Project: COMPASS

Grant Agreement: 287829

Comprehensive Modelling for Advanced Systems of Systems

COMPASS

Linkage to Executable Software

Document Number: D32.3

Date: July 2014

Public Document

http://www.compass-research.eu

Contributors:

Adrian Larkham, ATEGO

Joey Coleman, AU

Kenneth Guldbrandt Lausdahl, AU

Klaus Kristensen, B&O

Editors:

Adrian Larkham, ATEGO

Reviewers:

Ana Cavalcanti, York

Lucas Lima, UFPE

Ken Pierce, Newcastle

Document History

Release Versions

Version	Date	Author	Description
1.0	05.08.2014	A. Larkham	Final release version with comments addressed.

Pre-Release Versions

Version	Date	Author	Description
0.1	27.03.2014	A. Larkham	Initial draft outline
0.2	04.06.2014	A. Larkham	Incorporate Arhus contribution.
0.3	26.06.2014	A. Larkham	Correction to existing text.
0.4	02.07.2014	A. Larkham	Further corrections to existing text.
0.5	07.07.2014	A. Larkham	Add introduction and conclusion.
0.6	08.07.2014	A. Larkham	Add additional paragraph to conclusion.
0.7	08.07.2014	J. Coleman	Edit pass
0.8	14.07.2014	J. Coleman	Edit pass from internal review
0.9	16.07.2014	J. Coleman	Another edit pass
0.91	18.07.2014	A. Larkham	Address internal review comments

Table of Contents

Figures

Tables

1. Introduction

The co-simulation engine described in COMPASS Deliverable D32.4 [LMNC14] provides ability to simulate models where a portion of the modelled system is actually being simulated by other tools or systems that are external to the simulator. This allows an SoS model to be simulated with an external Constituent System (CS) by enabling the interaction between the CML interpreter in Symphony and the external CS.

The need for a co-simulation capability is due to the autonomy and independence of the CS providers that participate in SoS design. In some cases, CSs of the SoS may be delivered by a supplier as a COTS product. In other cases some of the CSs of the SoS may be legacy systems. In both cases the internal design of the CSs may not be available creating a situation where the CSs are only described in terms of their interfaces. This makes it difficult to create CML models of all of the CSs of the SoS. As a result, simulation of the SoS is a challenging task, as a meaningful simulation depends on the actual behaviour of the CSs involved. In order to include the behaviour of such external CSs, the co-simulation engine can be used to simulate SoS models for which the behaviour of some of the CSs is obtained via the external system.

This document describes the co-simulation framework, which provides support for linking external CSs to a CML model to support the simulation of a SoS. The co-simulation framework provides the necessary infrastructure to link external CSs with the co-simulation engine.

The deliverable is structured as follows Section 2 provides a technical overview of the co-simulation framework; Section 3 provides a User's Guide to how to use co-simulation framework using C++; and Section 4 provides conclusions.

2. Technical Overview

The co-simulation framework presented here is designed to support the linking of external CSs with the co-simulation engine. The framework is designed from the specification given in [LMNC14]. It uses the TCP [Pos81] communication protocol and the JSON data format [Cro06], and is based on the *Inversion Of Control* (IOC) design. The IOC design is a style of software construction where reusable code controls the execution of problem-specific code. It makes the assumption that the reusable code and the problem-specific code are developed independently, which often results in a single integrated application. It can be described by the following properties:

- There is a decoupling of the execution of a certain task from its implementation.
- Modules are self-contained, and make no assumptions about the behaviour of other modules apart from relying on their contracts.
- Replacing a module does not affect other modules.

The co-simulation framework contains two layers: *Transport* and *CompassCosim*:

Transport: The transport layer is responsible for the communication protocol at the network level and the encoding/decoding of JSON. It does not implement the execution semantics of [LMNC14], but provides an interface for the *CompassCosim* layer with services that include the necessary controls to implement this semantics.

CompassCosim: The *CompassCosim* layer exposes a framework that implements the execution semantics from [LMNC14]. The framework provides a call-back interface that must be implemented for external CSs as shown in Figure 1, and a wrapper class for channel synchronization to make it easy to express synchronization in the target language.

```
class ACoSimulationCallback{
public:
   ChannelEventObjectSet inspect();
   void execute(ChannelEventObjectSmartPtr evt);
   bool finished() const;
   void init();
   void deInit();
}
```

Figure 1 Abstract class ACoSimulationCallback that external CSs must implement

The call-back interface `ACoSimulationCallback` defines three methods that are used to implement the semantics:

inspect: The `inspect` method is used to return all possible synchronization points currently offered by the external CS. The set of returned synchronization points must either be a subset of the possible

synchronization points altered by the execute method, or simply all synchronization points that are possible on the available channels.

execute: The `execute` method implements the behaviour of external CSs. It must examine the synchronization event, modify the external CS, and reflect external CS's state change by altering the set of events returned from the `inspect` method.

finished: This method is used to signal that the external CS has completed all execution and is shutting down.

The call-back also provides two framework methods to allow the external CS to be initialized (`init`) and de-initialized (`deInit`).

An overview of the layers are given in Figure 2 and Figure 3 The class diagram shown in Figure 2 outlines the transport layer including the usage of a socket layer that provides TCP communication. The layer provides the essential parts for communication with the CML interpreter. The *CompassCosim* layer outlined in Figure 3 adds the necessary classes to express the synchronization points used in CML. Figure 4 illustrates how an external CS interfaces with the framework. This is described in detail in Section 2.3; however, the in-depth detail of the internal framework design and how it implements the CSP-derived CML operator semantics is beyond the scope of this document and has been omitted. More details regarding this can be found in [LMNC14].

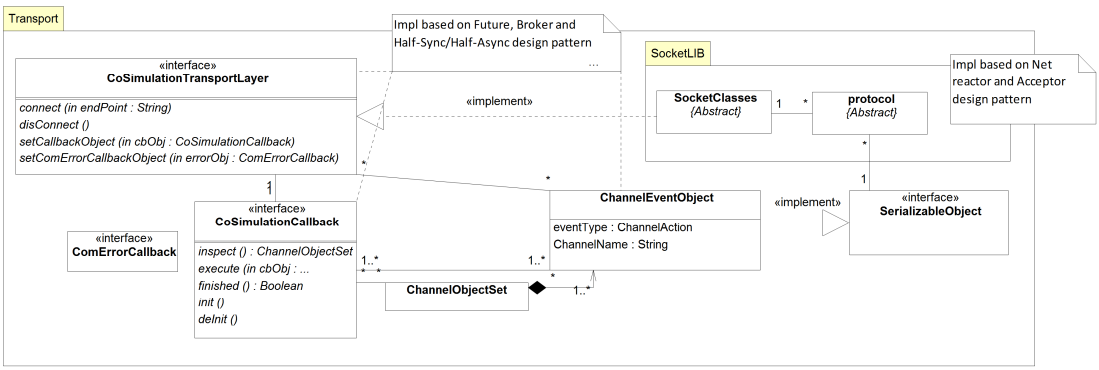

Figure 2 Transport layer of the co-simulation framework

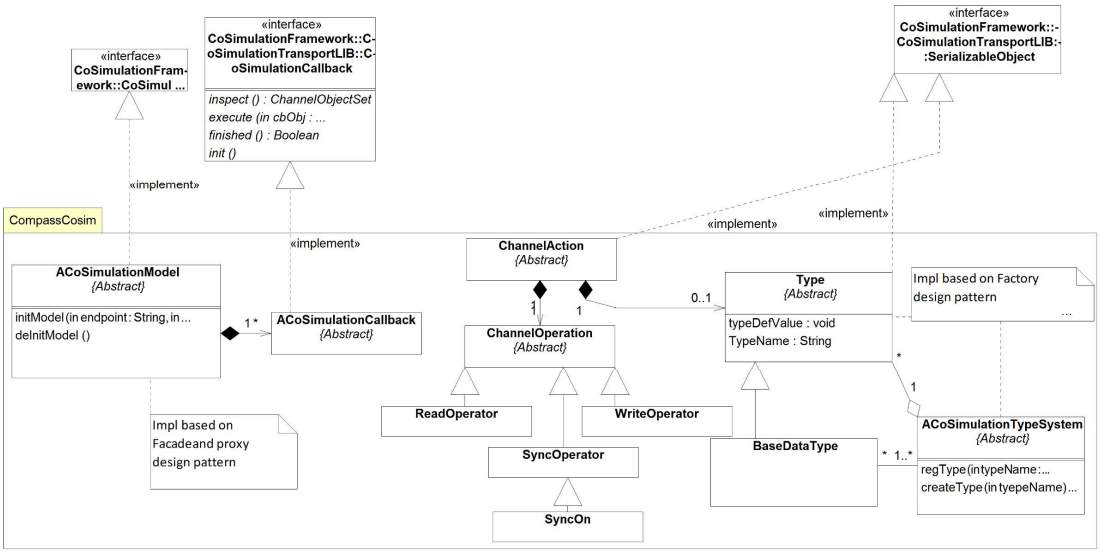

Figure 3 CompassCoSim layer of the co-simulation framework

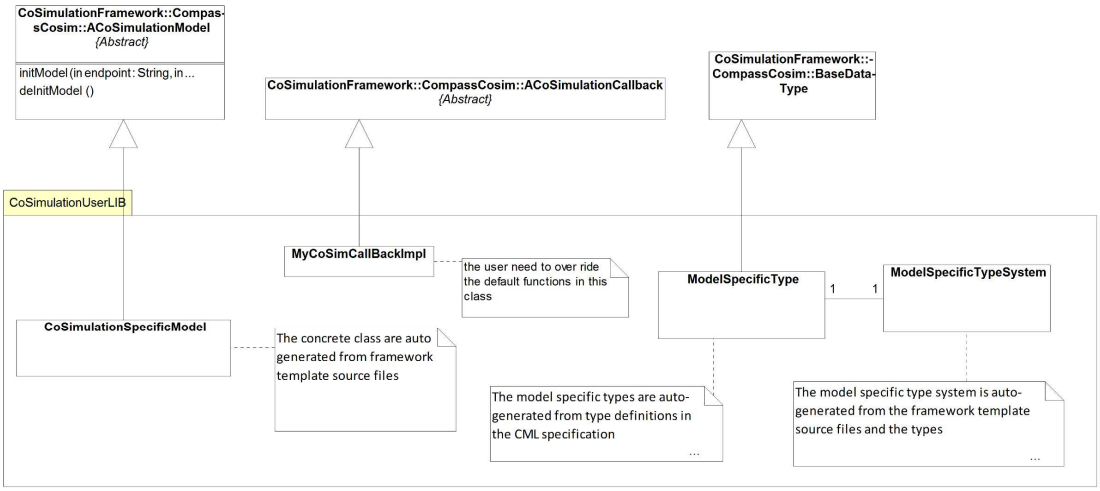

Figure 4 External CS interface with CompassCoSim layer of the co-simulation framework

2.1. Type Support

The framework uses C++ Runtime Type Identification (RTI) to identify channel types, and therefore it is able to automatically detect the type of channels with a single CML basic type. A mapping between CML and C++ types are shown in Table 1[1].

CML Type Name	C++ Type Name
quote	`string`
bool	`bool`
char	`char`
int	`int`
nat	`int`
nat1	`int`
rat	`double`
real	`double`
seq of char	`string`
others[2]	-

Table 1 Relation between CML basic types and C++ basic types used to represent channel values

[1] Note that the implementation does not support records or other model specific types, but this is not a restriction of the framework itself nor the C++ language, but a limitation in the provided prototype.

[2] Other types are not handled by the framework but can be manually added by the user of the framework. Manual addition of other types can be used to overcome the limitation of this initial version of the framework to simple types.

2.2. Synchronization / Channel Communication

The *CompassCosim* framework represents channel communication events using the class `ChannelEventObjectSmartPtr`. This class defines a number of methods to obtain the channel name, communication type (Read/Write) as well as the type and value of the communication. Table 2 illustrates how instances of this class can be obtained using helper functions, where the function calls represent the given CML channel communications. The table uses `c` as the channel and `A` as a place-holder for whatever action follows after the communication.

Channel	CML	C++
	Sync	
c	c->A	createSyncEventObject("c")
	Write	
c : int	c.1->A	createWriteSyncOnEventObject<int>("c",SYNC ON, 1)
c : int	c!1->A	createWriteSyncOnEventObject<int>("c",WRITE,1)
	Read	
c : int	c?x->A	createReadEventObject<int>("c")

Table 2 Mapping between CML channel communication and the creation of a framework representation

The methods presented in Table 2 are part of the call-back class that must be implemented by users of the framework. The created events must be returned by the `inspect` method when communicating with the simulator. The `inspect` method must return a set of events that the external CS offers to synchronize on, and likewise the `execute` method must receive the event that has been chosen for synchronization by the simulator.

An illustration of a basic `inspect` method is shown in Figure 5. It offers synchronization on the untyped channel `i`.

```
ChannelEventObjectSet inspect()
{
   ChannelEventObjectSet eventOptions;
   eventOptions.push_back(createSyncEventObject("i"));
   return eventOptions;
}
```

Figure 5 Basic implementation of i->A

The `execute` method receives a synchronization event as parameter, and this event is examined to determine which synchronisation is requested by the co-simulation coordinator, and how this should affect the external CS itself. In

Figure 6 an example is given of an `execute` method that sets an exit variable to `true` when a synchronization on `i` is executed or, when a synchronization on a channel `c` is executed then it compares the channel value associated with the string `"SOURCE NODE"` and sets the field `activeNode` to the value of 1 if they match.

```
void execute(ChannelEventObjectSmartPtr  evt)
{
  if (evt->getChannelName() == "i")
    exitTrue = true;
  else if(evt->getChannelName() == "c")
  {
    ChannelEventObject<std::string>* robj = static_cast<
    ChannelEventObject<std::string>*> (evt.get());
    if(robj->action.type() == "SOURCE_NODE")
      activeNode = 1;
    ...
  }
}
```

Figure 6 Basic example of the execute method handing i->A and c?x

2.3. Configuring Constituent System Entry Point

The framework is delivered with a standard entry point as shown in Figure 7. It is configured to run an empty implementation of the call-back class. The entry point shown is a Windows `_tmain` function. The configuration of the entry point is separated into a define file `CoSimConfig.hpp` that defines the connection point and the process name from the CML specification that the CS represents. The second part is a `typedef` that configures the call-back class that the framework should instantiate when invoked.

```
int _tmain(int argc, _TCHAR* argv[])
{
  using namespace ExternalSystem;
  try{
    typedef CoSimulationFramework::ACoSimulationModel<
      ExternalSystemBasicImpl> ExternalSystem;
    ExternalSystem model(EXTERNAL_PROCESS);
    model.initModel(SYMPHONY_HOST, SYMPHONY_PORT);
  }
  catch (...)
  {
    std::cerr << "Unknown error \n";
  }
...
}
```

Figure 7 Windows Entry point for the framework

The configuration of the connection point and process is shown in Figure 8, and defines the host URL, port, and the process name of this external CS as defined in the CML specification. The name links the call-back class implementation to the expected behaviour from the CML specification.

```
#define SYMPHONY_HOST "localhost"
#define SYMPHONY_PORT 8882
#define EXTERNAL_PROCESS "ExternalProcess"
```

Figure 8 CoSimConfig.hpp file that defines the connection point and process

The type definition in Figure 8 can be changed to define a different call-back implementation. Such a call-back will be handwritten to wrap an existing CS

implementation or as a new implementation custom built for the purpose. In Figure 9 a custom definition is shown that may replace the call-back in Figure 7 with a concrete implementation named `CustomExternalSystemImpl`.

```
typedef CoSimulationFramework::ACoSimulationModel<
  CustomExternalSystemImpl > ExternalSystem;
```

Figure 9 Custom typedef overriding the basic implementation

2.4. Prototype Limitations

The prototype implementation of the framework presented here does not cover the complete CML language, and as a result the following restrictions apply:

channel types: Synchronisation is only supported over simply-typed or untyped channels, specifically, those declared as `c : N` or just as `c`. The implementation cannot deal with channel synchronization over channels of the form: `c : N * N`.

complex types: Automatic type mapping is only supported for the simple CML types as shown in Table 1. As a result model-specific record types and other composite types are not supported.

The limitations listed above are entirely artefacts of the prototype implementation, and not limitations of the general approach taken here.

3. User Guide

Section 3 describes how to use co-simulation framework to link an external CS with the co-simulation engine using C++.

Symphony is shipped with built-in support for externalizing CML processes (representing external CSs) to C++ using Microsoft Visual Studio 2010 or 2013. The tool is able to generate the necessary projects with the required build configuration to compile and link the applications. The generated projects can be executed with a basic implementation of the call-back described in Section 2.3.

3.1. System Requirements

To be able to use and link external CS using the framework shipped with Symphony the following applications and frameworks must be present on the target machine:

- Microsoft Visual Studio 2010 or 2013, able to build 32 bit applications.
- Boost 1.55 compiled with either MSVC 1600 or 1800 (Visual Studio 2010 or 2013 C++ compiler) in 32 bit.
- *CMake* version 2.8.12.2 or greater.

Note that the framework can be compiled to any platform supporting Boost 1.55 but the shipped interface library to the Symphony interpreter are only pre-compiled for this listed Windows environment.

3.2. Preparing Library, Sources and Generating the C++ Project

It is assumed that a CML project has already been imported into Symphony, and that the project has been tested in co-simulation mode between two Symphony instances as described in [LMNC14]. The first step when externalizing a CML process is to obtain the framework, template source files and a configured Visual Studio project.

Preparing the libraries and project is performed following these steps:

1. Copying the template configuration.
2. Running *CMake* to create a Visual Studio project configured for the framework.
3. Generating or creating the `CoSimConfig.hpp` file.

3.3. Copying the Template Configuration into Place

Symphony provides a context menu on a project selection that is capable of copying the template and pre-compiled interface library into the project itself. The context menu is shown in Figure 10 as *Create External System Template*.

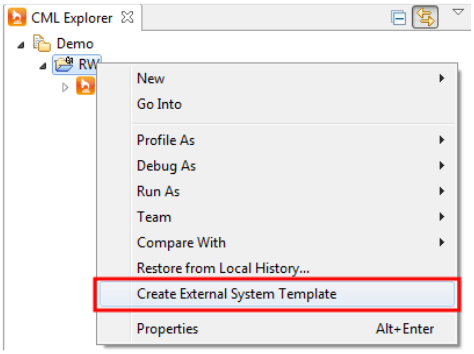

Figure 10 Symphony template creation

After the template is copied, the project folder structure will be as shown in Figure 11 with the external system folder containing the *CMakeLists.txt* file that is needed to generate the Visual Studio project.

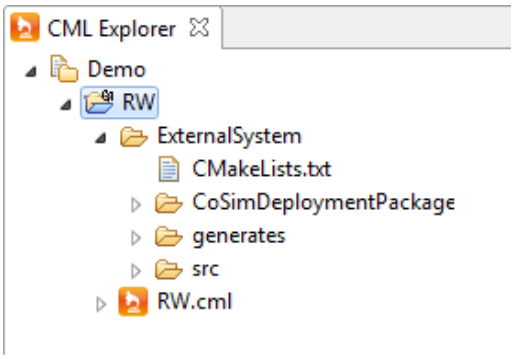

Figure 11 Folder view with template in place

3.3.1. Creating the Visual Studio Project

The Visual Studio project can be generated after the template folder *ExternalSystem* has been added to the CML project. The Visual Studio project is generated using *CMake* as shown in Figure 12. *CMake* must initially be configured with two options:

Where is the source code: Containing the path to the generated template folder *ExternalSystem*.

Where to build the binaries: A path to where the Visual Studio project should be created e.g. *ExternalSystem/VisualStudio.*

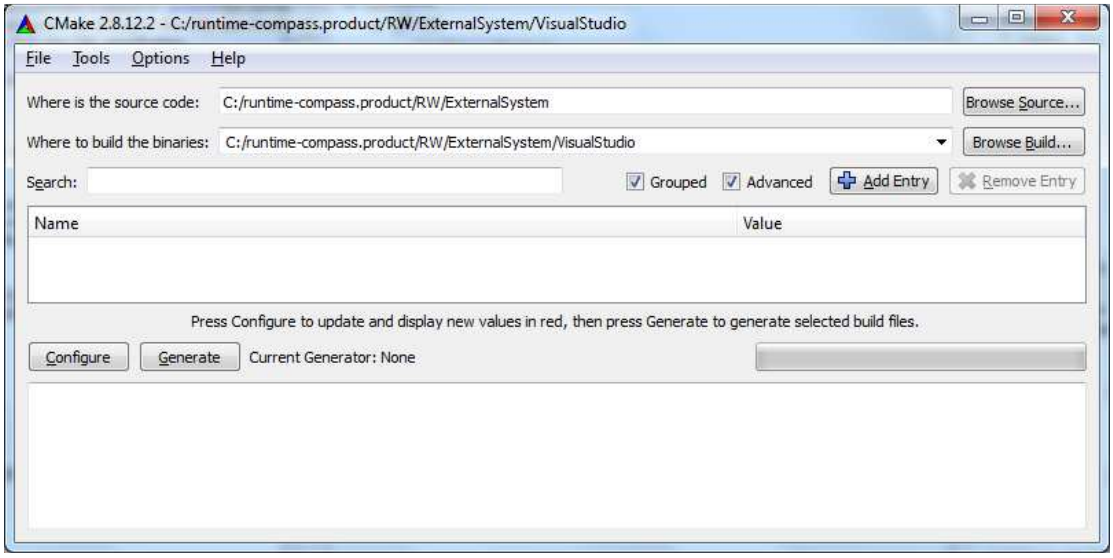

Figure 12 *CMake* main window

When these settings are entered, the *Configure* button must be pressed, and the version of Visual Studio selected. It is **important** to select the same version that has been used to compile Boost. The selection window is shown in Figure 13 with the default settings.

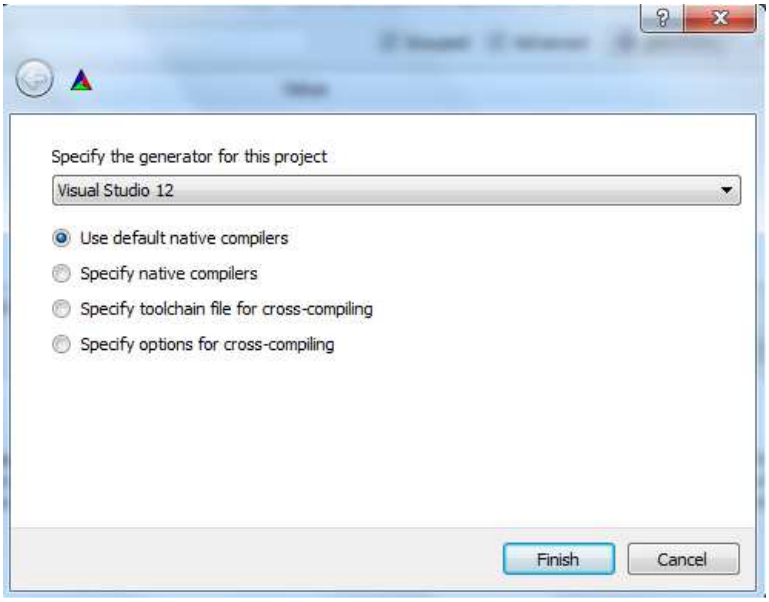

Figure 13 Project generator selection

After the *Finish* button is pressed, *CMake* produces the output log shown in Figure 14, complaining that it is unable to find Boost. The error is expected.

```
The C compiler identification is MSVC 18.0.21005.1
The CXX compiler identification is MSVC 18.0.21005.1
Check for working C compiler using: Visual Studio 12
Check for working C compiler using: Visual Studio 12 -- works
Detecting C compiler ABI info
Detecting C compiler ABI info - done
Check for working CXX compiler using: Visual Studio 12
Check for working CXX compiler using: Visual Studio 12 -- works
Detecting CXX compiler ABI info
Detecting CXX compiler ABI info - done
Could NOT find Boost
CMake Error at CMakeLists.txt:50 (message):
   Boost not found (or too old) please set ENV: Boost_INCLUDE_DIR to
      boost home

Configuring incomplete, errors occurred!
```

Figure 14 Initial log output from *CMake*

When this appears, the middle section in Figure 12 changes and looks as shown in Figure 15, enabling the user to enter the path for *Boost_INCLUDE_DIR*.

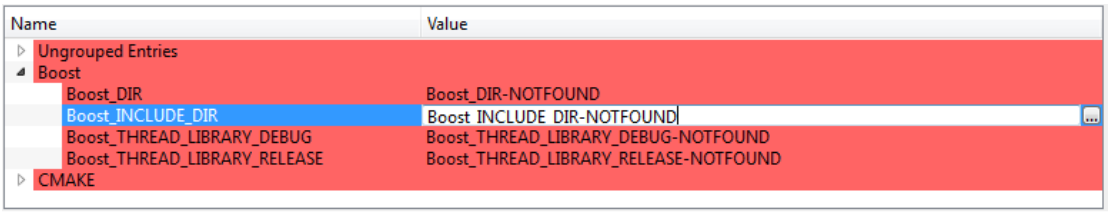

Figure 15 *CMake* property configuration

Enter the path to the include directory for Boost in *Boost_INCLUDE_DIR*, then click the *Configure* button (the small button with an ellipsis at the right edge of the value field) in *Cmake.* It should now correctly detect the Boost library dependencies. The *Generate* button can then be pressed and *CMake* will output a Visual Studio project in the specified path containing a solution and project named *ExternalSystem* configured for co-simulation.

3.3.2. Generating or Creating the CoSimConfig.hpp file

The `CoSimConfig.hpp` file is now located under the *ExternalSystem* folder. This file can either be manually edited or automatically updated from a co-simulation launch. Figure 16 shows the co-simulation launch dialogue. Pressing the *Configure External System* button updates the `CoSimConfig.hpp` file to match the launch configuration itself.

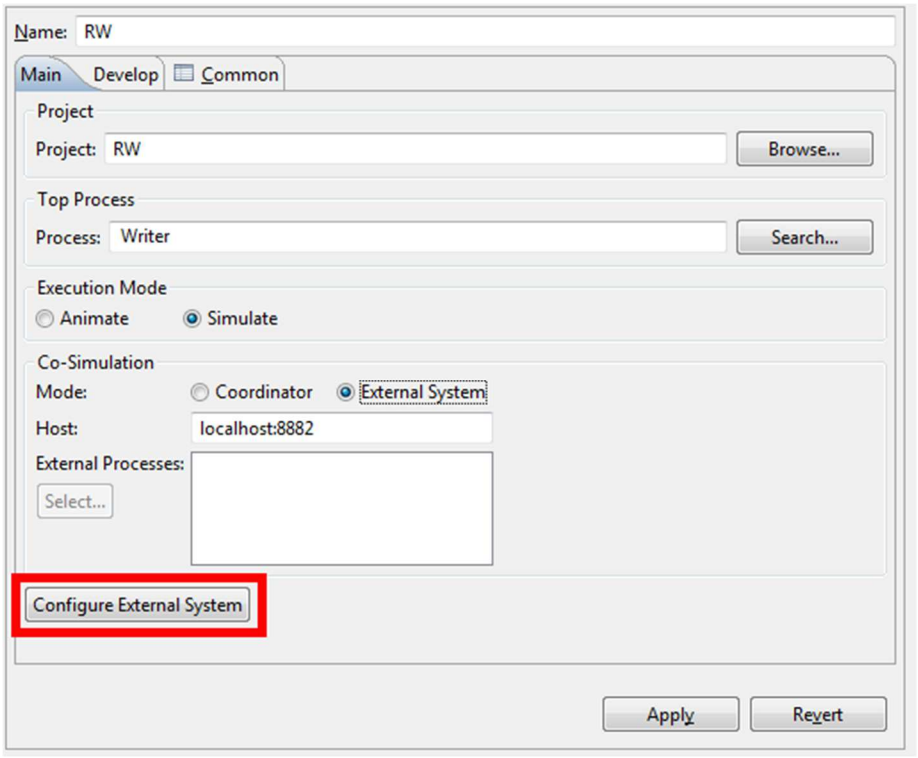

Figure 16 Co-simulation launch configuration

3.4. Simulating and Launching

A new call-back class must be created for every implementation of a CML process that needs to be external to the co-simulation. This is done as described in Section 2.3. In this section the `Writer` process in Figure 17 is externalized; it is part of the reader-writer example supplied with Symphony.

```
process Writer  =
begin
actions
s= val x : int @ a!x -> b?y ->
   let n  =  y+1
   in
      (
         [n <= MAX] & s(n)
         []
         [n > MAX] & Skip
      )
@ s(1)
end
```

Figure 17 The CML Writer process

In our example, the new call-back implementation is named `WriterProcess` as shown in Figure 18 where the class skeleton is given.

```
class WriterProcess:  public  CoSimulationFramework::
   ACoSimulationCallback<>
{
   public:
      ...
   private:
      MyNativeWriterObject myNativeObject;
      typedef enum {READ,WRITE}WRITER_STATE;
      WRITER_STATE state;
};
```

Figure 18 Windows Entry point for the framework

The type definition in the main function must be updated to configure the framework to load this new call-back class instead of the default basic implementation. This is done by importing the header file for the new call-back implementation into the main cpp file and changing the `typedef` as shown in Figure 19.

```
typedef CoSimulationFramework::ACoSimulationModel< WriterProcess>
   ExternalSystem;
```

Figure 19 New typedef for the `WriterProcess` call-back class

It is important that the call-back is placed in a header file (hpp) because of C++ template processing, and that the space (`< Writer..`) before `WriterProcess` is preserved. The latter is due to C++ compatibility with C++ compilers other than Microsoft Visual C++.

To complete the implementation of the writer process, an implementation of `inspect`, `execute` and `finished` must be supplied. One possible implementation is shown in Figure 20, Figure 21, and Figure 22.

```
CoSimulationTransportLayer::IChannelEventObject::ChannelEventObjectSet
      inspect(){
   CoSimulationTransportLayer::IChannelEventObject::
      ChannelEventObjectSet  eventOptions;
   switch(state)
   {
      case WRITE: //val x : int @ a!x == a.x:{1,...,10} event option
         eventOptions.push_back(createWriteSyncOnEventObject<int>("a",
            CoSimulationFramework::ChannelOperation::SYNC_ON,
            myNativeObject.myCount));
         break;
      case  READ: // CML spec b?y event option
         eventOptions.push_back(createReadEventObject<int>("b"));
         break;
      default:
         break;
   }
return eventOptions;
}
```

Figure 20 Writer process implementation of the `inspect` method

```
void execute(CoSimulationTransportLayer::IChannelEventObject::
   ChannelEventObjectSmartPtr  evt){
   switch(state)
   {
      case WRITE:
         {  //val x : int @ a!x = the process has sync on a.x:{1,...,10}
            // just change the local state and do some print
            CoSimulationFramework::ChannelEventObject<int>* robj =
               static_cast<CoSimulationFramework::ChannelEventObject<int
               >*> (evt.get());
            state = READ; // -> b?y
            break;
         }
      case READ:
         {  // update the local objects value and change state
            CoSimulationFramework::ChannelEventObject<int>* robj =
               static_cast<CoSimulationFramework::ChannelEventObject<int
               >*>  (evt.get());
            // from the CML  spec b?y -> let n = y+1
            myNativeObject.myCount =  robj->action.type.value+1;
            state = WRITE; // CML spec s(n) or Skip
            break;
         }
      default:
         break;
   }
}
```

Figure 21 Writer process implementation of the `execute` method

```
bool finished() const{
   return (myNativeObject.myCount > MyNativeWriterObject::MAX);
}
```

Figure 22 Writer process implementation of the `finished` method

4. Conclusions

In this document a technical overview of the co-simulation framework is provided and its use described to link an external CS using C++. The co-simulation framework provides support for linking external processes (representing external CSs) to a CML model of a SoS to support co-simulation.

The prototype implementation of the co-simulation framework presented here does not cover the complete CML language. However, this is simply a limitation of the prototype implementation, and not the general approach taken here.

Generation of the CML process specific framework code from the SysML model has not been implemented in the prototype. The existing SysML to CML generator could be extended to generate the CML process specific framework code in addition to CML from the SysML model.

The co-simulation framework has been trialled using a media streaming device from B&O. The streaming device was linked as an external process using the co-simulation framework to a CML model of the SoS in which the streaming device would be used, and the SoS simulated using the CML simulator.

5. References

[Cro06] D. Crockford. The application/json media type for javascript object notation (json). RFC 4627, Internet Engineering Task Force, July 2006.

[LMNC14] Kenneth Lausdahl, Anders Kaels Malmos, Claus Ballegaard Nielsen, and Joey W. Coleman. Co-simulation engine. Technical report, COMPASS Deliverable, D32.4, June 2014.

[Pos81] J. Postel. Transmission control protocol. RFC 793, Internet Engineering Task Force, September 1981.