im__START__received*:

```
void Philosopher_modMain_Im__START__received
   (Philosopher_modMain_Im__context* context)
{
      ECOA__uint32    IAm;
      ECOA__log       msg;
      //
      context->user.PhiloState = philosopher__State_GETTINGSTICKS;
      Philosopher_modMain_Im_container__INITIALIZE__Philosopher_Ticker( context );
      //
      Philosopher_modMain_Im_container__get_Id_value( context, &IAm );
      msg.current_size = sprintf( msg.data,
            "\n\tPhilosopher %d is ready...\n", IAm );
      Philosopher_modMain_Im_container__log_info( context, msg );
}
```

When a Module or Trigger Instance has changed Module Lifecycle state, the ECOA Software Platform issues a Lifecycle Notification to its parent Supervision Module. In the present case, the notification for the *Philosopher* ASC's *Philosopher_Ticker* Trigger Instance is implemented by the *Philosopher_modMain_Im__lifecycle_notification__Philosopher_Ticker* code function:

```
void Philosopher_modMain_Im__lifecycle_notification__Philosopher_Ticker
   (Philosopher_modMain_Im__context* context,
    ECOA__module_states_type previous_state,
    ECOA__module_states_type new_state)
{
      ECOA__uint32 IAm;
      ECOA__log    msg;
      //
      Philosopher_modMain_Im_container__get_Id_value( context, &IAm );
      //
      if( previous_state == ECOA__module_states_type_IDLE &&
                  new_state == ECOA__module_states_type_READY ){
            msg.current_size = sprintf( msg.data,
                  "\n\tPhilosopher %d Ticker is READY...\n", IAm );
            Philosopher_modMain_Im_container__log_info( context, msg );
            //
            Philosopher_modMain_Im_container__START__Philosopher_Ticker( context );
      }
      // No need to worry about other module state changes...
}
```

That is, if the *Philosopher_Ticker* Trigger Instance has transitioned from the **IDLE** to the **READY** state (i.e. it has been initialized), then it is started. Other Module Lifecycle state transitions, such as transitioning from the **READY** to the **RUNNING** state (as a result of a **START** operation) are ignored in this example.

Since we have made the decision not to change the Service Availability once it has been set by the *Table* ASC, the *Philosopher_modMain_Im__service_availability_changed* and *Philosopher_modMain_Im__service_provider_changed* function stubs can be left empty.

All we need to do now then is to program the remaining (free-running) part of the philosopher state machine. In order to preserve ECOA Inversion of Control principals, the state machine is implemented by sampling the state periodically, triggered by the *Philosopher_Ticker* ECOA Trigger Instance, and implemented in the code function *Philosopher_modMain_Im__Tick__received* (as on the pages following):

```
void Philosopher_modMain_Im__Tick__received(Philosopher_modMain_Im__context* context)
{
    ECOA__uint32     IAm;
    ECOA__log        msg;
    ECOA__return_status     erc;
    ECOA__hr_time    timeNow;
    ECOA__boolean8   Taken, available;
    ECOA__int32      LeftStick, RightStick;
    //
    Philosopher_modMain_Im_container__get_Id_value( context, &IAm );
    //
    LeftStick  = IAm - 1;// IAm is {1..5} therefore this is {0..4}
    RightStick = IAm % 5;// IAm is {1..5} therefore this is {1..4,0}
    //
    Philosopher_modMain_Im_container__get_relative_local_time( context, &timeNow );
    msg.current_size = sprintf( msg.data, "Philosopher %d Tick. State = %s", IAm, philosopher__State_value( context-
                                >user.PhiloState ) );
    Philosopher_modMain_Im_container__log_info( context, msg );
    //
    Philosopher_modMain_Im_container__get_service_availability( context,
                                Philosopher_modMain_Im_container__reference_id__svc_Chopsticks, &available);
    if( !available ){
        // No point in doing anything 'til the Chopsticks are available
        msg.current_size = sprintf( msg.data, "Philosopher %d Tick, but Chopsticks not available...", IAm );
        Philosopher_modMain_Im_container__log_info( context, msg );
        return;
    }
    //
```

```
if( context->user.PhiloState == philosopher__State_GETTINGSTICKS ){
    //  ***************************************
    //  * Dijkstra's Resource Hierarchy Solution.... *
    //  ***************************************
    // The lower numbered chopstick is always taken first...
    /**/
    if( RightStick < LeftStick ){
        if( !context->user.HaveRightStick ){
            if(( erc = Philosopher_modMain_Im_container__take__request_sync( context, RightStick, IAm, &Taken )) !=
                    ECOA__return_status_OK ){
                msg.current_size = sprintf( msg.data, "take__request( RightStick=>%d, by=>%d ) failed with %d",
                        RightStick, IAm, erc );
                Philosopher_modMain_Im_container__log_info( context, msg );
            }
            if( !Taken ){
                return; // We'll have another go on the next tick...
            }else{
                context->user.HaveRightStick = ECOA__TRUE;
            }
        }
    }
    if( !context->user.HaveLeftStick ){
        if(( erc = Philosopher_modMain_Im_container__take__request_sync( context, LeftStick, IAm, &Taken )) !=
                ECOA__return_status_OK ){
            msg.current_size = sprintf( msg.data, "take__request( LeftStick=>%d, by=>%d ) failed with %d", LeftStick,
                    IAm, erc );
            Philosopher_modMain_Im_container__log_info( context, msg );
        }
        if( !Taken ){
            return; // We'll have another go on the next tick...
        }else{
            context->user.HaveLeftStick = ECOA__TRUE;
        }
    }
    if( RightStick > LeftStick ){
        if( !context->user.HaveRightStick ){
            if(( erc = Philosopher_modMain_Im_container__take__request_sync( context, RightStick, IAm, &Taken )) !=
                    ECOA__return_status_OK ){
```

```
        if( !Taken ){
            return; // We'll have another go on the next tick...
        }else{
            context->user.HaveRightStick = ECOA__TRUE;
        }
    }
    printf( "%d begin eating.\n", IAm );
    context->user.PhiloState = philosopher__State_EATING;
    context->user.EatUntil = timeAdd( timeNow, EatPeriod );
}
//
if( context->user.PhiloState == philosopher__State_EATING ){
    if( timecmp( timeNow, context->user.EatUntil ) < 0 ){
        return; // Still chomping...
    }else{
        printf( "%d finished eating.\n", IAm );
        context->user.PhiloState = philosopher__State_SURRENDERING;
    }
}
//
```

```
if( context->user.PhiloState == philosopher__State_SURRENDERING ){
    if(( erc = Philosopher_modMain_Im_container__surrender__request_sync( context, LeftStick, IAm )) !=
                        ECOA__return_status_OK ){
        msg.current_size = sprintf( msg.data, "\n\tPhilosopher %d failed to surrender chopstick %d...\n", IAm, LeftStick );
        Philosopher_modMain_Im_container__log_warning( context, msg );
    }
    if(( erc = Philosopher_modMain_Im_container__surrender__request_sync( context, RightStick, IAm )) !=
                        ECOA__return_status_OK ){
        msg.current_size = sprintf( msg.data, "\n\tPhilosopher %d failed to surrender chopstick %d...\n", IAm, RightStick
                           );
        Philosopher_modMain_Im_container__log_warning( context, msg );
    }
    context->user.HaveLeftStick = context->user.HaveRightStick = ECOA__FALSE;
    printf( "%d begin thinking.\n", IAm );
    context->user.PhiloState = philosopher__State_THINKING;
    context->user.ThinkUntil = timeAdd( timeNow, ThinkPeriod );
}
//
if( context->user.PhiloState == philosopher__State_THINKING ){
    if( timecmp( timeNow, context->user.ThinkUntil ) < 0 ){
        return; // Still cogitating...
    }
    printf( "%d finished thinking.\n", IAm );
    context->user.PhiloState = philosopher__State_GETTINGSTICKS;
}
//
if( context->user.PhiloState <= philosopher__State_UNDEFINED ||
    context->user.PhiloState > philosopher__State_THINKING ){
    msg.current_size = sprintf( msg.data, "\n\tPhilosopher %d has illegal state %s...\n", IAm, philosopher__State_value(
                       context->user.PhiloState ));
    Philosopher_modMain_Im_container__log_info( context, msg );
    return;
}
```

The code function *Philosopher_modMain_Im__Tick__received* requires two constant values that are used to determine when the philosopher has had enough eating or thinking.

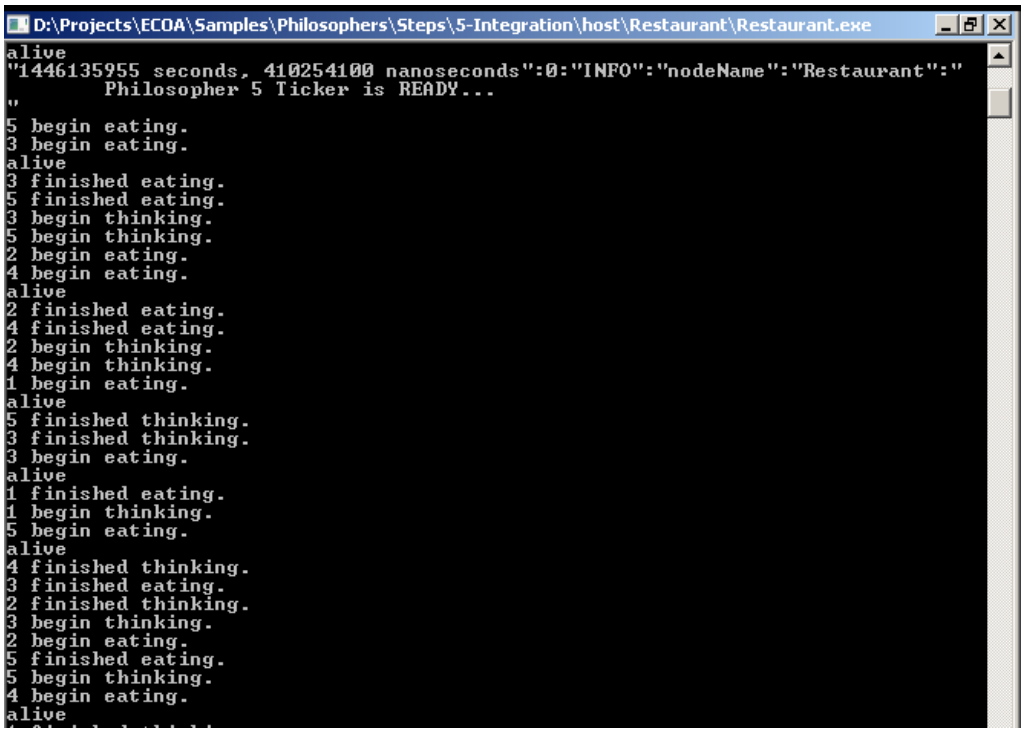
```
static const ECOA__hr_time EatPeriod   = {  7, 0 };
static const ECOA__hr_time ThinkPeriod = { 11, 0 };
```

In each case, the function records the time that the state is entered ("*EATING*" or "*THINKING*") and each time it is triggered it checks the current time against the recorded time. Only when the period set by the constant has passed will the state machine progress (either to the "*SURRENDERING*" or "*GETTINGSTICKS*" states of Figure 6).

Program Output

When the ECOA "*Dining Philosophers*" (*Restaurant)* Assembly is built and run, an output similar to Figure 7 should be achieved[5]. The *Philosopher* ASC start-up messages are output to the system console, prefixed by miscellaneous logging data (time stamp, logging type, etc.) (one of which is shown from "Philosopher 5" (i.e. ASC *P5* of the Assembly)). Each philosopher then outputs state changes, reporting his *Id* number and whether he is beginning or finishing eating or thinking. These outputs are interleaved with any other ECOA Platform logging messages (such as the 10 second periodic "alive" messages in the example shown):

Figure 7 ECOA "*Dining Philosophers*" (*Restaurant)* Assembly in Execution



[5] On Windows some ignorable error messages are output from the ECOA Platform code on start-up, due to the ECOA Platform code attempting to set thread naming attributes that Windows doesn't support. The functioning of the ECOA Platform is unaffected by these errors.

References

1	European Component Oriented Architecture (ECOA) Collaboration Programme: Architecture Specification (Parts 1 to 11) "ECOA" is a registered trade mark.
2	Dining philosophers problem Dijkstra, Hoare *https://en.wikipedia.org/wiki/Dining_philosophers_problem*
4	