Ejercicio de formalización de modelos de simulación

PID_00249441

Pau Fonseca Casas

Tiempo mínimo de dedicación recomendado: 1 hora

Universitat Oberta de Catalunya

Ninguna parte de esta publicación, incluido el diseño general y la cubierta, puede ser copiada, reproducida, almacenada o transmitida de ninguna forma, ni por ningún medio, sea éste eléctrico, químico, mecánico, óptico, grabación, fotocopia, o cualquier otro, sin la previa autorización escrita de los titulares del copyright.

Índice

1. Introducción 5

2. Combinar modelos de simulación 6
2.1. Gestión del reloj de simulación 6

3. Co-simulación en SDLPS 9
3.1. Punteros y tipos 9
3.2. Procedimientos 10
3.2.1. PROCEDURES integrados en el SDLPS 11
3.3. Código externo: SDLCodeExternal 12

4. Ficheros generados 13
4.1. Model file (.sdlps) 13
4.1.1. Utilidades para depurar la aplicación 14
4.2. Representation file (.eye) 15
4.3. SDL PROCESS File (.sdlprocess) 16
4.4. PARAMETRIZATION File (.param) 17
4.5. TRACE FILE (.trace) 17

5. El ejercicio 19

Bibliografía 21

1. Introducción

En este ejercicio se pretende revisar los principales conceptos aprendidos en el módulo de simulación. Para ello, seguiremos los siguientes artículos para la descripción de los agentes inteligentes:

Fonseca Casas, P.; Pi Palomés, X.; Casanovas Garcia, J.; Jové, J. (2013). «Definition of virtual reality simulation models using specification and description language diagrams». En: *SDL 2013: Model Driven Dependability Engineering* (págs. 258-274). Berlín / Heidelberg: Springer-Verlag.

Fonseca i Casas, P. (2014, mayo). «Using Specification and Description Language to Formalize Multiagent Systems». *Applied Artificial Intelligence* (vol. 28, núm. 5, págs. 504-531).

Y para un ejemplo clásico que presente formalizar el funcionamiento de un «steam-boiler», utilizaremos:

Boudjelal, F. A.; Lafaye, J.-Y.; Louis, G. (1997). «Specifying the Steam-Boiler Case Study with SDL». En: *SDL '97: Time for Testing* (págs. 491-505). Ámsterdam: Elsevier, 1997.

El objetivo del ejercicio es completar una especificación completa de SDL que pueda ser ejecutada en alguna de las herramientas que actualmente existen.

Las dos herramientas que se proponen para desarrollar la especificación son SDLPS (ver Fonseca i Casas, 2010) o *PragmaDev Studio* (2016).

El objetivo de usar estas herramientas no es tanto conseguir que el modelo se ejecute correctamente sino entender la sintaxis de SDL y cómo se puede combinar con otros sistemas a partir del concepto de co-simulación. En este manual se explicará genéricamente cómo usar SDLPS.

2. Combinar modelos de simulación

Existen diferentes mecanismos para combinar sistemas de simulación o físicos en un solo modelo. Las principales técnicas que existen son (Vangheluwe, 2000):

- Meta-formalismo: que implica la definición de un meta-formalismo al que traduciremos cualquier sistema definido en un lenguaje formal.

- Formalismo común: que implica definir mecanismos de transformación entre los diferentes subsistemas a un formalismo común que será el que se implementará.

- Co-Simulation: que implica el establecimiento de mecanismos de comunicación entre los diferentes subsistemas que seguirán usando su lenguaje de especificación y/o sus implementaciones propias.

En este trabajo nos centraremos en el mecanismo de co-simulación.

2.1. Gestión del reloj de simulación

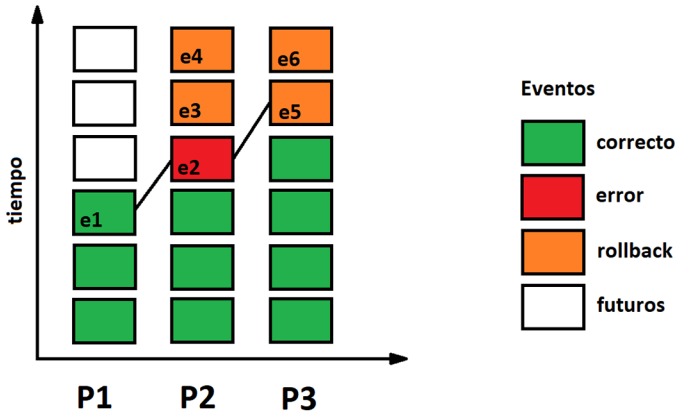
Uno de los elementos clave para definir un modelo usando la técnica de la co-simulación es establecer cómo se gestionará la evolución del reloj global del modelo. Usualmente, los sistemas definen un reloj global que representa la evolución temporal del sistema, viendo la simulación de forma completa, sin componentes.

El problema de la sincronización de los diferentes relojes parciales de cada subsistema que integramos en el co-simulador es evidente, dado que es posible que algún subsistema avance el reloj de simulación hasta un estado futuro. Este estado futuro podría verse afectado por un evento del pasado (estando ya en un estado futuro), lo que daría lugar a situaciones erróneas (ver figura 1). En este ejemplo, el evento e2 del proceso P2 depende del evento e1 del proceso P1 que se ejecuta más lentamente. P2 ha llevado a ejecutar hasta el evento e4, pero tendrá que efectuar un *rollback* de e4 y e3 dado que se han generado partiendo de e2, que dependía de e1, que ha sido ejecutado posteriormente. De la misma forma, los eventos e5 y e6 del proceso P3 también tendrán que ser reevaluados.

El mecanismo del Lower Bound Time Stamp (LBTS en HLA1.3 [ver Zeigler, Praehofer y Kim, 2000] o http://www.pitchtechnologies.com/hlatutorial, para descargar la API y los tutoriales que existen), permite establecer *federaciones* donde se determina cuándo es seguro avanzar sin necesidad de implementar un mecanismo de *rollback* para los diferentes componentes.

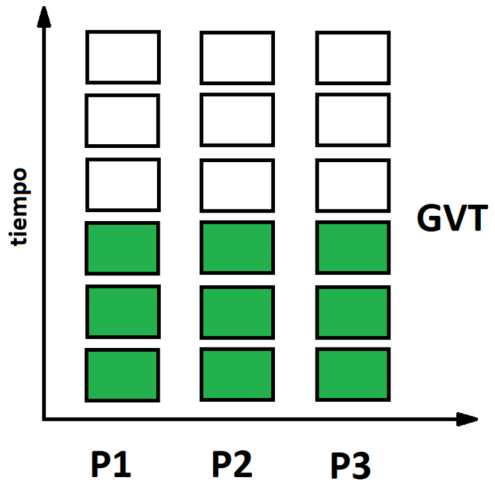
En el caso de SDL, el valor del reloj está guardado en el agente SYSTEM y es por tanto compartido con todos los elementos (AGENTES) del modelo, denominado GVT (Global Virtual Time), el tiempo menor de todo el sistema. Por tanto, cuando consideramos un modelo SDL, podemos entender que existe una variable global que contiene el tiempo del modelo de simulación, a la que podemos acceder a través de la instrucción TIME.

Figura 1. El problema de sincronizar diferentes procesos en un sistema de simulación.



Este tiempo es el que determina que todos los elementos federados (usando terminología HLA) pueden avanzar de forma segura sin necesidad de implementar mecanismos de *rollback*. En la figura 2 podemos ver el mismo ejemplo que en la figura 1, pero ahora P2 y P3 no avanzan hasta que el GVT no sea correcto.

Figura 2. Usando el GVT en el ejemplo para asegurar que no requeriremos hacer *rollback*.



En SDLPS se puede considerar que este tiempo está guardado en el agente SYSTEM, aunque podemos ver que cada AGENTE SDL tendrá un tiempo local que le permitirá gestionar sus eventos.

Hay que tener presente que, desde el punto de vista de la definición del modelo, no es necesario entender cómo funcionan los mecanismos de sincronización del sistema. SDL asegura que un modelo bien definido se podrá ejecutar de forma distribuida, y que cada agente podrá ejecutarse independientemente de los otros.

La gestión de la sincronización se divide en dos grandes familias:

- Conservadoras: no implementan mecanismos de *rollback*, por ejemplo, trabajan con el GVT.

- Optimistas: permiten que los procesos se sitúen en estados en los que para mantener la coherencia se haga necesario efectuar un *rollback*.

La sincronización optimista acostumbra a dar resultados, desde el punto de vista de la eficiencia del modelo, mejores. Es decir, usualmente los mecanismos optimistas tardan menos tiempo en ejecutar el modelo, no obstante, presentan una complejidad añadida al tener que implementar el mecanismo de *rollback*.

En este punto interesa insistir en el hecho de que el modelo SDL es independiente del mecanismo de sincronización implementado en el sistema que ejecutará el modelo. Este aspecto es importante dado que tenemos claramente una separación entre el modelo y su implementación.

Para saber más sobre este tema se puede consultar Fujimoto (2001) y Malik, Park y Fujimoto (2010).

3. Co-simulación en SDLPS

En el sistema SDLPS se emplea el mecanismo de co-simulación para combinar modelos de simulación. En esta sección describiremos los mecanismos que implementa y que nos serán de utilidad para completar el ejercicio.

El lenguaje que usaremos para conectar los diferentes subsistemas será C, si bien con algunas particularidades que describiremos a continuación.

3.1. Punteros y tipos

El código que usemos **no contendrá punteros**; esto se hace para simplificar la interacción entre los diferentes sistemas que implementan cada uno de los diferentes procesos. Por ejemplo, podría darse el caso de que el proceso SDL estuviera ejecutándose en una máquina concreta y se comunicara con otro proceso que estuviera ejecutándose en otra máquina. Obviamente, la estructura de la pila de memoria de cada máquina es independiente y, por tanto, pasar un puntero como parámetro puede representar un problema.

Dicho esto, los tipos permitidos son:

int, double, char*, int[#], double[#].

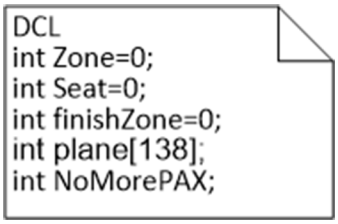
Como se puede ver, hay una excepción, que es para la definición de cadenas de caracteres, donde el puntero se usa simplemente para especificar que la cadena tiene una longitud no especificada.

La definición de vectores se haría de forma usual en C:

int myArray[100];

Dentro de los diagramas SDL se harían las declaraciones en los diagramas de proceso, tal como se muestra en la figura 3, donde tenemos definido un vector denominado *plane* de 138 posiciones.

Figura 3. Declaración de variables en SDLPS.

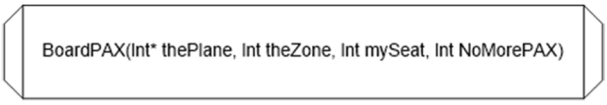


3.2. Procedimientos

Se puede usar código externo, que se incluirá en el modelo a través de los PROCEDURES. En este caso, el código externo no tiene limitaciones y permite conectar el modelo con cualquier sistema físico o simulado.

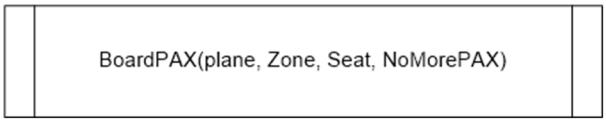
Para poder usar PROCEDURES hay que definirlos en los procesos que los usarán. En la figura 4 se puede ver la definición de un PROCEDURE denominado *BoardPAX*; en él estamos pasando un vector por referencia para que sea modificado dentro del PROCEDURE.

Figura 4. Definición de un PROCEDURE.



Una vez el procedimiento ha sido definido, puede ser usado en un PROCEDURECALL tal como se muestra en la figura 5. Todas las variables que se pasan al PROCEDURE, ya sea por referencia o por valor, han de definirse en un DCL.

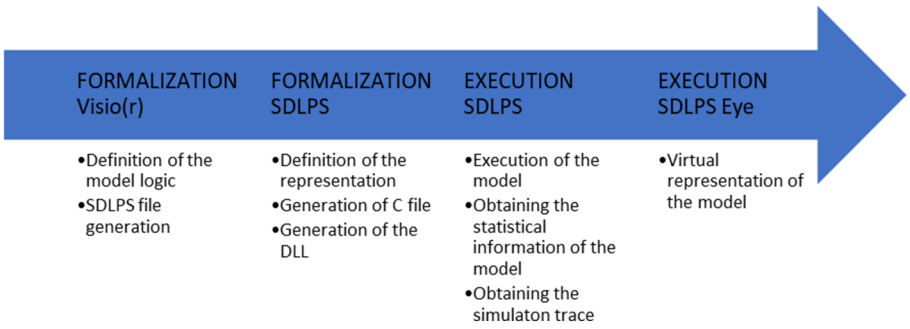
Figura 5. Usando un PROCEDURE dentro de un PROCEDURECALL.



Plane en este ejemplo se está pasando **por referencia**.

Para pasar un puntero el símbolo «&» no está permitido en SDLPS, se usará «@.p». Esto es debido a una limitación existente en el actual analizador, que genera el código que finalmente se usará para ejecutar el co-simulador. En la figura 6 se puede ver el proceso que se sigue para ejecutar un modelo en SDLPS.

Figura 6. Proceso de ejecución de un modelo en el sistema SDLPS.



No se puede llamar a un PROCEDURE dentro de un TASK; la excepción a esta norma es si el procedimiento está definido en código externo.

3.2.1. PROCEDURES integrados en el SDLPS

El sistema SDLPS tiene diferentes PROCEDURES implementados para poder interactuar con elementos del modelo, como por ejemplo la variable TIME.

Nota

Los PROCESS y BLOCKS se identifican añadiendo recursivamente el nombre del agente padre hasta llegar al SYSTEM, por ejemplo: «Block1_SubBlock1_Process1».

1) Procedimientos de animación

Estos procedimientos permiten representar el modelo en un sistema de realidad virtual o aumentada denominado SDLPSEye.

a) AnimTo

AnimTo(char* strID, char* meshFileName, double dDelay, double dPosX, double dPosY)

Este procedimiento se utiliza para representar el modelo. Representa el objeto con ID *strID* con una malla definida por *meshFileName*. La representación se posicionará mediante dPosX y dPosY y se puede definir la demora mediante el parámetro *delay*.

Si no existe la representación del objeto identificado por *strID*, se creará uno nuevo.

b) RemoveAt

RemoveAt(char* strID)

El PROCEDURE elimina el objeto con identificador *strID*.

2) PROCEDURES para la adquisición de información estadística

Estos procedimientos no modifican el funcionamiento del modelo, pero permiten recuperar información estadística del modelo. Esta información será útil para determinar el comportamiento del modelo y compararlo con el sistema físico real (validación); además, permite obtener información de los diferentes escenarios que podemos ejecutar con los sistemas simulados.

a) Report

Report(double Variable)

Escribid en la traza un evento <Eye_Report> especificando el nombre de la variable, el tipo, el valor, al agente SDL al que pertenece y el tiempo de ejecución del evento. Este PROCEDURE es muy útil para depurar el modelo y recuperar información estadística. Recoged información de valores de tipo *double*.

b) ReportArray

ReportArray(char* desc, double Variable)

Escribid en la traza un evento <Eye_Report> especificando el nombre de la variable, el tipo, el valor, al agente SDL al que pertenece y el tiempo de ejecución del evento. El parámetro que esperamos ahora es un vector; está pensado para recuperar información del modelo en forma de series temporales.

c) GetAgentClock

GetAgentClock (double Variable)

Permite obtener la variable TIME del agente que llama a este procedimiento.

d) GetAgentSender

GetSender(char* PId)

Obtiene el PId del AGENT que ha enviado la señal.

3.3. Código externo: SDLCodeExternal

Para simplificar la definición de modelos es posible utilizar rutinas implementadas en código de C en un archivo externo. El código debe ser implementado en un archivo C y debe estar acompañado por un archivo de encabezado con el mismo nombre, por ejemplo:

nombreDeArchivo1.c y *nombreDeArchivo1.h.* Cada función implementada en C debe declararse en el archivo de cabecera correspondiente.

Cada par de archivos debe incluirse en un archivo llamado SDLCodeExternal. h que es el archivo que incluye por defecto el simulador SDLPS. Por ejemplo, podría ser el contenido del archivo SDLCodeExternal. h:

#include “fitxero1.h”

#include “fitxero2.h”

Seguido de todos los *includes* que se necesiten.

4. Ficheros generados

El modelo SDL está definido en diagramas representados cada uno de ellos en un fichero que sigue el formato estándar vsdx de Microsoft Visio. A partir de esta descripción gráfica del modelo, el sistema SDLPS genera una serie de ficheros que pueden ser ejecutados. Este conjunto de ficheros se describe a continuación.

4.1. Model file (.sdlps)

Este fichero contiene la estructura de BLOCKS y PROCESSES y sus comunicaciones, los canales. Este fichero se genera automáticamente a través del sistema SDLPS que interpreta los diagramas vsdx de Visio que representan modelos SDL (ver figura 7).

Figura 7. Fichero xml representando un modelo SDL para SDLPS.

```
<?xml version="1.0"?>
<system id="0" name="ACRE" implementation="" IP="" portRead="">
    <channels>
        <channel name="channel1" start="BADP" end="BCinta" dual="yes">
            <event name="NewReference"></event>
            <event name="TakeReference"></event>
            <event name="Full"></event>
        </channel>
        <channel name="channel2" start="BCinta" end="BRobot" dual="yes">
            <event name="NewReference"></event>
            <event name="TakeReference"></event>
            <event name="Full"></event>
        </channel>
        <channel name="channel3" start="BRobot" end="BElevator" dual="yes">
            <event name="NewReference"></event>
            <event name="TakeReference"></event>
        </channel>
    </channels>
    <block id="1" name="BADP" implementation="" IP="" portRead="">
        <channels>
            <channel name="ADPCh" start="BADP" end="PADP" dual="yes">
                <event name="NewReference"></event>
                <event name="TakeReference"></event>
                <event name="Full"></event>
            </channel>
        </channels>
        <process id="1" name="PADP" implementation="ADP.sdlprocess" IP="" portRead="">
        <DCLS>
            <DCL name="PrevElem" type="char*" value="''"></DCL>
            <DCL name="NextElem" type="char*" value="'BCinta_PCinta'"></DCL>
            <DCL name="ElemID" type="double" value="1000"></DCL>
            <DCL name="OutputVel" type="double" value="0"></DCL>
            <DCL name="minVel" type="double" value="0"></DCL>
            <DCL name="rnd_seed" type="double" value="3793"></DCL>
            <DCL name="rnd_a" type="double" value="16807"></DCL>
        </DCLS>
        </process>
    </block>
    <block id="2" name="BCinta" implementation="" IP="" portRead="">
        <channels>
            <channel name="CintaCh" start="BCinta" end="PCinta" dual="yes">
                <event name="NewReference"></event>
                <event name="TakeReference"></event>
                <event name="Full"></event>
```

4.1.1. Utilidades para depurar la aplicación

Para depurar las aplicaciones, se puede usar el parámetro oculto *debug* en el bloque TASK, editando directamente el fichero de código xml que contiene la descripción del modelo, tal como se muestra en la figura 8.

Figura 8. Edición del campo debug en el bloque TASK.

```
<input id="19" name="OnBusy"/>
<task id="20" name="">
    exit_state = 0;
    <debug message="executa" variable="exit_state"/>
</task>
<task id="21" name="">
    <debug message="executa2" variable="exit_state"/>
</task>
<procedurecall id="22" name="Report">
    <param name="PARAM_1" value="exit_state"/>
</procedurecall>
<setstate id="23" name="FREE"/>
```

Con este campo, el sistema SDLPS mostrará un mensaje en una ventana emergente mostrando el mensaje «executa» y el valor de la variable «exit_state».

4.2. Representation file (.eye)

Este archivo contiene la información relativa a la representación del modelo en un espacio 3D. Cada agente que quiere ser representado habrá definido una representación para cada uno de sus estados (ver figura 9). La información de la representación consiste en:

- Escalado y fichero de la malla (formato *Wavefront Object File* *.obj).
- Posición x, y, z.
- Rotación x, y, z.

Figura 9. Fichero de representación.

```
<ModelInfo>
<ModelName>Ausonia</ModelName>
<Agents>
  <Agent name='BRobot_PRobot'>
    <state name='IDLE'>
      <mesh scale='1'>box_green.obj</mesh>
      <pos x='220' y='0' z='0' />
      <rot x='0' y='0' z='0' />
    </state>
    <state name='BLOCKED'>
      <mesh scale='1'>box_red.obj</mesh>
      <pos x='220' y='0' z='0' />
      <rot x='0' y='0' z='0' />
    </state>
    <state name='WORKING'>
      <mesh scale='1'>box_yellow.obj</mesh>
      <pos x='220' y='0' z='0' />
      <rot x='0' y='0' z='0' />
    </state>
  </Agent>
  <Agent name='BElevator_PElevator'>
    <state name='RUNNING'>
      <mesh scale='1'>box_green.obj</mesh>
      <pos x='250' y='0' z='0' />
      <rot x='0' y='0' z='0' />
    </state>
  </Agent>
  <Agent name='BADP_PADP'>
    <state name='RUNNING'>
      <mesh scale='1'>box_green.obj</mesh>
      <pos x='-20' y='0' z='0' />
      <rot x='0' y='0' z='0' />
    </state>
    <state name='BLOCKED'>
      <mesh scale='1'>box_red.obj</mesh>
      <pos x='-20' y='0' z='0' />
      <rot x='0' y='0' z='0' />
    </state>
  </Agent>
  <Agent name='BCinta_PCinta'>
    <state name='ROLLING'>
      <mesh scale='1'>cinta200x20.obj</mesh>
      <pos x='100' y='-10' z='0' />
      <rot x='0' y='0' z='0' />
    </state>
  </Agent>
</Agents>
</ModelInfo>
```

4.3. SDL PROCESS File (.sdlprocess)

Un proceso puede estar implementado en un fichero externo para permitir la construcción de librerías y reutilizar objetos (ver figura 10).

Figura 10. Fichero de proceso.

```
<?xml version="1.0"?>
<sdlprocess>
    <process id="1" name="PElevator" implementation="" IP="" portRead="">
        <DCLS>
            <DCL name="PrevElem" type="char*" value="'PID'"></DCL>
            <DCL name="ElemID" type="double" value=""></DCL>
        </DCLS>
        <start>
            <setstate id="1" name="RUNNING"></setstate>
        </start>
        <state name = "RUNNING">
            <input id="1" name="NewReference"></input>
            <procedurecall id="2" name="RemoveAt">
                <param name="PARAM_1" value="ElemID"></param>
            </procedurecall>
            <output id="2" name="TakeReference" self="" to="PrevElem" via="">
            </output>
            <setstate id="3" name="RUNNING"></setstate>
        </state>
    </process>
</sdlprocess>
```

4.4. PARAMETRIZATION File (.param)

Este archivo contiene la parametrización de un modelo en particular. Una parametrización específica puede ser la configuración de una ejecución de un conjunto de experimentos. Definir los valores específicos de variables que caracterizan el modelo y variables DCL a ejecutarse en la simulación.

Ejemplo

```
<Parametrizations>
  <DCLS>
    <DCL agent="BRobot_PRobot" name="PackagingTime" type="double" value="3" />
  </DCLS>
  <ExternalEvents>
    <ExternalEvent agent="BRobot_PRobot" type="BreakDown" xTime="4" />
    <ExternalEvent agent="BRobot_PRobot" type="Fix" xTime="15" />
  </ExternalEvants>
</Parametrizations>
```

4.5. TRACE FILE (.trace)

Este fichero contiene la salida de la ejecución del modelo (ver figura 11). Se compone de dos partes:

- Cabecera: definida a través del tag *ModelInfo*, contiene la información del modelo, así como el nombre, el fichero de representación, el tipo de ejecución, la parametrización, etc.

- Eventos: tenemos de dos tipos:
  - Eventos de ejecución: input, output, setstate…

– Eventos de representación: Eye_AnimTo, Eye_Remove, Eye_SetState, Eye_Report.

Figura 11. Fichero de traza.


```xml
<?xml version='1.0' encoding='utf-8'?>
<Trace>
  <parametrization>D:\SOFT\SDLPS\SDLPS\SH_SDL\Models\00DemoArxiuPartit\ACRE\params\prova5.param</parametrization>
  <ModelInfo>...</ModelInfo>
  <Events>
    <setstate process="BElevator_PElevator" id="1" simTime="0.000000" state="RUNNING"></setstate>
    <EYE_SetState xTime='0.000000' agent="BElevator_PElevator" state="RUNNING"></EYE_SetState>
    <setstate process="BRobot_PRobot" id="1" simTime="0.000000" state="IDLE"></setstate>
    <EYE_SetState xTime='0.000000' agent="BRobot_PRobot" state="IDLE"></EYE_SetState>
    <output process="BADP_PADP" name="NewReference" id="1" self="" to="NextElem" via="">
      <params to="" creationtime="0.000000" delay="0.000000" executiontime="0.000000" priority="0" type="NewRefere
      <userparams>
        <userparam name="ElemID" type="" value="1000.000000"></userparam>
      </userparams>
    </output>
    <setstate process="BADP_PADP" id="2" simTime="0.000000" state="RUNNING"></setstate>
    <EYE_SetState xTime='0.000000' agent="BADP_PADP" state="RUNNING"></EYE_SetState>
    <setstate process="BCinta_PCinta" id="2" simTime="0.000000" state="ROLLING"></setstate>
    <EYE_SetState xTime='0.000000' agent="BCinta_PCinta" state="ROLLING"></EYE_SetState>
    <input process="BCinta_PCinta" name="NewReference" id="1">
      <params to="BCinta_PCinta" creationtime="0.000000" delay="0.000000" executiontime="0.000000" priority="0" ty
      <userparams>
        <userparam name="ElemID" type="" value="1000.000000"></userparam>
      </userparams>
    </input>
    <output process="BCinta_PCinta" name="TakeReference" id="3" self="" to="PrevElem" via="">
      <params to="" creationtime="0.000000" delay="0.777778" executiontime="0.000000" priority="0" type="TakeRefer
    </output>
    <EYE_AnimTo xTime='0.000000' agent='1000.000000'>
      <mesh scale='1'>paquet.obj</mesh>
      <pos x='10.000000' y='0.000000' z='0' />
    </EYE_AnimTo>
    <output process="BCinta_PCinta" name="Roll" id="7" self="yes" to="" via="">
      <params to="" creationtime="0.000000" delay="7.000002" executiontime="0.000000" priority="0" type="Roll" cur
      <userparams>
        <userparam name="tmpID" type="" value="1000.000000"></userparam>
        <userparam name="tmpPos" type="" value="1.000000"></userparam>
      </userparams>
    </output>
    <setstate process="BCinta_PCinta" id="8" simTime="0.000000" state="ROLLING"></setstate>
    <input process="BADP_PADP" name="TakeReference" id="6">
      <params to="BADP_PADP" creationtime="0.000000" delay="0.777778" executiontime="0.777778" priority="0" type="
    </input>
    <EYE_report xTime='0.777778' agent="BADP_PADP" varname="rnd_seed" vartype="double" varvalue="226.000000" />
    <output process="BADP_PADP" name="NewReference" id="9" self="" to="NextElem" via="">
      <params to="" creationtime="0.000000" delay="0.000000" executiontime="0.000000" priority="0" type="NewRefere
      <userparams>
        <userparam name="ElemID" type="" value="1001.000000"></userparam>
      </userparams>
    </output>
```

5. El ejercicio

En el ejercicio se trabajará con SDL con el objetivo de entender cómo modelar y combinar un modelo de simulación con otros sistemas, ya sean físicos o no, usando el mecanismo de la co-simulación.

Se implementará un modelo de agente inteligente reflexivo simple usando alguna herramienta que entienda SDL. Una vez implementado el agente, se implementará una simplificación del modelo de «steam-boiler».

Se buscará que en los dos modelos se defina cómo establecer una comunicación entre los dos a través del mecanismo de co-simulación.

Bibliografía

Boudjelal, F. A.; Lafaye, J.-Y.; Louis, G. (1997). «Specifying the steam-boiler case study with SDL». En: *SDL '97: Time for testing* (págs. 491-505). Ámsterdam: Elsevier.

Fonseca i Casas, P. (2010). «Using specification and description language to define and implement discrete simulation models». En: *Actas de la 2010 Summer Computer Simulation Conference, SCSC '10* (núm. 1, págs. 419-426).

Fonseca i Casas, P. (2014, mayo). «Using specification and description language to formalize multiagent systems». *Applied Artificial Intelligence* (vol. 28, núm. 5, págs. 504-531).

Fonseca Casas, P.; Pi Palomés, X.; Casanovas Garcia, J.; Jové, J. (2013). «Definition of virtual reality simulation models using specification and description language diagrams». En: *SDL 2013: Model driven dependability engineering* (págs. 258-274). Berlín / Heidelberg: Springer-Verlag.

Fujimoto, R. M. (2001). *Parallel and distributed simulation systems*. Nueva York: John Wiley & Sons.

Malik, A. W.; Park, A. J.; Fujimoto, R. M. (2010). «An optimistic parallel simulation protocol for cloud computing environments». *SCS Modeling & Simulation Magazine* (vol. 1, núm. 4, págs. 1-9.

PragmaDev Studio [en línea]. [Fecha de consulta: 9 de enero de 2016]. PragmaDev SARL, 2016. <http://www.pragmadev.com/product/index.html>

Vangheluwe, H. L. M. (2000). «DEVS as a common denominator for multi-formalism hybrid systems modelling». En: *Actas del CACSD 2000. IEEE International Symposium on Computer-Aided Control System Design* (Cat. núm. 00TH8537).

Zeigler, B. P.; Praehofer, H.; Kim, T. G. (2000). *Theory of modelling and simulation: Integrating discrete event and continuous complex dynamic systems*. San Diego: Academic Press.