Cadmium

A tool for DEVS Modeling and Simulation

User’s Guide

**DRAFT – 21/10/2020**

Cristina Ruiz Martin

Gabriel A. Wainer

Department of Systems and Computer Engineering

Carleton University

1125 Colonel By Dr. Ottawa, ON. Canada

<http://cell-devs.sce.carleton.ca>

<http://www.sce.carleton.ca/faculty/wainer>  
[gwainer@sce.carleton.ca](mailto:gwainer@sce.carleton.ca)

Table of Contents

[Cadmium 4](#_Toc53043233)

[Windows - Installation and example 4](#_Toc53043234)

[Installing Cygwin, GCC and Boost 4](#_Toc53043235)

[Downloading and installing the Cadmium Simulator 12](#_Toc53043236)

[Compiling and Running a Cadmium DEVS Model 13](#_Toc53043237)

[Ubuntu - Installation and example 15](#_Toc53043238)

[System requirements 15](#_Toc53043239)

[Installing Boost 15](#_Toc53043240)

[Installing g++ 16](#_Toc53043241)

[Installing Git 18](#_Toc53043242)

[Installing the ‘make’ command 19](#_Toc53043243)

[Downloading and installing the Cadmium Simulator 20](#_Toc53043244)

[Compiling and Running a Cadmium DEVS Model 21](#_Toc53043245)

[MacOS - Installation and example 23](#_Toc53043246)

[System requirements 23](#_Toc53043247)

[Installing Command Line Tools 23](#_Toc53043248)

[Installing Homebrew and Boost 23](#_Toc53043249)

[Downloading and installing the Cadmium Simulator 23](#_Toc53043250)

[Compiling and Running a Cadmium DEVS Model 25](#_Toc53043251)

[DEVS Model definition: An Example 27](#_Toc53043252)

[Subnet: an atomic model example implemented in Cadmium 27](#_Toc53043253)

[Unit testing the Subnet atomic model 34](#_Toc53043254)

[A Summary on Port Definition 45](#_Toc53043255)

[Defining the make file to compile the test 47](#_Toc53043256)

[Simulating the complete ABP model 49](#_Toc53043257)

[Defining the make file to compile all the test and the ABP 50](#_Toc53043258)

[Cadmium’s Services for Atomic Models 52](#_Toc53043259)

[Declaring ports 53](#_Toc53043260)

[Implementing atomic models: a C++ class 53](#_Toc53043261)

[Using Atomic Models: Creating Instances from the Class 56](#_Toc53043262)

[Cadmium’s Services for Coupled Models 57](#_Toc53043263)

[Declaring ports 57](#_Toc53043264)

[Defining coupled models 58](#_Toc53043265)

[Cadmium’s Services to create Logs 61](#_Toc53043266)

[Cadmium’s Services to Run the Simulation 63](#_Toc53043267)

[Services to Export to Coupled Model to JSON 63](#_Toc53043268)

[Appendix A 65](#_Toc53043269)

[Appendix B 67](#_Toc53043270)

[Appendix C 69](#_Toc53043271)

[Appendix D 72](#_Toc53043272)

[Appendix E 76](#_Toc53043273)

[Appendix F 78](#_Toc53043274)

Cadmium

Cadmium is a tool for Discrete-Event modeling and simulation, based on the DEVS formalism. DEVS is a discrete event paradigm that allows a hierarchical and modular description of the models. Each DEVS model can be behavioral (atomic) or structural (coupled), consisting of inputs, outputs, state variables, and functions to compute the next states and outputs.

Cadmium is a cross-platform header-only library implemented in C++. This document is a user's guide to Cadmium, and we will only focus on tool-related aspects. Readers interested in the underlying theory should consult:

* *G. Wainer. Discrete-Event Modeling and Simulation: a practitioner's approach. Taylor and Francis. 2008.*
* *B. Zeigler, H. Praehofer, T. G. Kim. “Theory of Modeling and Simulation”. 2nd Edition. Academic Press. 2000.*

More references about related topics are available at <http://cell-devs.sce.carleton.ca>:

From now on, a complete understanding of DEVS models is assumed. Details about the DEVS formalism can be found in the literature above.

To report errors in this user manual please contact [gwainer@sce.carleton.ca](mailto:gwainer@sce.carleton.ca).

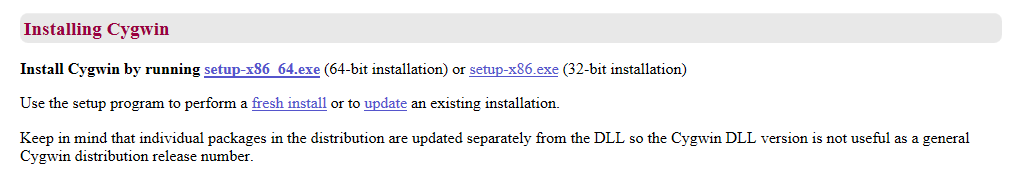
Windows - Installation and example

***NOTE:*** *If we follow these instructions step by step, we will be able to download Cadmium and to compile and execute models in Cadmium DEVS simulator. If we are an expert C++ programmer, we can install the tools in your own different way. Cadmium is a C++ header library only that depends on Boost library. In that case, we can get Cadmium here:*

[*https://github.com/SimulationEverywhere/cadmium*](https://github.com/SimulationEverywhere/cadmium)

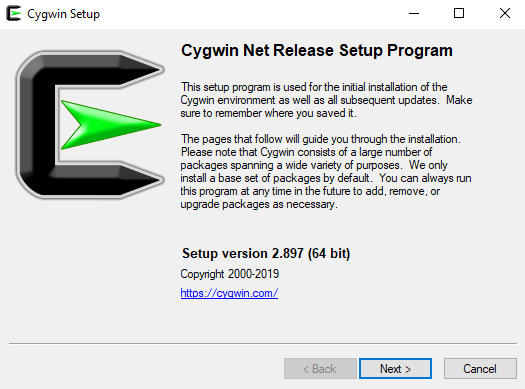
# Installing Cygwin, GCC and Boost

1. Create the folder **C:\*cygwin64***
2. Visit <http://www.cygwin.com/>. Look for the section "Installing Cygwin" and select the appropriate version (32 bit or 64 bit) for your PC. In this example, we will show how to install the 64-bit version.

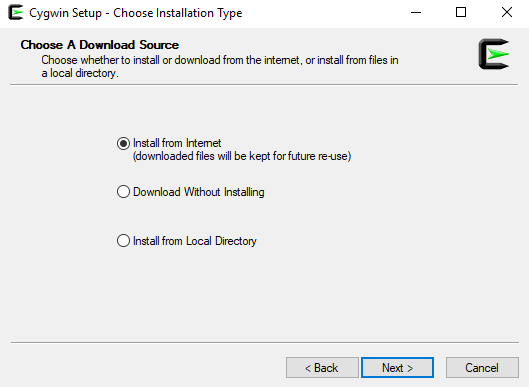


Download the setup file chosen in **C:\*cygwin64***. Based on the OS version we will get a file named setup-x86\_64.exe (64-bit installation) or setup-x86.exe (32-bit installation)

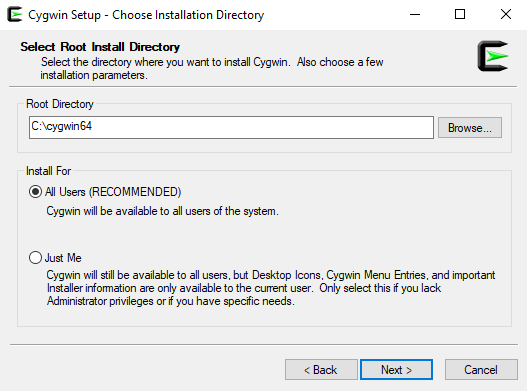
1. Execute setup-x86\_64.exe (64-bit installation) or setup-x86.exe (32-bit installation) and click on   
   “Next >”. We will see the following welcome screen.



1. Select the option "Install from Internet" and click on “Next >”



1. We need to select the Root Install directory for storage of Cygwin files. Choose the default (c:\cygwin64, as seen in the screenshot, and “All Users (RECOMMENDED)”. Click on “Next >”

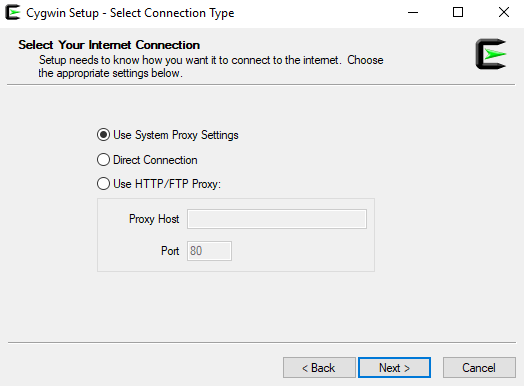


1. Choose your preferred directory for storage of Cygwin local package directory as in the screenshot (i.e. the folder we just created) and click on “Next >”

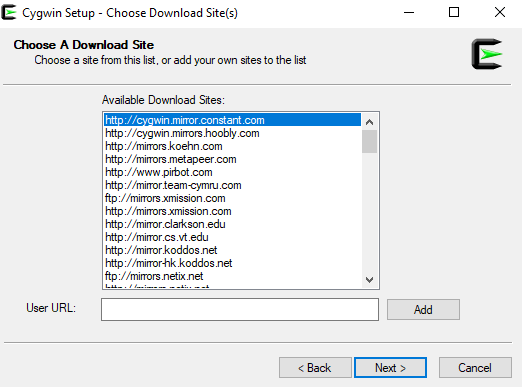
A screenshot of a cell phone

Description automatically generated

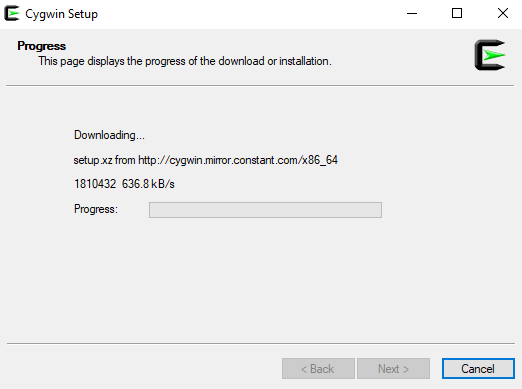
1. Select the option “Use System Proxy Setting” and click on “Next >”



1. After a few seconds, the following window will appear. Choose a Download Site as in the screenshot. Click on “Next >” (in this case, http://cygwin.mirror.constant.com)



1. Cygwin will start the installation process. The following window will appear

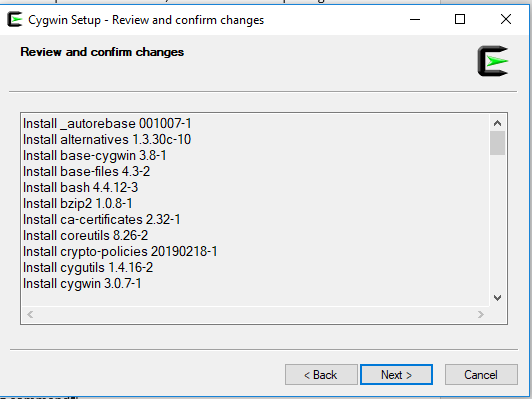


1. When we get the following window, if we click on “All”, we will see all the existing packages. Do not choose anything; simply click “Next >” leaving everything as default (as in the screenshot). This will install the default tools and libraries.

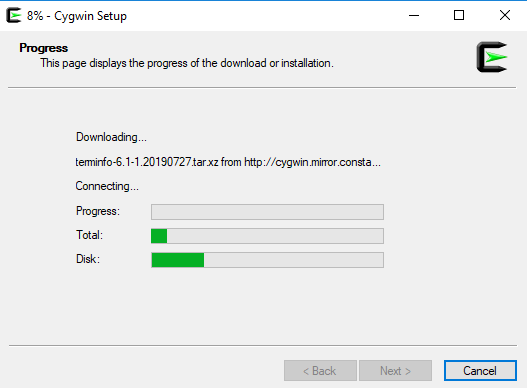
A screenshot of a cell phone

Description automatically generated

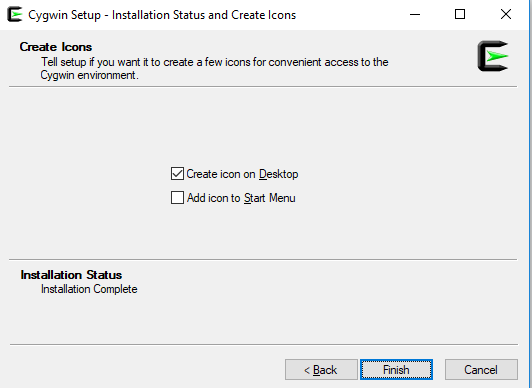
1. The following window will appear. Click on “Next >”



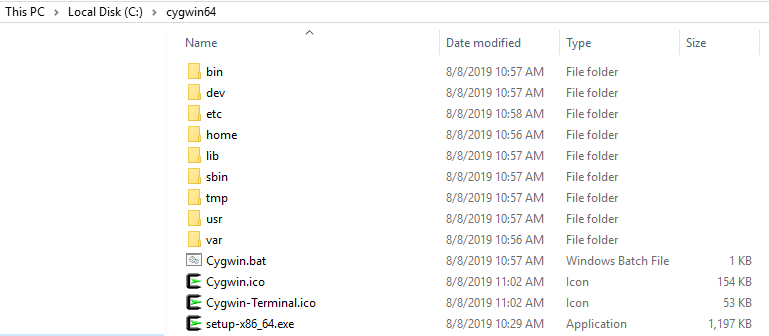
1. The progress window below will appear.



1. Once the installation finishes, select the option “Create icon on Desktop” to easily access the Cygwin terminal. Click on “Finish”



1. Once the installation finishes, if we open the cygwin64 folder, it should have the following content.  
   Make sure you copied setup-x86\_64.exe in the cygwin64 folder (or setup-x86.exe for 32bits installations).



1. Open the windows terminal (Command Prompt; type “cmd” on your Windows search).

Type

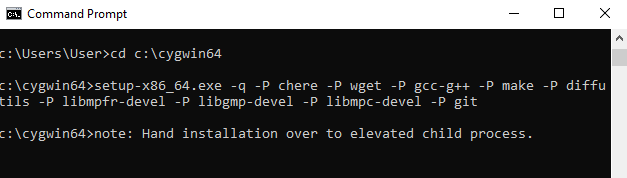
cd c:\cygwin64

For the **64-bit installation**, type:

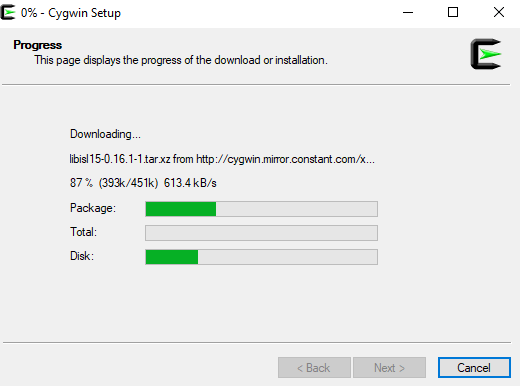
setup-x86\_64.exe -q -P chere -P wget -P gcc-g++ -P make -P diffutils -P libmpfr-devel -P libgmp-devel -P libmpc-devel -P git

**(For 32-bit installation, replace by** setup-x86.exe)

It will install all the necessary libraries and the last version of gcc/g++ compiler.



1. A Progress window will pop up while all the required packages along with their dependencies are downloaded and installed, as in the following screen capture.



The installation process will take several minutes. Once the installation process finishes, the window will disappear automatically, and we can close the Command Prompt.

1. Run Cygwin on your desktop, in administrator mode (right-click on the desktop icon and select the option “Run as administrator”; we can also use c:\cygwin64 and run the script “cygwin.bat” in Administrator mode). The skeleton files will be created:

Copying skeleton files.

These files are for the users to personalise their cygwin experience.

They will never be overwritten nor automatically updated.

'./.bashrc' -> '/YOURDIRECTORY//.bashrc'

'./.bash\_profile' -> '/YOURDIRECTORY//.bash\_profile'

'./.inputrc' -> '/YOURDIRECTORY//.inputrc'

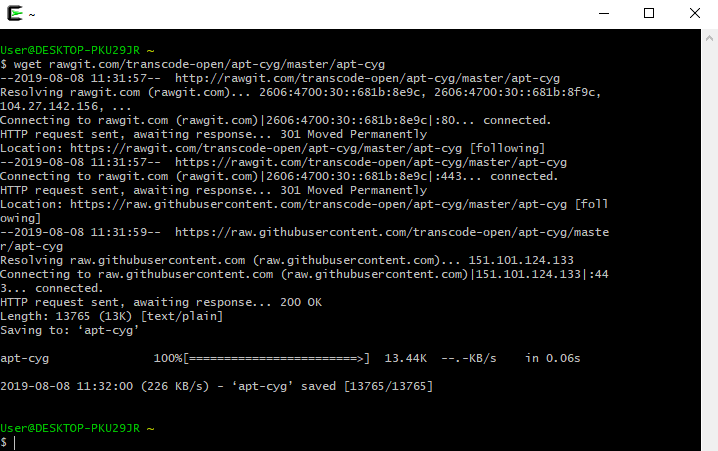
'./.profile' -> '/YOURDIRECTORY//.profile'

YOURDIRECTORY~

$

1. Type the following command on the terminal and press “Enter” (in this case, we show an example for user “User” running Cygwin on the Desktop):

wget rawgit.com/transcode-open/apt-cyg/master/apt-cyg



1. Type the following command and press “Enter”

install apt-cyg /bin

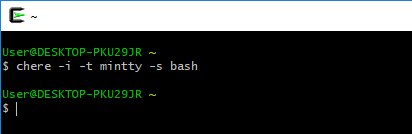


"apt-cyg" is a command in Cygwin similar to the "sudo apt-get" command in Linux. It is used to install packages, update them, list them, etc.

1. Type the following command and press “Enter”

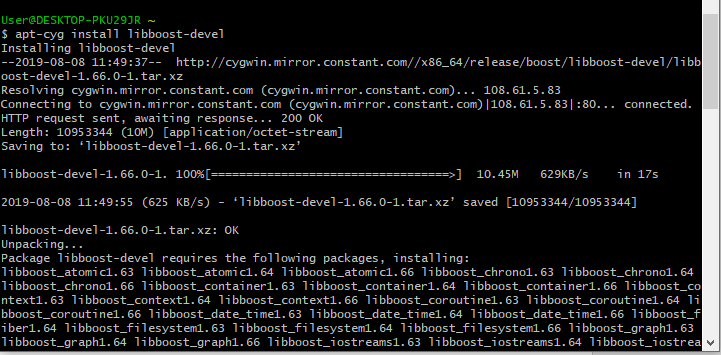
chere -i -t mintty -s bash

This will allow us to open a Cygwin bash terminal from any folder in your Windows File Explorer or other applications.

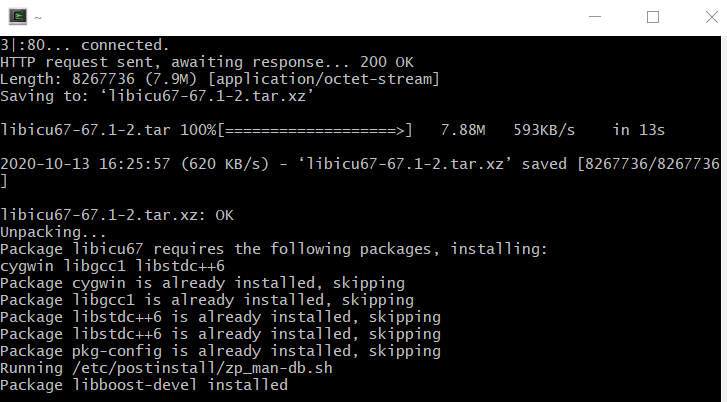


1. Type the following command on Cygwin terminal and press “Enter”. This installs the Boost Library. A progress message will show the installation. **Note that this process may take several minutes. It installs all packages in the boost library. The figure shows the start/end of the process only (and this could vary depending on your own installation).**

apt-cyg install libboost-devel

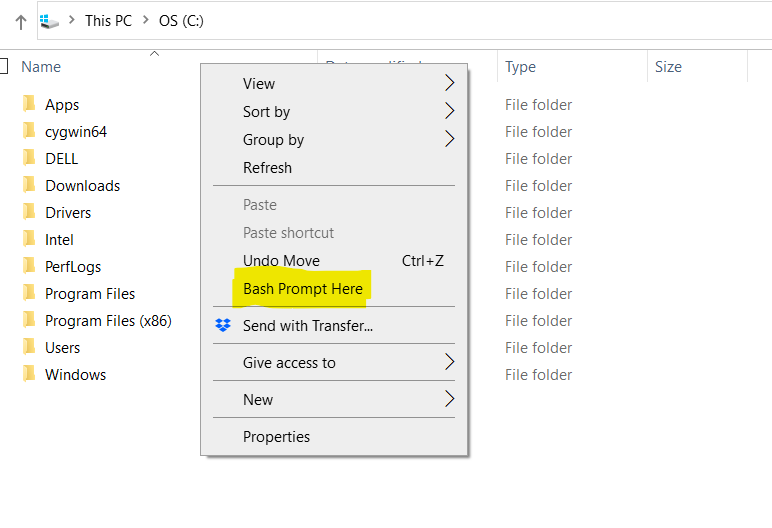


**…..**

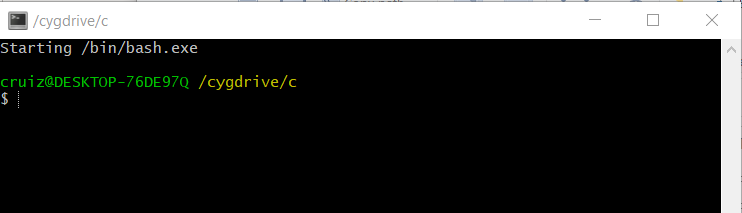
****

# Downloading and installing the Cadmium Simulator

1. Go to the C drive. Inside the C folder, right-click + “Bash Prompt Here”



This will open the cygwin terminal in the C folder.



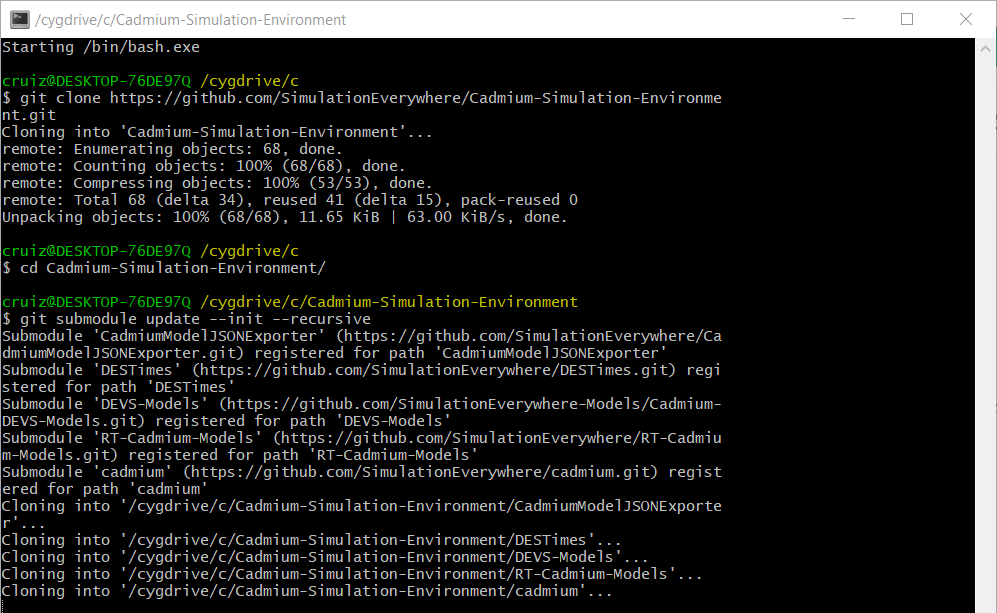
1. Type the following commands:

git clone <https://github.com/SimulationEverywhere/Cadmium-Simulation-Environment.git>

cd Cadmium-Simulation-Environment/

git submodule update --init --recursive

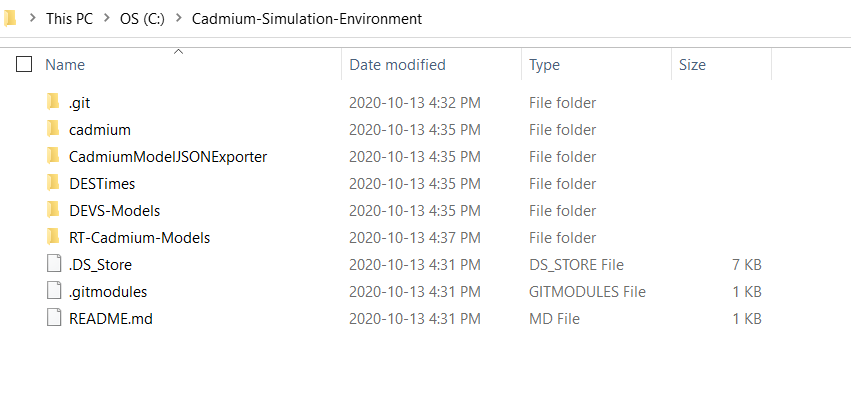
Note that this may take 15-30 min (or longer, depending on your internet speed).



….



Now we have Cadmium set up. If we open the folder Cadmium-Simulation-Environment (located in C, if you followed the instructions in this manual), it has to look as follows:

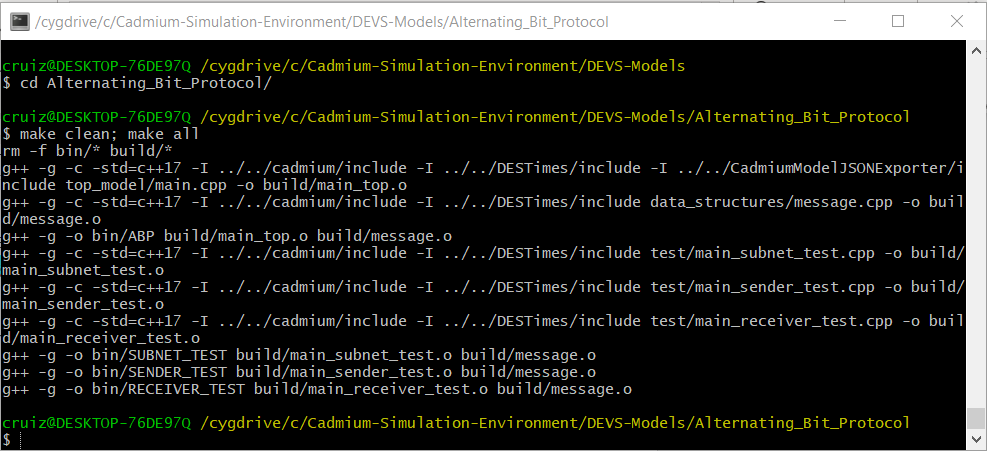


# Compiling and Running a Cadmium DEVS Model

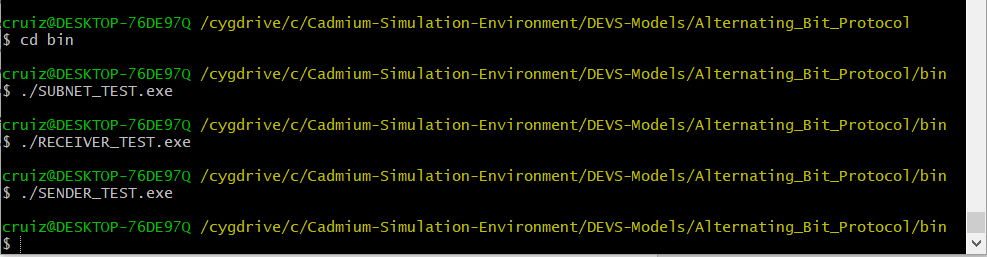
When we download Cadmium, we obtain a Model Library (Folder: DEVS-Models). We will use the Alternating\_Bit\_Protocol model found in that directory as an example to show how to compile a Cadmium model and how to run the tests for that model.

1. Compile the project and the tests
   1. Open a Bash Prompt inside the folder Alternating\_Bit\_Protocol:  
      Inside the Alternating\_Bit\_Protocol folder, right-click + “Bash Prompt Here”
   2. To compile the project and the tests, type in the Bash Prompt:

make clean; make all



1. Run tests
   1. A subfolder, called *bin*, has been created. The simulation examples we will execute are in that directory (cd bin).
   2. To run the subnet test, type in the Bash Prompt:  
      ./SUBNET\_TEST.exe
   3. To run the receiver test, type in the Bash Prompt:  
      ./RECEIVER\_TEST.exe
   4. To run the sender test, type in the Bash Prompt:  
      ./SENDER\_TEST.exe



* 1. To check the output of the tests, go to the folder “../simulation\_results” and open the respective files.

1. Run the top model
   1. Inside the subfolder “*bin*”, type  
      ./ABP.exe ../input\_data/input\_abp\_1.txt



* 1. To check the output of the model, go to the folder simulation\_results and open "ABP\_output\_messages.txt" and "ABP\_output\_state.txt"

1. To run the model with different inputs
   1. Create new .txt files with the same structure as input\_abp\_0.txt or input\_abp\_1.txt in the folder input\_data
   2. Run the model using the instructions in step 3
   3. If we want to keep the output, rename abp\_output.txt. Otherwise, it will be overwritten when we run the next simulation.

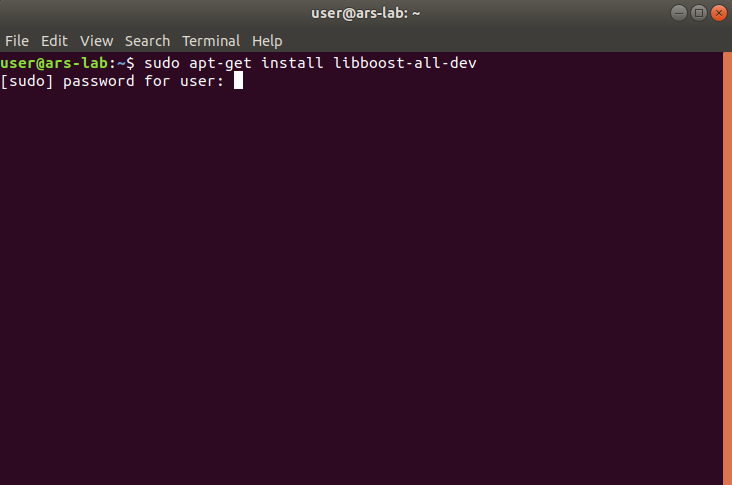
Ubuntu - Installation and example

# System requirements

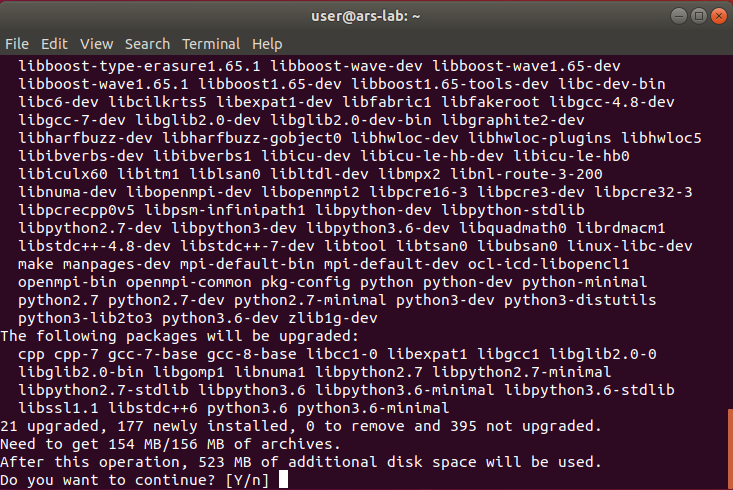
1. Ubuntu 16.04 or higher
2. RAM 16GB (we will be able to run small models with 4GB ram)

Installing Boost

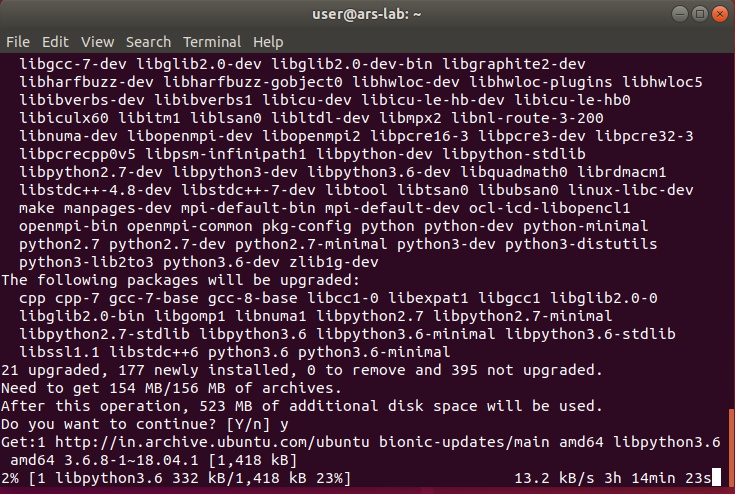
1. Open the Ubuntu terminal. To open Ubuntu terminal press: "*Ctrl + Alt + t*".
2. Type the following command in the Ubuntu terminal screen that appears, and press ENTER  
   sudo apt-get install libboost-all-dev
3. Type the administrative password, i.e. the password we use for signing in into your Ubuntu account and press ENTER.



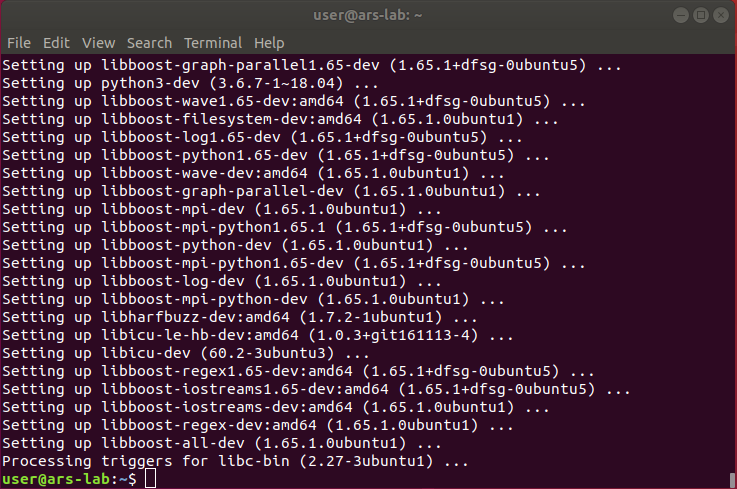
1. The installation begins. After a while, the installation is temporarily paused, and the following question appears: "Do we want to continue?", type: y and then press ENTER to resume the installation process.



1. Wait until the installation is finished.



….

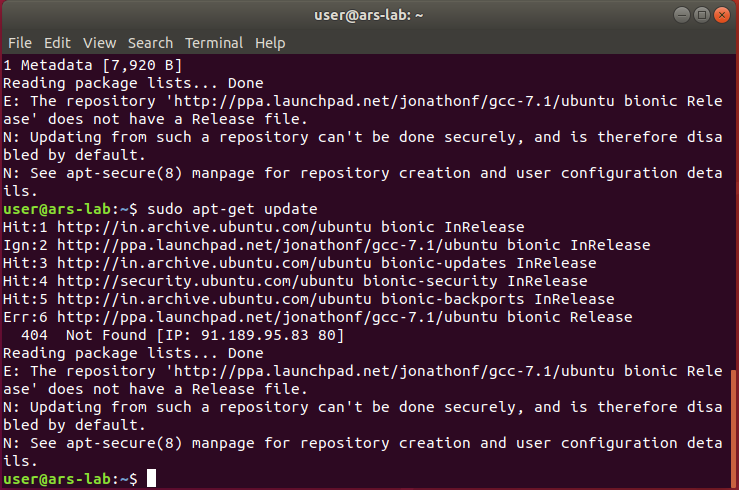


# Installing g++

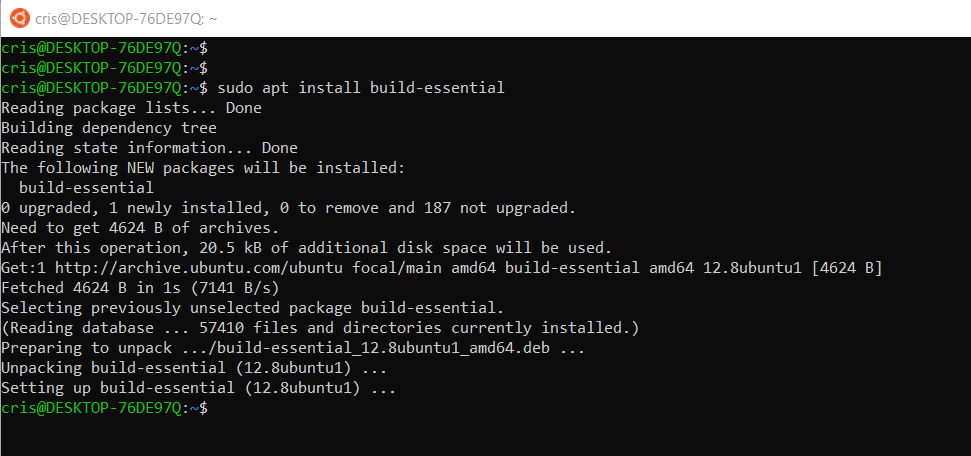
Cadmium is tested using g++7.2 compiler. Previous versions of g++ do not work because they cannot compile C++17 code. We recommend using the latest version of the compiler.

**Instructions to install the last version of gcc and g++**

1. Open Ubuntu terminal. To open Ubuntu terminal press: “Ctrl + Alt + t”. Do not close the terminal until the installation process is complete.
2. Type the following command on the terminal and press ENTER:   
   sudo apt update
3. Enter the administrative password if we are asked. Wait until the installation is finished

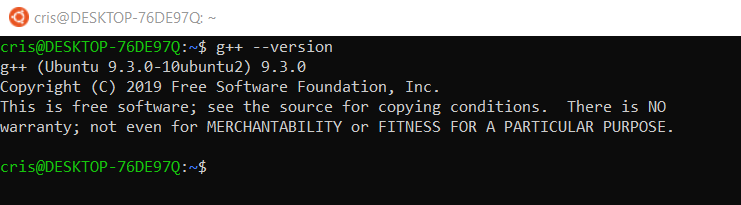


1. Type the following command in the Ubuntu terminal and press ENTER:  
   sudo apt install build-essential



1. To verify that the latest version has been installed on your computer, type the following command in the terminal and press ENTER: (You must see a version of g++ that is 7.2 or higher).

g++ --version



# Installing Git

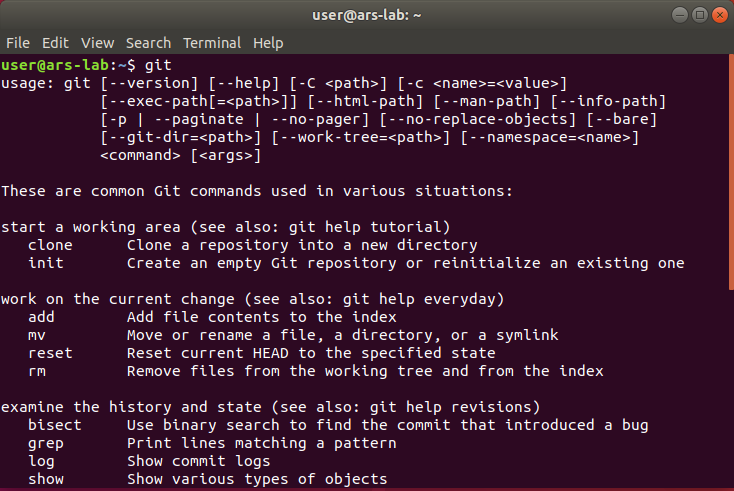
1. To check if your computer has Git installed in, open Ubuntu terminal by pressing: “Ctrl + Alt + t”. Do not close the terminal until the installation process is complete.
2. Type the following command and press ENTER:   
   git

If git is not installed, the terminal looks like this.

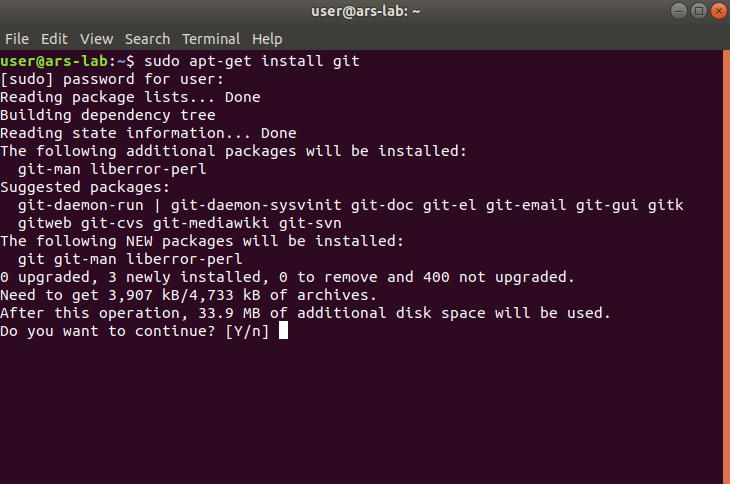
A screenshot of a cell phone

Description automatically generated

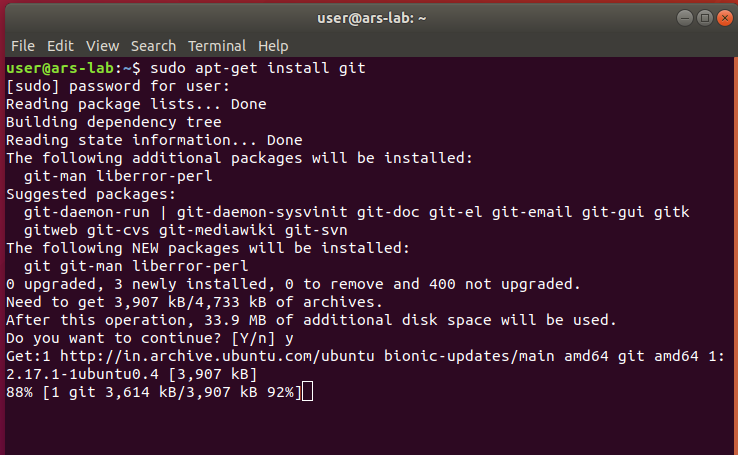
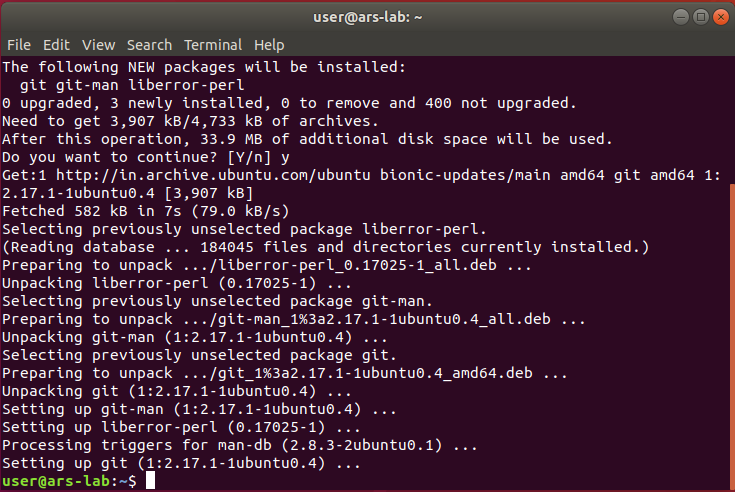
If git is already installed, the terminal looks as follows and we can skip the rest of this section.



1. To install git on your computer, type the following command   
   sudo apt-get install git
2. Enter the administrative password, i.e. the password we use for signing in into your Ubuntu account and press ENTER. The installation process begins.



1. After a while, the installation is temporarily paused, and the following question appears on the Ubuntu terminal "*Do we want to continue?*", type: y and then press ENTER to resume the installation process. Wait until the installation process is finished.

  
****

# Installing the ‘make’ command

1. Open the terminal (Press CTRL + Alt + t) and the type following command:

sudo apt-get install make

1. Enter the administrative password if we are asked. Wait until the installation is finished

A screenshot of a cell phone

Description automatically generated

# Downloading and installing the Cadmium Simulator

1. Create a new folder in the Home directory and name it as “CADMIUM”.

A screenshot of a cell phone

Description automatically generated

1. Open Ubuntu terminal by pressing: “Ctrl + Alt + t”. Type the following commands:

cd CADMIUM/

git clone <https://github.com/SimulationEverywhere/Cadmium-Simulation-Environment.git>

cd Cadmium-Simulation-Environment

git submodule update --init --recursive

This may take 15-30 min (or more, depending on your internet speed). It downloads more modules than the ones shown in the figure.

A screenshot of a cell phone

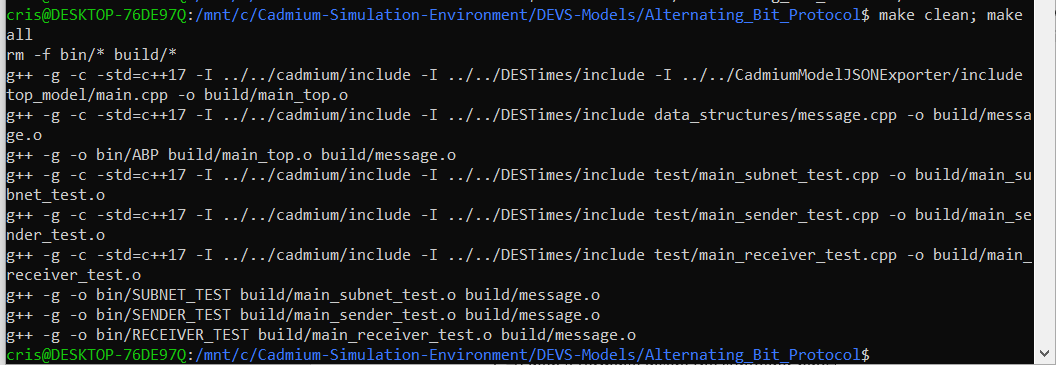
Description automatically generated

Now we have Cadmium set up.

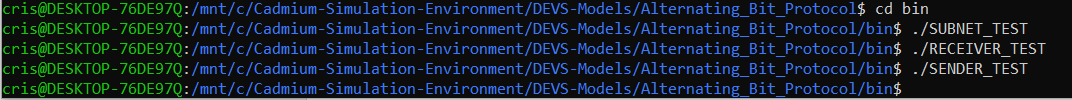
# Compiling and Running a Cadmium DEVS Model

As we could see, when we download the Cadmium Simulation Environment it comes with a Model Library (Folder: DEVS-Models). We will use the Alternating\_Bit\_Protocol model as an example to show how to compile a Cadmium model and how to run the tests and the model.

1. Compile the project and the tests  
   1. Open terminal inside the folder Alternating\_Bit\_Protocol:  
      Inside the Alternating\_Bit\_Protocol folder, (Press CTRL + Alt + t).
   2. To compile the project and the tests, type:  
      make clean; make all



1. Run tests
   1. Open a terminal inside the subfolder bin:  
       Inside the bin folder, (Press CTRL + Alt + t) to open the terminal.
   2. To run the subnet test, type:  
       ./SUBNET\_TEST
   3. To run the receiver test, type:  
       ./RECEIVER\_TEST
   4. To run the sender test, type:  
       ./SENDER\_TEST



* 1. To check the output of the tests, go to the folder *simulation\_results* and open the respective files

1. Run the top model
   1. Open a terminal inside the subfolder bin:  
       Inside the bin folder, (Press CTRL + Alt + t) to open the terminal.
   2. To run the model, type:  
       ./ABP ../input\_data/input\_abp\_1.txt



* 1. To check the output of the model, go to the folder *simulation\_results* and open "ABP\_output\_messages.txt" and "ABP\_output\_state.txt"

1. To run the model with different inputs  
   1. Create new .txt files with the same structure as input\_abp\_0.txt or input\_abp\_1.txt in the folder input\_data
   2. Run the model using the instructions in step 3
   3. If we want to keep the output, rename "ABP\_output\_messages.txt" and "ABP\_output\_state.txt". Otherwise, it will be overwritten when we run the next simulation.

MacOS - Installation and example

System requirements

1. MacOS 10.11 or higher
2. RAM 16GB (we will be able to run small models with 4GB RAM)

Installing Command Line Tools

In order to run Cadmium, we need to install different tools, such as make, git, or g++. To do so, follow the next steps:

1. Open a terminal:
   1. Use the keyboard shortcut *“Command + Space”* to open Spotlight Search.
   2. Type in “terminal”.
   3. You should see the Terminal application under Top Hit at the top of your results. Double-click it and Terminal will open.
2. Type the following command in the terminal screen, and press ENTER  
   xcode-select –install
3. A software update popup window will appear asking for permission to install the command line developer tools. Click “Install” to download them and agree to the Terms of Service (after reading them, of course).

# Installing Homebrew and Boost

Cadmium uses different C++ source libraries provided by Boost. We have to install first Homebrew, a package manager for MacOS.

1. Open a terminal:
   1. Use the keyboard shortcut *“Command + Space”* to open Spotlight Search.
   2. Type in “terminal”.
   3. You should see the Terminal application under Top Hit at the top of your results. Double-click it and Terminal will open.
2. Type the following command and press ENTER: /usr/bin/ruby –e “$(curl –fsSL https://raw.githubusercontent.com/Homebrew/install /master/install)”
3. Install Boost typing the following command: brew install boost

# Downloading and installing the Cadmium Simulator

1. Create a new folder in the Home directory and name it as “CADMIUM”.

A screen shot of a computer

Description automatically generated

1. Open a terminal:
   1. Use the keyboard shortcut *“Command + Space”* to open Spotlight Search.
   2. Type in “terminal”.
   3. You should see the Terminal application under Top Hit at the top of your results. Double-click it and Terminal will open.
2. Type the following commands:

cd CADMIUM/

git clone <https://github.com/SimulationEverywhere/Cadmium-Simulation-Environment.git>

cd Cadmium-Simulation-Environment/

git submodule update --init --recursive

This may take 15-30 min or longer based on your internet connection. It downloads more modules than the ones shown in the figure.

A screenshot of a cell phone

Description automatically generated

Now we have Cadmium set up.

# Compiling and Running a Cadmium DEVS Model

As we could see, when we download the Cadmium Simulation Environment it comes with a Model Library (Folder: DEVS-Models). We will use the Alternating\_Bit\_Protocol model as an example to show how to compile a Cadmium model and how to run the tests and the model.

1. Compile the project and the tests
   1. Open a terminal:
      1. Use the keyboard shortcut *“Command + Space”* to open Spotlight Search.
      2. Type in “terminal”
      3. You should see the Terminal application under Top Hit at the top of your results. Double-click it and Terminal will open
   2. Type the following to change the working directory to Cadmium-DEVS-Models/ABP folder:  
      cd CADMIUM/Cadmium-Simulation-Environment/DEVS-Models/Alternating\_Bit\_Protocol
   3. To compile the project and the tests, type:  
      make clean; make all
2. Run tests
   1. Open a terminal:
      1. Use the keyboard shortcut *“Command + Space”* to open Spotlight Search.
      2. Type in “terminal”
      3. You should see the Terminal application under Top Hit at the top of your results. Double-click it and Terminal will open
   2. Type the following to change the working directory to Cadmium-DEVS-Models/ Alternating\_Bit\_Protocol/bin folder:  
      cd CADMIUM/Cadmium-Simulation-Environment/DEVS-Models/Alternating\_Bit\_Protocol/bin
   3. To run the subnet test, type:  
       ./SUBNET\_TEST
   4. To run the receiver test, type:  
       ./RECEIVER\_TEST
   5. To run the sender test, type:  
       ./SENDER\_TEST
   6. To check the output of the tests, go to the folder *simulation\_results* and open the respective files
3. Run the top model
   1. Open a terminal:
      1. Use the keyboard shortcut *“Command + Space”* to open Spotlight Search.
      2. Type in “terminal”.
      3. You should see the Terminal application under Top Hit at the top of your results. Double-click it and Terminal will open
   2. Type the following to change the working directory to   
      Cadmium-DEVS-Models/Alternating\_Bit\_Protocol /bin folder:  
      cd CADMIUM/Cadmium-Simulation-Environment/DEVS-Models/ Alternating\_Bit\_Protocol/bin
   3. To run the model, type:  
       ./ABP ../input\_data/input\_abp\_1.txt
   4. To check the output of the model, go to the folder *simulation\_results* and open "ABP\_output\_messages.txt" and "ABP\_output\_state.txt"
4. To run the model with different inputs
   1. Create new .txt files with the same structure as input\_abp\_0.txt or input\_abp\_1.txt in the folder input\_data
   2. Run the model using the instructions in step 3

If we want to keep the output, rename "ABP\_output\_messages.txt" and "ABP\_output\_state.txt". Otherwise, it will be overwritten when we run the next simulation.

DEVS Model definition: An Example

This section describes the mechanism to define and incorporate new atomic models into Cadmium. These models can be used to interact directly with other models or to be part of a DEVS coupled model.

Atomic models have to be defined in an .hpp file and coded in C++. These .hpp files can be created with our preferred text editor. We will start defining a simple example of an atomic model. We use this example to explain how to define an atomic model. In the following sections, we will continue using this example to explain how to define a coupled model, how to define simulation loggers and how to call the simulator.

# Subnet: an atomic model example implemented in Cadmium

When we download Cadmium following the instructions in this Manual, a library of models will be downloaded. These models are available in the folder called *DEVS-Models*. One of them is an Alternate Bit Protocol (ABP) model, and it is stored in the folder Alternating\_Bit\_Protocol. Repository available at: <https://github.com/SimulationEverywhere/Cadmium-Simulation-Environment>. We will use this Alternating\_Bit\_Protocol example to explain how to implement models in Cadmium.

Figure 1 shows the ABP model coupled model. The Alternating Bit communication protocol tries to provide reliable transmission on an unreliable network. The ABP model consists of 3 components: A sender, which transmits messages; a network, and a receiver, which receives the messages transmitted by the sender and returns acknowledgement messages (positive or negative). The network is decomposed further to two subnets corresponding to the sending and receiving channels, respectively. The sender and the receiver communicate with each other through the network component.

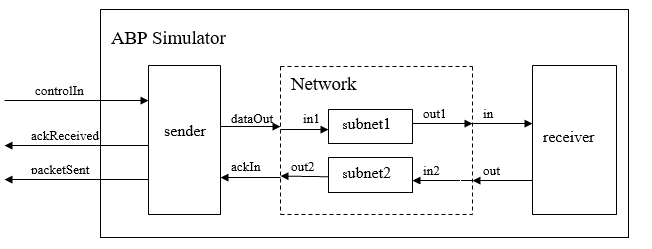


Figure 1 ABP Simulator coupled model

In this section, we will discuss the definition of the *subnet* atomic model, as an example to introduce the definition of atomic models in Cadmium. The remaining models are available in the simulator package.

The Subnet atomic model uses one input port and one output port, and the model passes the data it receives after a time delay. To model the unreliability of the network, only approximately 95% of the data will be transferred (i.e. 5% of the data will be lost through the subnet).

Figure 2 shows the subnet model implementation in Cadmium.

#ifndef \_SUBNET\_HPP\_\_

#define \_SUBNET\_HPP\_\_

#include <cadmium/modeling/ports.hpp>

#include <cadmium/modeling/message\_bag.hpp>

#include <limits>

#include <assert.h>

#include <string>

#include <random>

#include "../data\_structures/message.hpp"

using namespace cadmium;

using namespace std;

/\*\*\*\*\* (1) \*\*\*\*\*/

//Port definition

struct Subnet\_defs{

struct in : public in\_port<Message\_t> {};

struct out : public out\_port<Message\_t> {};

};

/\*\*\*\*\* (2) \*\*\*\*\*/

template<typename TIME> class Subnet{

public:

using input\_ports=tuple<typename Subnet\_defs::in>;

using output\_ports=tuple<typename Subnet\_defs::out>;

/\*\*\*\*\* (3) \*\*\*\*\*/

// state definition

struct state\_type{

bool transmitting;

Message\_t packet;

int index;

};

state\_type state;

/\*\*\*\*\* (4) \*\*\*\*\*/

// default constructor

Subnet(){

state.transmitting = false;

state.index = 0;

}

/\*\*\*\*\* (5) \*\*\*\*\*/

// internal transition

void internal\_transition() {

state.transmitting = false;

}

/\*\*\*\*\* (6) \*\*\*\*\*/

// external transition

void external\_transition(TIME e, typename   
 make\_message\_bags<input\_ports>::type mbs) {

vector<Message\_t> bag\_port\_in;

bag\_port\_in = get\_messages<typename Subnet\_defs::in>(mbs);

state.index++;

if ((double)rand() / (double) RAND\_MAX < 0.95){

state.packet = bag\_port\_in[0];

state.transmitting = true;

}else{

state.transmitting = false;

}

}

/\*\*\*\*\* (7) \*\*\*\*\*/

// confluent transition

void confluence\_transition(TIME e, typename

make\_message\_bags<input\_ports>::type mbs) {

internal\_transition();

external\_transition(TIME(), move(mbs));

}

/\*\*\*\*\* (8) \*\*\*\*\*/

// output function

typename make\_message\_bags<output\_ports>::type output() const {

typename make\_message\_bags<output\_ports>::type bags;

vector<Message\_t> bag\_port\_out;

bag\_port\_out.push\_back(state.packet);

get\_messages<typename Subnet\_defs::out>(bags) = bag\_port\_out;

return bags;

}

/\*\*\*\*\* (9) \*\*\*\*\*/

// time\_advance function

TIME time\_advance() const {

TIME next\_internal;

if (state.transmitting) {

next\_internal = TIME("00:00:03:000");

}else {

next\_internal = numeric\_limits<TIME>::infinity();

}

return next\_internal;

}

/\*\*\*\*\* (10) \*\*\*\*\*/

friend ostringstream& operator<<(ostringstream& os,  
 const typename Subnet<TIME>::state\_type& i) {

os << "index: " << i.index << " & transmitting: " << i.transmitting;

return os;

}

};

#endif // \_SUBNET\_HPP\_\_

Figure 2. Cadmium implementation of the Subnet atomic model

#### Creating the hpp where the atomic model is defined

We first create the subnet.hpp file, using the structure provided in Appendix A.

It is important to notice that we cannot have two atomic models with the same name. We use a macro to avoid multiple “includes” in the atomic model (in this case, we call it \_\_SUBNET\_HPP\_\_).

Then, we need to include the simulator libraries that provide services to define new ports (**<cadmium/modeling/ports.hpp>)** and to handle bags of messages **(<cadmium/modeling/message\_bag.hpp>).** We then include any C++ library needed to implement the model. In this example, we use the limits library to set the time advance value to infinity (when we need to passivate the model). We also use assert.h, which is useful to stop the simulation and check for errors for non-desired behavior. For example, let us assume that the DEVS atomic model definition states that the inputs to the model are only integers between 0 and 9. When the model is implemented, we can use a conditional statement and the methods provided in assert.h to check that the condition is satisfied. If the condition is not satisfied, the simulation stops, and an error message is displayed. The rest of the libraries provides some services we use in this specific C++ implementation. In this example, we use String as we need to manipulate strings, and random to generate random numbers. We use those functions to generate different delays in message transmission.

In Cadmium, we can transmit messages containing built-in C++ types (integer, float, string, double, bool, etc.) or we can define our own types. In this case, we define the message as a structure. In this example, we include the path to the hpp file where the structure is defined (e.g., #include "../data\_structures/message.hpp"). If we define more than one type of message (i.e. structure), we will need to include all the ones used in the model. We will explain the content of message.hpp in the next section.

Finally, we declare the namespaces we are using, in this case: cadmium and std (otherwise, every time we use a method/service from the standard C++ library (std), we have to write std::; the same for cadmium::).

#### Declaring ports

As seen in Figure 1, the Subnet uses one input port called “in” and one output port called “out”; both ports carry the same type of message, in this case, a C++ structure called Message\_t, which is declared in message.hpp.

As shown in Figure 3, Message\_t uses two integer variables: packet and bit.

struct Message\_t{

Message\_t(){}

Message\_t(int i\_packet, int i\_bit)

:packet(i\_packet), bit(i\_bit){}

int packet;

int bit;

};

istream& operator>> (istream& is, Message\_t& msg);

ostream& operator<<(ostream& os, const Message\_t& msg);

Figure 3. Message\_t data type declaration

The struct Message\_t shown here is defined in message.hpp. It is a C++ structure with two components: packet (which contains the packet number sent through the network), and bit (which contains an alternating bit used to identify two consecutive packets to provide reliability in the transmission). Inside the structure, we also have two constructors. The default one (without parameters) generates a variable of type Message\_t filled with “garbage”. The second one also generates a variable of type Message\_t, but it is filled with the values used to call the constructor.

Inside message.hpp we also declare operators << and >>. We use >> to read data from a file and fill the structure – optional if you do not have input data coming from a file – and << to save the content of the structure in a file – needed by the simulator to log the messages.

The two operators are implemented inside a new file called message.cpp (Figure 4). For the output operator, we need to specify how we want to output the content of the struct. In this case, we output “the packet space the bit”. For the input operator, we need to specify in which order the data we read comes. In this case, we will have two elements, the first one will be assigned to the packet and the second one to the bit.

It is important to define the >> operator when we are not using built-in data types for messages and we need to read inputs for the model from a text file. Considering the current definition of the operator, we need to define the inputs in the input file as “TIME packet\_value bit\_value”. If we define the inputs in another order, for example, “TIME bit\_value packet\_value, packet will not contain “bit\_value” and bit will contain “packet\_value”.

//Output the content of the structure

ostream& operator<<(ostream& os, const Message\_t& msg) {

os << msg.packet << " " << msg.bit;

return os;

}

//Fill the structure

istream& operator>> (istream& is, Message\_t& msg) {

is >> msg.packet;

is >> msg.bit;

return is;

Figure 4. Implementation of the << and >> operators

Under “//Port definition” (/\*\*\*\*\* (1) \*\*\*\*\*/) we define the ports used by the atomic model. We define them as a structure that contains all the input and output ports of the atomic model (in this case called Subnet\_defs). We use a structure (named using the convention “AtomicModelName\_defs”; in this example, Subnet\_defs) to avoid compilation problems (multiple declaration errors). For example, if we have two different atomic models with a port called in and we declare the ports outside the structure, we will get a compilation error stating ambiguous definition for type “in”. We avoid this issue declaring the ports inside a structure with a unique name.

Each port is defined as a structure that inherits from the template structures out\_port and in\_port defined in the simulator, specifying the type of message handled by the port. In this case, we defined an output port called out that handles messages of type Message\_t and an input port called in that also handles messages of type Message\_t.

#### Declaring the atomic model

Under “/\*\*\*\*\* (2) \*\*\*\*\*/”, we define the atomic model as a C++ class that implements the model state and all the DEVS functions following the template in Appendix A.

The models are implemented following a template-based C++ programming style. This style allows us to use different *time* classes without changing the model implementation. For experienced users, it also allows implementing models that can be instantiated with different messages types. For example, we could implement a subnet model that can transmit any type of message and not just messages of type Message\_t.

We give a name to the class used to represent the model; in this case, we call it Subnet and we define the input and output ports in the class. Everything inside the class is public, as the simulator has to access the methods of the class to execute the simulation. As discussed earlier, the ports were declared inside the structure Subnet\_defs. To access the input port, we would need to use Subnet\_defs::in and to use the output port Subnet\_defs::out.

Once we have declared the types of ports we have (i.e. Subnet\_defs), we need to assign those ports to the corresponding atomic model (in this case, defined by the class Subnet). We assign them as follows:

using input\_ports=tuple<typename Subnet\_defs::in>;

using output\_ports=tuple<typename Subnet\_defs::out >;

The C++ keyword using that allows us to rename a data type. Each atomic model must define their input and output ports as a data type called “input\_ports” and “output\_ports”, respectively. We need to use these specific names because the simulator will use them to check that the atomic model has all the needed components and that the ports are properly defined (e.g. there are not two input ports with the same name). Both the input and out ports are defined as a tuple (tuple<>), a C++ object that packs elements of possibly different types together in a single object. We can see it as a vector with elements of different types. Because of this, we need to specify the type of each of the elements in the tuple. In this example, both tuples use only one element. Subnet\_defs::in is the type of the input port tuple and Subnet\_defs::out is the type of the output port type. The typename specifies that Subnet\_defs::in and Subnet\_defs::out are data types that will overwrite the template class in the simulator.

Under “/\*\*\*\*\* (3) \*\*\*\*\*/”, we declare the state variables of the model inside a structure called state\_type. All the state variables of the model must be declared inside a structure called state\_type, and a single state variable of type state\_type name state must be defined. We need to use these specific names because they are explicitly used by the simulator to check that the model is implemented according to the DEVS formalism. The simulator verifies if this structure (which represents the model’s state) is updated inside the output function or the time advance function (two invalid operations according to DEVS specifications).

In this example, the state comprises three variables: transmitting, packet, and index. Transmitting is a Boolean used to define that the model has something to output. Packet stores the packet to be sent. Index counts the packets that went through the network. Once the state structure has been declared, we create an instance of the structure called state.

Under “/\*\*\*\*\* (4) \*\*\*\*\*/”, we define the constructor for the model, including the initial state. We must define a default constructor (i.e. without parameters).

We define the default constructor Subnet(). We set index to 0 and transmitting to false. The content of “packet” is “garbage”, as we do not care about the content of packet until an input message arrives at the atomic model.

We then define the behavior of all the DEVS functions.

* **Internal transition function (**/\*\*\*\*\* (5) \*\*\*\*\*/**)**: defined by internal\_transition(), here the model sets the state variable transmitting to false.
* **External transition function (**/\*\*\*\*\* (6) \*\*\*\*\*/**)**: defined in external\_transition, it takes two parameters: the elapsed time (e) and a bag of message (mbs). The declaration of the bag of messages is as follows: typename make\_message\_bags<input\_ports>::type mbs. As we already mentioned, typename indicates that the expression that follows is a data type. make\_message\_bags<> is a template data type declared in the simulator in <cadmium/modeling/message\_bag.hpp>, used to declare a bag of messages for input or output ports. We need to instantiate the template with the word input\_ports to define the input bag, using ::type. The parameter declaration, in this case, declares mbs as a tuple whose elements are the message bags on the input ports. Here, mbs is a tuple of one bag: the message bag in port in. The messages inside the set of messages in the bag are stored in a C++ vector.

We use get\_messages<typename Subnet\_defs::in>(mbs) to get the message bag from the input port in. The method **get\_messages** uses a template parameter for the port we want to access, in this case, the in port, defined by typename Subnet\_defs::in. The function parameter is the bag of messages we want to access, in this case mbs.

In this example, the bag of the input port in has a vector of elements of type Message\_t. We define the auxiliary variable bag\_port\_in (of type vector of Message\_t) to store the bag in the port called in. We use the method **get\_messages** to retrieve the bag. When a message is received, it is stored in the state variable packet. Because we are assuming that we receive a single message, we retrieve the first element of the bag in the in port and we assign it the state variable packet (state.packet = bag\_port\_in[0]). Then, we set transmitting to true with a 95% probability. With a 5% probability, the message received is lost and therefore the model is not transmitting anything (transmitting = false).

* **Confluent transition function (**/\*\*\*\*\* (7) \*\*\*\*\*/**)**: In this example, we use the default implementation for the confluence function, which is executing the model’s internal transition first, and the external transition after that, with an elapsed time equal to zero.
* **Output function (**/\*\*\*\*\* (8) \*\*\*\*\*/**)**: output uses a bag of messages declared as follows: typename make\_message\_bags<output\_ports>::type bags, where typename indicates that the expression that follows is a data type; make\_message\_bags<> is a template data type that the simulator needs (found in **<cadmium/modeling/message\_bag.hpp>**), which is instantiated as output\_ports to define the output bag. Therefore, bags is a tuple whose elements are the message bags available on the different output ports.

We then declare an auxiliary variable (bag\_port\_out) of type vector<Message\_t> to build the message bag for the output port out.

We add the packet stored in the state variable state.packet to bag\_port\_out. We use the C++ method push\_back(), which takes as parameter the element that we want to append to the vector, in this case, state.packet.

Finally, we copy the content of bag\_port\_out to the bag of the port out. To access the content of the bag of the output port out, we use the method get\_messages< >. As we already explained get\_messages uses a template parameter for the port we want to access, in this case, the port out, defined as typename Subnet\_defs::out. The function parameter is the bag of messages we want to access, in this case bags.

* **Time advance function (**/\*\*\*\*\* (9) \*\*\*\*\*/**)**: time\_advance is used to implement the time advance function of the model. In this case, if we are transmitting, the time advance is 3 seconds. If we do not transmit, the model passivates. The model uses next\_internal to store the next time advance. If the state of the model is transmitting, we define the next time advance by updating the variable next\_internal. TIME(“00:00:03:000”) is the time in hours, minutes, seconds, and milliseconds. If the model is not transmitting, we passivate the model, by making next\_internal infinity using the statement numeric\_limits<TIME>::infinity() (a method in the limits library).

**IMPORTANT**: **According to ST-DEVS, only the transition functions (i.e. external, internal and confluence) can be stochastic. The time advance function and the output function MUST be deterministic.**

Once all the DEVS functions are defined, we specify how we want to output the state of the model in the state log (/\*\*\*\*\* (10) \*\*\*\*\*/). In this case, we only display two of the state variables: index and transmitting.

To declare how to log the state of the model, we need to define the << operator for the structure state\_type. The operator takes as input parameters the address of the stream where we want to log (i.e. os) and the state of the model (i.e. i) We use the keyword const before specifying the type of the state to assure that it will not be modified inside the operator. It is important to notice that we need to use typename Subnet<TIME>::state\_type to specify the type of the state. That sentence means that we are accessing the structure state\_type inside the template class Subnet<TIME>. We need to declare the operator using the keyword friend to specify that the function can access the private members of the structure state\_type. In this example, the output of our state looks as follows: “index: index\_value & transmitting: transmiting\_value”.

# Unit testing the Subnet atomic model

To test the subnet atomic model, we will define a coupled model that contains a generator of test cases connected to the model, in order to generate simulations scenarios to verify the execution of the model:

input\_reader

subnet1

out top\_out

out in

Figure 5. Coupled model for testing the subnet atomic model

The coupled model includes two atomic components: input\_reader and subnet1. The input\_reader reads a list of input events stored in a text file that we use to test the Subnet model; the entries in this file have the format “TIME Message”, and it includes one entrance per line. Cadmium provides a template version of this model (iestream.hpp) that need to be instantiated with the type of message we want to read.

In Cadmium, all the coupled models are defined in a cpp file (in this case, the file is named main\_subnet\_test.cpp). The logger definition and the call to the simulator runner are also implemented inside this file.

Figure 6 shows the subnet test coupled model implementation in Cadmium.

//Cadmium Simulator headers

#include <cadmium/modeling/ports.hpp>

#include <cadmium/modeling/dynamic\_model.hpp>

#include <cadmium/modeling/dynamic\_coupled.hpp>

#include <cadmium/modeling/dynamic\_model\_translator.hpp>

#include <cadmium/engine/pdevs\_dynamic\_runner.hpp>

#include <cadmium/logger/common\_loggers.hpp>

//Time class header

#include <NDTime.hpp>

//Messages structures

#include "../data\_structures/message.hpp"

//Atomic model headers

#include "../atomics/subnet.hpp"

#include <cadmium/basic\_model/pdevs/iestream.hpp> //Atomic model for inputs

//C++ libraries

#include <iostream>

#include <string>

using namespace std;

using namespace cadmium;

using namespace cadmium::basic\_models::pdevs;

using TIME = NDTime;

/\*\*\*\*\* (1) \*\*\*\*\*/

/\*\*\*\*\* Define input port for coupled models \*\*\*\*\*/

/\*\*\*\*\* Define output ports for coupled model \*\*\*\*\*/

struct top\_out: public **out\_port**<Message\_t>{};

/\*\*\*\*\* (2) \*\*\*\*\*/

/\*\*\*\*\*\* Input Reader atomic model declaration \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

template<typename T>

class InputReader\_Message\_t : public iestream\_input<Message\_t, T> {

public:

InputReader\_Message\_t () = default;

InputReader\_Message\_t (const char\* file\_path) :   
 iestream\_input<Message\_t,T> (file\_path) {}

};

/\*\*\*\*\* (3) \*\*\*\*\*/

int main(){

/\*\*\*\*\*\* Input Reader atomic model instantiation \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

const char \* i\_input\_data = "../input\_data/subnet\_input\_test.txt";

shared\_ptr<**dynamic::modeling::model**> input\_reader;

input\_reader = **dynamic::translate::make\_dynamic\_atomic\_model** <InputReader\_Message\_t, TIME, const char\*>("input\_reader", move(i\_input\_data));

/\*\*\*\*\* (4) \*\*\*\*\*/

/\*\*\*\*\*\* Subnet atomic model instantiation \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

shared\_ptr<**dynamic::modeling::model**> subnet1;

subnet1 = **dynamic::translate::make\_dynamic\_atomic\_model**<Subnet, TIME>("subnet1");

/\*\*\*\*\* (5) \*\*\*\*\*/

/\*\*\*\*\*\*\*TOP MODEL\*\*\*\*\*\*\*\*/

**dynamic::modeling::Ports** iports\_TOP;

iports\_TOP = {};

**dynamic::modeling::Ports** oports\_TOP;

oports\_TOP = {typeid(top\_out)};

**dynamic::modeling::Models** submodels\_TOP;

submodels\_TOP = {input\_reader, subnet1};

**dynamic::modeling::EICs** eics\_TOP;

eics\_TOP = {}; // \_EIC WOULD GO HERE ; NOT NEEDED BECAUSE IT IS EMTPY IN THIS EXAMPLE

**dynamic::modeling::EOCs** eocs\_TOP;

eocs\_TOP = {

**dynamic::translate::make\_EOC**<Subnet\_defs::out,top\_out>("subnet1")

};

**dynamic::modeling::ICs** ics\_TOP;

ics\_TOP = {

**dynamic::translate::make\_IC**<iestream\_input\_defs<Message\_t>::out,Subnet\_defs::in>(  
 "input\_reader","subnet1")

};

shared\_ptr<**dynamic::modeling::coupled<TIME>**> TOP;

TOP = make\_shared<**dynamic::modeling::coupled<TIME>**>(

"TOP", submodels\_TOP, iports\_TOP, oports\_TOP, eics\_TOP, eocs\_TOP, ics\_TOP

);

/\*\*\*\*\* (6) \*\*\*\*\*/

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* Loggers \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

static ofstream out\_messages("../simulation\_results/subnet\_test\_output\_messages.txt");

struct oss\_sink\_messages{

static ostream& sink(){

return out\_messages;

}

};

static ofstream out\_state("../simulation\_results/subnet\_test\_output\_state.txt");

struct oss\_sink\_state{

static ostream& sink(){

return out\_state;

}

};

using state = **logger::logger<logger::logger\_state, dynamic::logger::formatter<TIME>,** oss\_sink\_state**>;**

using log\_messages = **logger::logger<logger::logger\_messages, dynamic::logger::formatter<TIME>,** oss\_sink\_messages**>;**

using global\_time\_mes = **logger::logger<logger::logger\_global\_time, dynamic::logger::formatter<TIME>,** oss\_sink\_messages**>;**

using global\_time\_sta = **logger::logger<logger::logger\_global\_time, dynamic::logger::formatter<TIME>,** oss\_sink\_state**>;**

using logger\_top = **logger::multilogger<**state, log\_messages, global\_time\_mes, global\_time\_sta**>;**

/\*\*\*\*\* (7) \*\*\*\*\*/

/\*\*\*\*\*\*\*\*\*\*\*\*\*\* Runner call \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

**dynamic::engine::runner**<NDTime, logger\_top> r(TOP, {0});

r.**run\_until**(NDTime("04:00:00:000"));

return 0;

}

Figure 6. Subnet test coupled model implementation

We first include the simulator libraries that provide the different services needed to build and run the simulation. We need to be able to:

* Define new ports (<cadmium/modeling/ports.hpp>)
* Create every element of a coupled model definition: input ports, output ports, submodels, external input couplings, external output couplings and internal couplings (<cadmium/modeling/dynamic\_model.hpp>)
* Define the data types for coupled models <cadmium/modeling/dynamic\_coupled.hpp>
* Create new instances of atomic models and make EIC, EOC and IC (<cadmium/modeling/dynamic\_model\_translator.hpp>)
* Build coupled models (<cadmium/modeling/dynamic\_coupled.hpp>)
* Use the Runner <cadmium/engine/pdevs\_dynamic\_runner.hpp>
* Define the loggers we are using (state, message, debug, etc.) <cadmium/logger/common\_loggers.hpp>

We then include the header of the **Time** class we are using, in this case <NDTime.hpp>. NDTime is a C++ class that implements time operations and allows defining the time as in digital clock format (“hh:mm:ss:mss”) or as a list of integer elements ({ hh, mm, ss, mss}).

As we already mentioned, in Cadmium, we can transmit messages containing built-in C++ types (integer, float, string, double, bool, etc.) or we can define our own types. In this case, we need to transmit our own message, which is defined as a structure. Therefore, we include the path to the hpp file where such structure is defined ("../data\_structures/message.hpp"). If we need to define more than one message type, we need to include all the ones used in the model.

The content of message.hpp is the one explained in the previous section (i.e. it contains the definition of the message structure Message\_t).

We then need to include the headers of all the atomic models we are using as components of our coupled model. In this case, "../atomics/subnet.hpp", where the Subnet atomic model class is defined. We will use it to create the instance subnet1. We also include <cadmium/basic\_model/pdevs/iestream.hpp>**,** where the template classiestream\_input is defined. We will instantiate this general class to create the instance of our atomic model input\_reader.

We also need to include the headers of any C++ library needed to implement the model. In this example, we use the ioestream library to generate simulation logs in files, and String to manipulate strings.

We then declare the namespaces we are using, in this case: cadmium , cadmium::basic\_models::pdevs and std (otherwise, every time we use a method/service from the standard C++ library (std), we have to write std::; the same for cadmium:: and cadmium::basic\_models::pdevs). Then, we define that the template parameter TIME is instantiated with the type NDTime.

Cadmium provides different methods and data types to create instances of atomic models, define, and create instances of coupled models. It also uses one advanced C++ data type, shared\_ptr<>, and one advanced C++ method, make\_shared<>(), both of them defined in the standard library. shared\_ptr<> is a smart pointer that allows shared ownership of an object through a pointer. make\_shared<>() is a method that allows creating a shared\_ptr<>. It uses as a template parameter the data type that will be stored in the pointer, and as function parameters the constructor parameters for the data type. We will show a few examples later.

The data types and methods defined in Cadmium are as follows:

* **out\_port** is a structure used to declare the output ports of a model. It is the same structure we used to declare the output ports of an atomic model. Each port is defined as a structure that inherits from the template structure **out\_port** specifying the type of message handled by the port. It is defined in <cadmium/modeling/ports.hpp>.
* **in\_port** is a templated structure similar to **out\_port**, but for input ports.
* **model** is an empty class defined in <cadmium/modeling/dynamic\_model.hpp> under the namespace dynamic::modeling. It allows pointer-based polymorphism between classes derived from atomic and coupled models. This means, that it is an abstract class that encapsulates both atomic and coupled models in such a way that they can be elements in a vector of models.
* **make\_dynamic\_atomic\_model<>()** is a template method defined in <cadmium/modeling/dynamic\_model\_translator.hpp>. It is used to create an instance of an atomic model. It takes the class type of the atomic model, TIME (because all atomic models are templated classes that need to be instantiated with a TIME data type), and all the types of the parameters for the model constructor. The parameters of the method are the name of the atomic model (a string) and the parameters we need to pass to the constructor. If a parameter in the constructor is a pointer, we need to use the C++ method move() to pass the pointer to the constructor.
* **Ports** is a data type used to defined input and output ports. It is defined in <cadmium/modeling/dynamic\_model.hpp> under the namespace dynamic::modeling. It is a vector that takes as elements the typeid of the port structure declaration. To provide the type of a port, we use the method typeid() defined in the std C++ library (typeid() takes a data type as input).
* **Models** is a data type used to define the components of a coupled model. It is defined in <cadmium/modeling/dynamic\_model.hpp> under the namespace dynamic::modeling. It is a vector that takes as elements pointers to models (shared\_ptr<dynamic::modeling::model>>)
* **EICs** is a data type used to define the set of external input couplings. The set is stored as a vector with elements of type **EIC,** which is another data type to define each external input. It is implemented as a structure with two elements: the name of the submodel connected to the external input (implemented as a string), and a link that represents the external input (implemented as a shared\_ptr<>). Both **EICs** and **EIC** are defined in <cadmium/modeling/dynamic\_model.hpp> under the namespace dynamic::modeling.
* **make\_EIC<>()** is used to create an EIC structure. It is defined in <cadmium/modeling/dynamic\_model\_translator.hpp>, and it returns an element of type EIC. It takes template parameters of the types of the input ports of the coupled model and the submodel inside the coupled model, in this specific order (i.e. from – to). It uses a parameter that is a string with the name of the submodel.
* **EOCs** is a data type similar to **EICs** above, but for the External Output Couplings.
* **make\_EOC<>()** is a method similar to **make\_EIC<>()** above, but for the External Output Couplings. It returns an element of type EOC, using the types of the output port of the submodel in the coupled model, and the output port of the coupled model, in this specific order (i.e. form – to). The parameter of the method is a string with the name of the submodel.
* **ICs** is a data type to define internal couplings. It is stored as a vector that takes elements of type **IC,** used to define each internal connection. It is implemented as a structure with three elements: (1) the name of the “from” component (i.e. a string), (2) the name of the “to” component (i.e. a string), and (3) a link to connect the output port of one component with the input port of the other component. They are defined in <cadmium/modeling/dynamic\_model.hpp> under the namespace dynamic::modeling.
* **make\_IC<>()** is used to create the internal couplings, i.e., elements of type IC. It is defined in <cadmium/modeling/dynamic\_model\_translator.hpp>. It uses the type of the output port of the submodel “from” and the type of the input port of the submodel “to”, in this specific order (i.e. from output port– to input port). The parameters of the method are two strings, the first one with the name of the “from” submodel and the second one with the name of the “to” submodel (i.e. from submodel name – to submodel name).
* **coupled<TIME>** is a class that defines a coupled model. We use it to create coupled models’ instances. It is defined in <cadmium/modeling/dynamic\_coupled.hpp> under the namespace dynamic::modeling. The class uses seven variables: (1) a string with the model name, (2) a variable of type Models representing the subcomponents, (3) a variable of type Ports for the input ports, (4) a variable of type Ports for the output ports, (5) a variable of type EICs for external input couplings, (6) a variable of type EOCs for external output couplings and (7) a variable of type ICs for internal couplings. The constructor of this class takes all these parameters in this specific order.

Using these services, under /\*\*\*\*\* (1) \*\*\*\*\*/ we declare the input and output ports of the coupled model. In this example, we have a coupled model with two atomic components. If there are two **different** coupled models using the same port type (i.e., with the same name and message type), we declare the port type once and we use it for both models. However, the **same** coupled model cannot have two ports with the same name (i.e. in our C++ implementation, they cannot be the same type).

In our example, we only need one output port called top\_out, as seen in Figure 5. This port handles the same type of message that the output port out from the Subnet model: messages of type Message\_t.

As we can see in Figure 6, we only define one output port, as there are no inputs in the coupled model.

Under /\*\*\*\*\* (2) \*\*\*\*\*/ we instantiate the template model iestream\_input (defined in <cadmium/basic\_model/pdevs/iestream.hpp> to parse input messages included in a text file. In this case, the text file will contain messages of type Message\_t.

We define an atomic model class called InputReader\_Message\_t that inherits all the methods of iestream\_input. We instantiate iestream\_input with Message\_t and we leave the time as a template parameter (iestream\_input<Message\_t, T>). **In brief, this creates a new atomic class that can read text input files that contains messages of type Message\_t as inputs.**

We then need to override the constructors of the model to instantiate the template using Message\_t as a parameter. In this case, we define the default constructor (marking with the keyword default). We define a second constructor that takes the path to the text file where the model inputs are defined (InputReader\_Message\_t (const char\* file\_path)). We use a const parameter because the input parameter file\_path cannot be modified inside the constructor. The definition inherits from the atomic class; we need to instantiate the parameter that represents the type of message (iestream\_input<Message\_t, T> (file\_path) {}). **In summary, this definition instantiates the class constructors for the new atomic class we created.**

Under /\*\*\*\*\* (3) \*\*\*\*\*/ we define the main function. We create atomic and coupled models’ instances, loggers and we finally call the simulator runner to start the simulation cycle.

In this example, we first have a hardcoded path to the input file and save it in i\_input\_data (a pointer to a string).

We then **create an instance of InputReader\_Message\_t** (i.e. input\_reader). We define a variable of type shared\_ptr<dynamic::modeling::model> to store a pointer to the instance, in this case, input\_reader. As discussed earlier, we use make\_dynamic\_atomic\_model<>()to create the instance. In this case, the method uses (1) InputReader\_Message\_t**,**(2) TIME and (3) const char\* as template parameters. The method parameters are “input\_reader” and move(i\_input\_data). **We will use this instance as the atomic model inside our coupled model.**

Under /\*\*\*\*\* (4) \*\*\*\*\*/ we **create an instance of Subnet** (i.e. subnet1). We define a variable of type shared\_ptr<dynamic::modeling::model> to store a pointer to the instance subnet1. Then, make\_dynamic\_atomic\_model<>() creates the instance. It uses the class type of the atomic model (Subnet), (2) TIME and (3) the parameters in the model constructor (in this case, there are no parameters). The parameter string of the method is, in this case, "subnet1", and the constructor takes no parameters. **In brief, this declaration creates an instance of the atomic Subnet. We will use this instance as the atomic model inside our coupled model.**

Under /\*\*\*\*\* (5) \*\*\*\*\*/ **we define the top-level coupled model**. In this particular case, the top coupled model uses two atomic components. We first must define the input ports, the output ports, the components, the external input couplings (EICs), the external output couplings (EOCs) and the internal couplings (ICs).

**Input ports:** We first create a variable of type Ports to store the input ports (in this case, iports\_TOP). Because our top model has no input ports, we define the variable iports\_TOP as an empty vector ({}).

**Output ports:** We then create a variable of type Ports to store the output ports (in this case, oports\_TOP). Our top model has one output port: top\_out. We already declared it under /\*\*\*\* (1) \*\*\*\*/. Now, we need to assign it to our top model. Therefore, we define a vector with one element: the type of the output port ({typeid(top\_out)}).

**Submodels:** We then create a variable of type Models to store the components of the coupled model (in this case, submodels\_TOP). It contains the instances of submodels inside the coupled model. In this case, subnet1 and input\_reader. It does not matter the order we use to specify the components of the top model.

**External Input Couplings (EICs):** We then create a variable of type EICs to store the external input couplings (in this case, eics\_TOP). In our coupled model, we do not have EICs, therefore, we assign an empty vector to the variable eics\_TOP ({}).

**External Output Couplings (EOCs):** To define the external output couplings, we create a variable of type EOCs (in this case, eocs\_TOP). In our coupled model, we just have one external coupling connecting the atomic model subnet1 to the output port top\_out. The external coupling is defined with the simulator method make\_EOC<>() instantiated with the names of the output ports as template parameters (in this case, Subnet\_defs::out, top\_out) and the name of the submodel as the parameter of the method (in this case, “subnet1”).

**Internal Couplings (ICs):** To define the internal couplings, we create a variable of type ICs (in this case, ics\_TOP). Our coupled model just has one internal coupling connecting the output port of the atomic model input\_reader to the input port of the atomic model subnet1. The internal coupling is defined with the simulator method make\_IC<>() instantiated with the names of the output and input ports as template parameters (in this case, iestream\_input\_defs<Message\_t>::out, Subnet\_defs::in) and the name of the submodel as the parameter of the method (in this case, “input\_reader” and “subnet1”).

Once all the components of the coupled model are defined, we can create the instance of the coupled model. We first declare the variable where the coupled model will be stored, in this case, TOP. TOP is a variable of the data type shared\_ptr<dynamic::modeling::coupled<TIME>> defined in the simulator. We create the instance our top model using the C++ method make\_shared<>(). The parameters of the method are the name of the coupled model (i.e. “TOP”), and all the components we have defined in the following order: submodels\_TOP, iports\_TOP, oports\_TOP, eics\_TOP, eocs\_TOP, ics\_TOP

Once we define all the coupled models and the top model (in this case, we just have the top model), we need to define the loggers for the simulation.

To run a test, we need to define the inputs for the top model. These inputs are stored in a text file (called subnet\_input\_test.txt) that the model input\_reader will parse and use to generate messages. Each line of the file is an external input, coded as follows: an event time, a packet number, and the alternating bit. We need to specify the packet before the bit, exactly as defined by the >> operator we discussed earlier in Message\_t.

If we look at the input file (Figure 7), we can see, for example, that at time 10s, we are generating a packet with id 1 and alternate bit 1; at time 20s, we are generating a packet with id 2 and alternate bit 0; etc. It is the responsibility of the modeler to define the input file properly.

00:00:10 1 1

00:00:20 2 0

...

00:02:30 15 1

00:02:40 16 0

...

00:03:20 20 0

Figure 7. Test input file (subnet\_input\_test.txt)

Figure 8 shows a message log of the simulation for the subnet test coupled model we discussed earlier using the input file in Figure 7. The log includes the global simulation time followed by the messages generated by each atomic model on each port at that simulation time.

...

00:00:10:000

[iestream\_input\_defs<Message\_t>::out: {1 1}] generated by model input\_reader

00:00:13:000

[Subnet\_defs::out: {1 1}] generated by model subnet1

...

00:02:30:000

[iestream\_input\_defs<Message\_t>::out: {15 1}] generated by model input\_reader

00:02:40:000

[iestream\_input\_defs<Message\_t>::out: {16 0}] generated by model input\_reader

00:02:43:000

[Subnet\_defs::out: {16 0}] generated by model subnet1

...

00:03:20:000

[iestream\_input\_defs<Message\_t>::out: {20 0}] generated by model input\_reader

00:03:23:000

[Subnet\_defs::out: {20 0}] generated by model subnet1

Figure 8. Message log of the simulation for the subnet test coupled model

When the simulation starts, the atomic models are initialized. The input\_reader model is initialized in a state with time advance zero, so it can start by reading the input event file. Similarly, if we recall our definition of subnet, we can see that it was initialized in a passive state.

The log shows all the message bags generated by the atomic models every time the simulator collects the outputs.

At time 10s, input\_reader generates a message with value {1 1}. This message is the first input event retrieved from the input file subnet\_input\_test.txt. If we recall the operator << we defined for the structure Message\_t, the message we get has the format {packet bit}.

At time 13s, the subnet1 generates the message {1 1}. The message has the same meaning as before. If we recall the subnet1 implementation, the model resends the message received with a 95% probability to simulate failures in a network. In this case, there was no failure.

This pattern is repeated through the simulation every time there is an event on the input file. However, as we mentioned, the subnet model has a 5% probability of not transmitting a packet. This is what happened at time 2min 30s. In the log, we can see that input\_reader generates an output message that is not transmitted by subnet1. At time 2min 30s, input\_reader generates the message {15 1} and the subnet1 does not generate any message at time 2min 33s.

The simulation process continues until the simulation finishes at time 3min23s. At that time, there are no more events in the input files and both atomic models are passivated.

We can also generate a log of the state of each atomic model (Figure 9) (we will explain later on how we define and change the logs). The log of the state is generated base on the operator << we defined for each atomic model class. The state log generates the global time when a state on the top model changes, and the states of **all** the atomic models at that time. For the atomic model input\_reader, the state is the time of the next internal event. For example, at time 10s, the state is “next time: 00:00:00:000” The state log for the atomic model subnet1 is the index (i.e. the number of packets the network has received so far) and if the model is transmitting a message (i.e. 1) or not (i.e. 0). For example, at time 10s, the state is “index: 1 & transmitting: 1”.

...

00:00:10:000

State for model input\_reader is next time: 00:00:10:000

State for model subnet1 is index: 1 & transmitting: 1

00:00:13:000

State for model input\_reader is next time: 00:00:10:000

State for model subnet1 is index: 1 & transmitting: 0

...

00:02:30:000

State for model input\_reader is next time: 00:00:10:000

State for model subnet1 is index: 15 & transmitting: 0

00:02:40:000

State for model input\_reader is next time: 00:00:10:000

State for model subnet1 is index: 16 & transmitting: 1

00:02:43:000

State for model input\_reader is next time: 00:00:10:000

State for model subnet1 is index: 16 & transmitting: 0

...

00:03:20:000

State for model input\_reader is next time: inf

State for model subnet1 is index: 20 & transmitting: 1

00:03:23:000

State for model input\_reader is next time: inf

State for model subnet1 is index: 20 & transmitting: 0

Figure 9. Log of the state of each atomic model

Looking in more detail, at time 10s, input\_reader generated the message {1 1}. After executing the internal transition, the next event is in 10s, which is the state of the model. At time 10s, subnet1 executed the external transition with the input message {1 1}. After the external transition, the state of the model is as follows: (1) the number of packets the network has received so far is 1 (index: 1) and (2) the network has something to transmit (transmitting: 1). Once the elapsed time of the atomic model subnet1 is over (in this case 3s), the internal transition is triggered in the subnet1 atomic model. As we can see in the message log, the message {1 1}is transmitted and the model state changes from transmitting equal true (i.e. 1) to transmitting equal false (i.e. 0). As we can see, the state of the input\_reader atomic model does not change because it was not imminent. This pattern is repeated through the whole simulation every time there is an event on the input file.

We need to notice, that when a packet is lost (e.g. time 2min30sec), the state variable index increases because the network has received a new packet. However, the state variable transmitting is set to false because that packet will not be transmitted to the output of the model.

We define the loggers under /\*\*\*\*\* (6) \*\*\*\*\*/ in our cpp file above (Figure 6).

First, we need to define the file where we will output the message log. To do so, we create a variable (out\_messages) of type ofstream. We initialize out\_messages with the path to the output file for the message log ("../simulation\_results/subnet\_test\_output\_messages.txt").

We then define the structure oss\_sink\_messages to tell the simulator where we will save the output log. The structure uses a method (sink) that returns a pointer to out\_messages. We use oss\_sink\_messages to declare the message logger.

We need to do the same for the state variable log. To do so, we define a variable (out\_state) of type ofstream. We initialize out\_state with the path to the output file for the state log (in this case, "../simulation\_results/subnet\_test\_output\_state.txt").

Finally, we define the structure (oss\_sink\_state) to tell the simulator where to save the state log. The structure has a method (sink) that returns a pointer to out\_state. We will use oss\_sink\_state to declare the state logger.

To define the logger, we need to include the following declarations:

**using state = logger::logger<logger::logger\_state, dynamic::logger::formatter<TIME>, oss\_sink\_state>;**

It defines the state logger. We instantiate the logger with: (1) the logger we are using, in this case **logger\_state** (defined in <cadmium/logger/logger.hpp>), (2) the formatter (defined in <cadmium/logger/ dynamic\_common\_loggers.hpp>), and (3) the sink we just defined (i.e. oss\_sink\_state).

All logs are defined in the same way. Only the first and third template parameters changes because they are the ones that specify which log we are using and where we generate the log.

**using log\_messages = logger::logger<logger::logger\_messages, dynamic::logger::formatter<TIME>, oss\_sink\_messages>;**

It defines the message logger. As in the previous case, we instantiate (1) the logger we are using, in this case, **logger\_messages** (defined in <cadmium/logger/logger.hpp>), (2) the formatter (defined in <cadmium/logger/ dynamic\_common\_loggers.hpp>), and (3) the sink just defined (oss\_sink\_messages).

In order to include the global time of the simulation inside the state and message log, we need to declare a new logger: global\_time. In this specific case, we need two: one for the messages and one for the states because the logs are generated on different files.

**using global\_time\_mes = logger::logger<logger::logger\_global\_time, dynamic::logger::formatter<TIME>, oss\_sink\_messages>;**

It defines the global time for the message logger. As in the previous case, we instantiate with (1) the logger we are using, in this case **logger\_global\_time** (defined in <cadmium/logger/logger.hpp>), (2) the formatter (defined in <cadmium/logger/ dynamic\_common\_loggers.hpp>), and (3) the sink (oss\_sink\_messages).

**using global\_time\_sta = logger::logger<logger::logger\_global\_time, dynamic::logger::formatter<TIME>, oss\_sink\_state>;**

It defines the global time for the state logger as in the previous cases.

Once we have declared all the loggers we need, we have to combine them, so our simulation generates all the logs at the same time. For this purpose, we use the **multilogger** structure defined in <cadmium/logger/logger.hpp> instantiated with the above log definitions (i.e. **state, log\_messages, global\_time\_mes, global\_time\_sta)** as template parameters:

**using logger\_top = logger::multilogger<state, log\_messages, global\_time\_mes, global\_time\_sta>;**

After defining the loggers, we need to call the runner to be able to execute the simulation (Figure 7 /\*\*\*\*\* (7) \*\*\*\*\*/).

We first create an instance of the runner for our top model, in this case, r. It is an instance of the runner class defined in <cadmium/engine/ pdevs\_dynamic\_runner.hpp> under the namespace dynamic::engine::

The runner class takes two parameters: the class used for the time (in this case, NDTime) and a logger (in this case, logger\_top). The parameters for the class constructor are the name of the top model (TOP in our case) and the initial time for the simulation (0 in this case).

Then, we define the end time of the simulation. We have two options: (1) run the simulation until a specific time or (2) run the simulation until all models are passivated.

To run the simulation until a specific time we use the runner method run\_until(). This method takes as parameter the end time of the simulation.

To run the simulation until all models are passivated, we use the runner method run\_until\_passivate(). This method does not take any parameter.

In our example, we run the simulation until the time is 4h.

# A Summary on Port Definition

When we define a DEVS model, we assign them set of input and output ports. Each port can be defined with a name and a set of values that it can carry. In the example above (Figure 10), we define an input port called “in” and an output port called “out”. Both of them use the same types, and in Cadmium, this is represented as messages of type Message\_t, which represents a bag of values.

Subnet

out <Message\_t>

in<Message\_t>

Figure 10. Atomic model Subnet with the ports and the message type on each port

To define this port in Cadmium, we must do the following:

1. **DECLARATION OF PORTS** (this was done in /\*\*\*\*\* (1) \*\*\*\*\*/ in Figure 2). In the declaration, we inform the simulator which ports we are defining. In our example, we need two ports that are associated to the Subnet model. The first one is called “in”, and it receives input messages of type Message\_t (struct in : public **in\_port**<Message\_t>{}). Here, **in\_port<>** is a templated structure (struct) defined in the simulator, which is used to define input ports with templates. It is mandatory that each input port (in this case, in) is defined as a structure that inherits from **in\_port<>** and uses a given type of message (in this case Message\_t). The output port called “out” is defined in a similar fashion, but using **out\_port<>**, a templated structure (struct) defined in the simulator to define output ports. In summary, we declare a new data type for each port in the atomic model, and they are declared as **in\_port (or out\_port)** that can only receive a Message\_t. Therefore, they can only be used within the Subnet model (and we can have other types called using the same name in other atomic submodels). We name the data type with the name of the port (in this case in and out) and they inherit from **in\_port<>** and **out\_port<>** based on the type of port.
2. **ASSIGNMENT OF PORTS TO THE ATOMIC MODEL:** (this was done in /\*\*\*\*\* (2) \*\*\*\*\*/ in Figure 2). Once we have declared the ports types we are using (in this case, in and out), we need to assign them to the model that will use them, in this case, the Subnet atomic model. We assign the input ports with using **input\_ports**=tuple<typename Subnet\_defs::in>. We **must** define the data type **input\_ports**, which is a tuple of the input ports declared in /\*\*\*\*(1)\*\*\*\*/. In this specific example, we only have one input port named in. If we needed, for instance, two input ports, the tuple will need to define the names of the two ports as elements (e.g. using **input\_ports**=tuple<typename Subnet\_defs::in1, typename Subnet\_defs:: in2>), which should have been previously declared in \*\*\*(1)\*\*\*. The input ports **must** be assigned under the name **input\_ports** because this is a mandatory simulation service (used to check port types; the simulator generates compilation errors if a data type assigned to the **input\_ports** tuple does not inherit from **in\_port<>** or if the data types inside the tuple have duplicated names). Output ports are assigned similarly.

The input parameter **mbs** in the external transition is a bag that contains the input messages classified by port. **make\_message\_bags**<input\_ports>:: type **mbs** takes the tuple **input\_ports** we defined earlier, and it generates the **mbs** tuple, whose elements are vectors of messages. **mbs** has the same number of elements as **input\_ports** (here, it is a tuple of one element: a vector of Message\_t a bag in port in; if we needed to use two ports, e.g. in1 and in2, **make\_message\_bags**<input\_ports>::typewould define **mbs** as a tuple with two elements: the first, a vector representing the bag of messages in port in1; the second, a vector representing the bag of messages in port in2). To retrieve the bag of messages in a specific port of **mbs**, we use **get\_messages<>**, which takes the port name Subnet\_defs::in. The bag retrieved is a vector, which we store in an auxiliary variable (bag\_port\_in). The bag of messages in for a specific port is a vector, so, to access the first element of the bag stored in bag\_port\_in we use bag\_port\_in[0].

Similarly, in the output function we return a bag of messages in the output ports (i.e. **bags**). **make\_message\_bags**<output\_ports>::type takes the tuple **output\_ports** and it generates the tuple **bags**; whose elements are vectors of messages. We use an auxiliary variable to generate each message bag (in this example, we use bag\_port\_out). To place a bag in **bags**, we use the method **get\_messages<>**, which takes the port name as template parameter (in this case, Subnet\_defs::out).

The declaration of ports for coupled models is similar to the one for atomic models. For our example (Figure 5), the coupled model only has one output port named top\_out (Figure 6 /\*\*\*\*\* (1) \*\*\*\*\*/) that handles the same type of message that the output port out from the Subnet model (i.e. messages of type Message\_t). As in the case of atomic models, the port is declared as a struct named top\_out that inherits from **out\_port<>** and uses a given type of message (in this case Message\_t).

In coupled models, we assign input and out ports differently. We use a data type named **dynamic::modeling::Ports**, a vector of ports as defined in the previous paragraph (Figure 6 /\*\*\*\*\* (5) \*\*\*\*\*/). We need to define two variables, one for input ports and one for output ports. In this specific example, the input ports variable (i.e. iports\_TOP) is an empty vector because the coupled model has no ports. The output ports (i.e. oports\_TOP) is a vector with one element: the top\_out port declared above. If we need, for instance, two output ports, we need to define a vector with the names of the two ports as elements (e.g. oports\_TOP = { typeid(top\_out1), typeid(top\_out1)}, which should have been previously declared.

The names of the data types that declare the ports of both the atomic models and coupled models are used as template parameters in the methods that the simulator provides to define the EOC, IC, and EIC. In this specific example, we will need to use the ports from the subnet atomic model (Subnet\_defs::in and Subnet\_defs::out), the ports from the Input reader atomic model (iestream\_input\_defs<Message\_t>::out) and the ports of the top model (top\_out).

**PORTS FOR EICs**: EICs are defined using **dynamic::translate::make\_EIC**<>(). The template parameters of the method are: (1) the name of the data type of the input port of the coupled model and (2) the name of the data type of the port from the submodel inside the coupled model, in this specific order (i.e. form – to). In our example, the coupled does not have EICs, therefore we do not use this method.

**PORTS FOR EOCs**: EOCs are defined using **dynamic::translate::make\_EOC**<>(). The parameters are (1) the name of the data type of the port from the submodel (Subnet\_defs::out) and (2) the name of the data type of the output port of the coupled model (top\_out), in this specific order (i.e. form – to). In our example, we connect the out port from Subnet1 with top\_out of the coupled model.

**PORTS FOR ICs**: ICs are defined using **dynamic::translate::make\_IC**<>(). The parameters are (1) the name of the data type of the output port of the submodel “from” (iestream\_input\_defs<Message\_t>::out) and (2) the name of the data type of the input port of the submodel “to” (Subnet\_defs::in), in this specific order (i.e. from output port– to input port). In our example, we are connecting the out port from input\_reader with in port of Subnet1.

# Defining the make file to compile the test

The model we have defined along with the simulator is a regular C++ program. Here, we will explain how to compile the program with a make file.

We first need to create a file called “makefile”. The file will have the statements defined in Figure 11:

CC=g++

CFLAGS=-std=c++17

INCLUDECADMIUM=-I ../../cadmium/include

INCLUDEDESTIMES=-I ../../DESTimes/include

**#CREATE BIN AND BUILD FOLDERS TO SAVE THE COMPILED FILES DURING RUNTIME**

bin\_folder := $(shell mkdir -p bin)

build\_folder := $(shell mkdir -p build)

results\_folder := $(shell mkdir -p simulation\_results)

**#TARGET TO COMPILE SUBNET TEST**

message.o: data\_structures/message.cpp

$(CC) -g -c $(CFLAGS) $(INCLUDECADMIUM) $(INCLUDEDESTIMES) data\_structures/message.cpp -o build/message.o

main\_subnet\_test.o: test/main\_subnet\_test.cpp

$(CC) -g -c $(CFLAGS) $(INCLUDECADMIUM) $(INCLUDEDESTIMES) test/main\_subnet\_test.cpp -o build/main\_subnet\_test.o

tests: main\_subnet\_test.o message.o

$(CC) -g -o bin/SUBNET\_TEST build/main\_subnet\_test.o build/message.o

**#TARGET TO COMPILE EVERYTHING**

all: tests

**#CLEAN COMMANDS**

clean:

rm -f bin/\* build/\*

Figure 11. Make file to compile the subnet test

First, we define the compiler we are using, in this case, g++.

Then we need to define the C++ standard we are using, in this case, C++17.

We also need to define the paths to Cadmium and DESTimes libraries, so the compiler can find the files we specified in the #includes <>. We define the paths in the INCLUDECADMIUM and INCLUDEDESTIMES variables. In a makefile, a path is preceded by -I. The paths are relative from the location of the make file. If we download the simulator as explained at the beginning of the manual and we create new models inside the folder “DEVS-Models” following the same structure as in the ABP, we will not need to modify these paths.

We store intermediate built files in a folder called build; the executables in a folder called bin and the simulation results in a folder called simulation\_results. To do so, we need to be sure that these folders exist, and if they do not exist, we need to create them. We can do this in the makefile as follows:

**command\_name := $(shell mkdir -p folder\_name)**

We need to assign a name to the make file command, in this generic case **command\_name**, we then write a shell command to create the directory if it does not exist. **mkdir** creates a directorywith the name **folder\_name**. The **-p** option specifies create the directory only if it does not already exist. In our case, we create the folders: **build**, **bin** and **simulation\_results**.

We then specify how to create the executable to run the subnet test. To create the executable, we first need to create the object files (.o) of all the cpp files involved in our program, in this case, message.cpp and main\_subnet\_test.cpp. The object files contain the compiled code.

To generate an object file in the context of our simulator, we need to use the following statements:

file\_name.o: relative\_path\_to\_cpp\_file

$(CC) -g -c $(CFLAGS) $(INCLUDECADMIUM) $(INCLUDEDESTIMES) relative\_path\_to\_cpp\_file -o build/file\_name.o

Where file\_name is the name we give to the object file (we usually use the same one we gave to the cpp file) and relative\_path\_to\_cpp\_file is the relative path to the cpp file from where the make file is located. $() is used to include the variables we defined at the beginning of the make file. **-g** is used to include debugging information, -c is an instruction to the preprocessor to keep comments, and **-o** is used to specify the name of the output file.

For the subnet test, we need to create the object files of message.cpp and main\_subnet\_test.cpp.

Once we have the object files, we need to link them together to create the executable. We need to use the following line of code:

tests: main\_subnet\_test.o message.o

$(CC) -g -o bin/SUBNET\_TEST build/main\_subnet\_test.o build/message.o

tests is the name we give to the make file command that performs the linking. We then need to specify all the other make file commands we need to execute before this one. In this case, main\_subnet\_test.o and message.o to generate the build objects. We then tell the compiler to perform the linkage of the .o files to generate the executable. We give the name SUBNET\_TEST to our executable.

To be able to use the command “make all” to compile, we need to define what all means. In this case, all means execute the command tests.

We all need to define a “clean” command that deletes all the object and executable files in the bin and build folders before compiling (i.e. rm -f bin/\* build/\*).

Once the make file is ready, to compile the test we open the bash terminal inside the folder ABP. To compile the project, type: make clean; make all.

To run the test, open a bash terminal inside the subfolder bin and type the command: ./SUBNET\_TEST. The simulation results will be in the folder simulation\_results.

# Simulating the complete ABP model

Figure 2 presented the structure of the ABP model coupled model discussing throughout this document. The Alternating Bit communication protocol tries to provide reliable transmission on an unreliable network. The ABP model consists of 3 components: A sender, which transmits messages; a network, and a receiver, which receives the messages transmitted by the sender and returns acknowledgement messages (positive or negative). The network is decomposed further to two subnets corresponding to the sending and receiving channels, respectively. The sender and the receiver communicate with each other through the network component.

As we already mentioned, the Subnet atomic model uses one input port and one output port, and the model passes the data it receives after a time delay of 3 seconds. To model the unreliability of the network, only approximately 95% of the data will be transferred (i.e. 5% of the data will be lost through the subnet).

The behavior of the receiver is to receive the data and send back an acknowledgment extracted from the received data after a time period. The implementation of the receiver atomic class is available in Appendix B.

The sender sends data packets and waits for an acknowledgment. If the acknowledgment is not received after a period of time, it sends the same packet again. If the acknowledgment is received, the sender sends the next packet. The implementation of the sender atomic class is available in Appendix C.

The implementation of the ABP coupled model is available in Appendix D.

The full logs of the simulation are available in Appendix E (message log) and Appendix F (state log). Here we explain the most relevant aspects of the logs.

The input data we use for our simulation is as follows: at time 10s, we tell the sender that it will need to send a message that is 5 packets long and at time 15min, we tell it to send a message that is 3 packets long.

00:00:10 5

00:15:00 3

In the next snippet, we can see the message generated when the sender transmits a packet until it receives the confirmation that the packet was received and starts sending a new packet.

At time 10s, we generate a message (coming from the input file) that tells the sender that it will need to send a message that is 5 packets long. The message is generated by the model input reader (i.e. the one in charge of transforming the input files in DEVS messages).

At time 20s, the sender sends the first packet with the alternate bit ({1 0}) through the port dataOut and the packet number (i.e. 1) through the packetSentOut port (the output of the top model). The port dataOut is connected to the subnet model. After a 3s delay, the subnet transmits the packet with the alternate bit i.e. it generates the message {1 0} in the port out. The out port of the subnet is connected to the receiver. Once the receiver receives the packet, after a 10s delay, it sends an acknowledgment ({0 0}). For the acknowledgment, the second element represents the alternate bit. The acknowledgment is transmitted through the network (i.e. at time 36s, subnet2 generates {0 0} on its out port). The out port of subnet2 is connected to the sender. As soon as the sender receives the acknowledgement (i.e. time advance 0), it generates a message in the ackReceivedOut. The message is the alternate bit (i.e. 0). After a 10s delay, i.e. at time 46s, the process starts again with the second packet.

...

00:00:10:000

[iestream\_input\_defs<int>::out: {5}] generated by model input\_reader

00:00:20:000

[Sender\_defs::packetSentOut: {1}, Sender\_defs::ackReceivedOut: {}, Sender\_defs::dataOut: {1 0}] generated by model sender1

00:00:23:000

[Subnet\_defs::out: {1 0}] generated by model subnet1

00:00:33:000

[Receiver\_defs::out: {0 0}] generated by model receiver1

00:00:36:000

[Subnet\_defs::out: {0 0}] generated by model subnet2

00:00:36:000

[Sender\_defs::packetSentOut: {}, Sender\_defs::ackReceivedOut: {0}, Sender\_defs::dataOut: {}] generated by model sender1

00:00:46:000

[Sender\_defs::packetSentOut: {2}, Sender\_defs::ackReceivedOut: {}, Sender\_defs::dataOut: {2 1}] generated by model sender1

…

In the next snippet, we can see some states of the atomic models. For example, at time 10s, after the external transition of the sender1 atomic model is executed, the state of the model is packetNum: 1 & totalPacketNum: 5 (i.e. the model is sending the first packet and it has to send 5 packets in total). At time 23s, once the subnet model has transmitted the packet, the state of subnet1 is index: 1 & transmitting: 0 (i.e. the subnet has received one packet so far and it does not need to transmit anything). After the receiver sends the acknowledgement (i.e. at time 33s), the state of the receiver is ackNum: 0 (i.e. the last alternate bit received is 0) and the state of subnet2 is index: 1 & transmitting: 1 (i.e. it has received a packet so far and it has something to transmit). At time 36s, once the sender receives the acknowledgment, it updates its state to packetNum: 2 & totalPacketNum: 5 (i.e. the next packet it has to send is 2 and the total number is 5, which means the full message is not sent yet).

...

00:00:10:000

State for model sender1 is packetNum: 1 & totalPacketNum: 5

...

00:00:23:000

State for model subnet1 is index: 1 & transmitting: 0

...

00:00:33:000

State for model receiver1 is ackNum: 0

State for model subnet2 is index: 1 & transmitting: 1

...

00:00:36:000

State for model sender1 is packetNum: 2 & totalPacketNum: 5

...

# Defining the make file to compile all the test and the ABP

As per good programming practices, a project should have a single makefile. Therefore, we modify the makefile we already create to include the generation of the executable for the unit tests of the receiver and subnet and the ABP simulator.

We need to generate an object file as we did for message.cpp and main\_subnet\_test.cpp for the following files: main\_sender\_test.cpp, main\_receiver\_test.cpp and main.cpp.

Once we have all the object files, we need to generate the executables. We generate the executables for the tests under the make command tests. To generate the ABP simulator executable, we create a new command, simulator, and we write the instructions to link the necessary object files.

Finally, we add simulator to the all command.

CC=g++

CFLAGS=-std=c++17

INCLUDECADMIUM=-I ../../cadmium/include

INCLUDEDESTIMES=-I ../../DESTimes/include

**#CREATE BIN AND BUILD FOLDERS TO SAVE THE COMPILED FILES DURING RUNTIME**

bin\_folder := $(shell mkdir -p bin)

build\_folder := $(shell mkdir -p build)

results\_folder := $(shell mkdir -p simulation\_results)

**#TARGET TO COMPILE ALL TESTS**

message.o: data\_structures/message.cpp

$(CC) -g -c $(CFLAGS) $(INCLUDECADMIUM) $(INCLUDEDESTIMES) data\_structures/message.cpp -o build/message.o

main\_subnet\_test.o: test/main\_subnet\_test.cpp

$(CC) -g -c $(CFLAGS) $(INCLUDECADMIUM) $(INCLUDEDESTIMES) test/main\_subnet\_test.cpp -o build/main\_subnet\_test.o

main\_sender\_test.o: test/main\_sender\_test.cpp

$(CC) -g -c $(CFLAGS) $(INCLUDECADMIUM) $(INCLUDEDESTIMES) test/main\_sender\_test.cpp -o build/main\_sender\_test.o

main\_receiver\_test.o: test/main\_receiver\_test.cpp

$(CC) -g -c $(CFLAGS) $(INCLUDECADMIUM) $(INCLUDEDESTIMES) test/main\_receiver\_test.cpp -o build/main\_receiver\_test.o

tests: main\_subnet\_test.o main\_sender\_test.o main\_receiver\_test.o message.o

$(CC) -g -o bin/SUBNET\_TEST build/main\_subnet\_test.o build/message.o

$(CC) -g -o bin/SENDER\_TEST build/main\_sender\_test.o build/message.o

$(CC) -g -o bin/RECEIVER\_TEST build/main\_receiver\_test.o build/message.o

**#TARGET TO COMPILE ONLY ABP SIMULATOR**

main\_top.o: top\_model/main.cpp

$(CC) -g -c $(CFLAGS) $(INCLUDECADMIUM) $(INCLUDEDESTIMES) top\_model/main.cpp -o build/main\_top.o

simulator: main\_top.o message.o

$(CC) -g -o bin/ABP build/main\_top.o build/message.o

**#TARGET TO COMPILE EVERYTHING (ABP SIMULATOR + TESTS TOGETHER)**

all: simulator tests

**#CLEAN COMMANDS**

clean:

rm -f bin/\* build/\*

Cadmium’s Services for Atomic Models

The atomic models are defined in an hpp file following the template provided in Appendix A.

Each atomic model implementation must include the following headers:

**#ifndef** ATOMIC\_MODEL\_NAME\_HPP

**#define** ATOMIC\_MODEL\_NAME\_HPP

//Include simulator headers

**#include <cadmium/modeling/ports.hpp>**

**#include <cadmium/modeling/message\_bag.hpp>**

//Include other headers needed for the C++ implementation of the model

#include <limits>

#include <assert.h>

//Include the relative path to message types for not built-in C++ types such as float, int, string, etc.

#include "../data\_structures/message.hpp"

**using namespace cadmium;**

**using namespace std;**

//Here goes the port declaration

//Here goes the atomic model class implementation

**#endif //**ATOMIC\_MODEL\_NAME\_HPP

First, we need to include the libraries of the simulator that provide the services to define new ports (**<cadmium/modeling/ports.hpp>)** and to handle bags of messages **(<cadmium/modeling/message\_bag.hpp>).** Then, we need to include any C++ library that we use in the model implementation. The library limits is used when we need to passivate a model (i.e. set the time advance to infinity). The rest of the libraries are optional and the ones to be included depends on the specific model implementation. Assert.h is useful to stop the simulation and generate an error if we have non-desired behavior. For example, the model definition states that the inputs to the model are integers between 0 and 9. When we implement our model, we can use a conditional statement and the methods provided in assert.h to check that the condition is satisfied. If not, the simulation is stopped and an error explaining the reason is displayed. Other libraries may be needed based on the model implementation.

In Cadmium, we can use built-in C++ types as messages (integer, float, string, double, bool, etc.) or we can define our own ones as C++ structures. In that case, we need to include the path to the hpp file where the structure is defined (e.g. #include "../data\_structures/message.hpp").

Finally, before starting with the model implementation (port and atomic model definition), we declare the namespaces we are using, in this case: cadmium and std. If we do not declare them, every time we use a method/service from the standard C++ library (std), we have to write std:: followed by the name of the service. The same occurs with cadmium (cadmium::).

# Declaring ports

We define the ports as a structure (in this general implementation, we called it model\_name\_ports\_defs) that contains all the input and output ports of the atomic/coupled model. Two ports cannot have the same name. Different ports can handle the same *message type*.

In this general implementation, each port is defined as a structure that inherits from the template structures out\_port and in\_port defined in the simulator, specifying the type of message handled by the port. In this case, we define two output ports, the first one is called out\_port\_name1 and it handles messages of type message\_type\_1; the second one is called out\_port\_name2 and it handles messages of type message\_type\_2. We also define two input ports, the first one is called in\_port\_name1 and it handles messages of type message\_type\_3; the second one is called in\_port\_name2 and it handles messages of type message\_type\_4.

//Port definition

**struct** model\_name\_ports\_defs {

**struct** out\_port\_name1 **: public out\_port<**message\_type\_1**> {};**

**struct** out\_port\_name2 **: public out\_port<**message\_type\_2**> {};**

**struct** in\_port\_name1 **: public in\_port<**message\_type\_3**> {};**

**struct** in\_port\_name2 **: public in\_port<**message\_type\_4**> {};**

**};**

# Implementing atomic models: a C++ class

Atomic models are implemented as a templated C++ class (atomic\_model\_name) in the hpp file we mentioned at the beginning of the section (//Here goes the atomic model class implementation). The template parameter of the class represents the type of time (TIME)

Each class representing an atomic model MUST contains the following variables, methods, and constructors. Everything inside the class is public as the simulator has to access the methods of the class to execute the simulation.

### Port definition

As discussed earlier, the ports were declared inside the structure model\_name\_ports\_defs. To access the input port 1, we would need to use model\_name\_ports\_defs::in\_port\_name1 and to use the output port 1 model\_name\_ports\_defs::out\_port\_name1.

The ports are assigned to the corresponding atomic model as follows:

using input\_ports=tuple<typename model\_name\_ports\_defs::in\_port\_name1 , typename model\_name\_ports\_defs::in\_port\_name2>;

using output\_ports=tuple< typename model\_name\_ports\_defs::out\_port\_name1 , typename model\_name\_ports\_defs::out\_port\_name2>;

The C++ keyword using allows us to rename a data type. Each atomic model must define their input and output ports as a data type called “input\_ports” and “output\_ports”, respectively. We need to use these specific names because the simulator will use them to check that the atomic model has all the needed components and that the ports are properly defined (e.g. there are not two input ports with the same name). Both the input and out ports are defined as a tuple (tuple<>), a C++ object that packs elements of possibly different types together in a single object. We can see it as a vector with elements of different types. Because of this, we need to specify the type of each of the elements in the tuple. The typename specifies that model\_name\_ports\_defs::out\_port\_name1, etc. are data types that will overwrite the template class in the simulator.

### Model parameters

If we want to define a parameterized model, the parameters are defined as variables inside the class. The value of these variables will be overwritten inside the constructor of the class. See Appendix C for an example.

### State definition

The state variables of the model are declared in a structure called state\_type. All the state variables of the model must be declared inside the structure, and a single state variable of type state\_type name state must be defined. We need to use these specific names because they are explicitly used by the simulator to check that the model is implemented according to the DEVS formalism. The simulator verifies if this structure (which represents the model’s state) is updated inside the output function or the time advance function (**remember that these two operations are not valid according to DEVS specifications**).

**struct state\_type {**

//Declare the state variables here

**};**

**state\_type state;**

### Class Constructor

Each class **must have at least one default constructor** (i.e. without parameters): atomic\_model\_name(). Inside the constructor, both the parameters (if we defined a parameterized model) and the state of the model are initialized. As in any C++ class, we can have more than one constructor as long as they take different parameters.

Having a constructor that takes the model parameters as inputs is useful if we want to create instances of the class with different parameters.

### Internal Transition Function

The internal transition function is defined as a void method called internal\_transition(), and it takes no parameters (because the method can access the state variable of the class).

**void internal\_transition() {**

//Define internal transition here

**}**

### External Transition Function

The external transition function is called when an external event arrives in one of the model’s output ports. It is defined as a void method called external\_transition. The method takes two parameters, the elapsed time (**e**) and a bag of input messages (**mbs**). There is one bag of messages per input port.

**void external\_transition(TIME e, typename make\_message\_bags<input\_ports>::type mbs){**

//Define external transition here

}

There are some primitives devoted to handling the messages:

* **typename make\_message\_bags<input\_ports>::type** mbs – It creates an input message bag called mbs. As we already mentioned, typename indicates that the expression that follows is a data type. make\_message\_bags<> is a template data type declared in <cadmium/modeling/message\_bag.hpp>, used to declare a bag of messages for input or output ports. We need to instantiate the template with the word input\_ports to define the input bag, using ::type. The parameter declaration, in this case, declares mbs as a tuple whose elements are the message bags on the input ports. The messages inside the set of messages in the bag are stored in a C++ vector.
* **get\_messages<typename model\_name\_ports\_defs::in\_port\_name1>(mbs)** – It gets the message bag from the input port in\_port\_name stored in the tuple mbs. The method get\_messages uses a template parameter for the port we want to access, in this case, the in\_port\_name1 port, defined by typename model\_name\_ports\_defs::in\_port\_name1. The function parameter is the bag of messages we want to access, in this case mbs. The retrieved bag is a C++ vector. The data type of the elements inside the vector is the one handled by the port.

### Confluent Transition Function

* The confluent transition function is called when an internal transition is scheduled at the same time as an external event arrives. The method to define this void function, called confluence\_transition, takes two parameters: the elapsed time (**e**) and a bag of messages (**mbs)** The default implementation for the confluent function is to execute the model’s internal transition first, and the external transition after that, with an elapsed time equal to zero.
* All the primitives useful for handling messages in the external transition can also be used here.

**void confluence\_transition(TIME e, typename make\_message\_bags<input\_ports>::type mbs){**

internal\_transition();

external\_transition(TIME(), std::move(mbs));

**}**

### Output Function

The output function is called when a model is imminent, and before calling the internal transition function (or the confluent function). It is defined as a constant method (i.e. we are not allowed to change the state of the model) that returns a bag of messages in the output ports. It does not take any parameter because the method can access the state variable of the class. It is called **output ()**

**typename make\_message\_bags<output\_ports>::type output() const {**

**typename make\_message\_bags<output\_ports>::type bags;**

//Define output function here

**return bags;**

**}**

To handle messages, we use the same primitives as in the external transition function but instantiated for the output ports instead of the input ports.

### Time Advance Function

The time advance function is defined as a constant method (i.e. we are not allowed to modify the state of the model) that returns the time of the next internal transition and takes no parameters. It is called **time\_advance.**

**TIME time\_advance() const {**

**TIME next\_internal;**

//Define time advance function here

**return next\_internal;**

**}**

There are two useful primitives to set the time advance of the model to zero and infinity

* **numeric\_limits<TIME>::infinity()** (a method in the limits library). It is used to passivate the model.
* **TIME ()** – It sets the time advance to zero.

**IMPORTANT**: **According to ST-DEVS, only the transition functions (i.e. external, internal and confluence) can be stochastic. The time advance function and the output function MUST be deterministic.**

### State the Logs

Once all the DEVS functions are defined, we specify how we want to output the state of the model in the state log. To declare how to log the state of the model, we define the << operator for the structure state\_type. The following method shows how to do this.

**friend ostringstream& operator << (ostringstream& os, const typename** atomic\_model\_name**<TIME>::state\_type& state) {**

//Define how to log the state here

**return os;**

}

The operator takes a pointer to the stream where we want to log (i.e. os) and the memory address of the state variable of the model (i.e. state)

We use a const type to ensure that it will not be modified inside the operator. It is important to notice that we need to use typename atomic\_model\_name<TIME>::state\_type to specify the type of the state. That expression is used to access the structure state\_type inside the template class atomic\_model\_name<TIME>. We need to declare the operator using the keyword friend to specify that the function can access the private members of the structure state\_type.

# Using Atomic Models: Creating Instances from the Class

To be able to use the atomic models we have defined or the ones available in the libraries, we need to create an instance. To create instances of atomic models, Cadmium provides a data type and a method:

* **model** is an empty class defined in <cadmium/modeling/dynamic\_model.hpp> under the namespace dynamic::modeling. It allows pointer-based polymorphism between classes derived from atomic and coupled models. It is an abstract class that encapsulates both atomic and coupled models in such a way that they can be elements in a vector of models.
* **make\_dynamic\_atomic\_model<>()** is a template method defined in <cadmium/modeling/dynamic\_model\_translator.hpp>. It is used to create an instance of an atomic model. It takes the class type of the atomic model, TIME (because all atomic models are templated classes that need to be instantiated with a TIME data type), and all the types of the parameters for the model constructor. The parameters of the method are the name of the atomic model (a string) and the parameters we need to pass to the constructor. If a parameter in the constructor is a pointer, we need to use the C++ method move() to pass the pointer to the constructor.

The instances of the atomic models are created (along with the coupled models, the logger and the runner) inside the main function defined in a cpp file. At the top of the cpp, we MUST include the headers of all the atomic classes we are using.

An atomic instance is created as follows:

**shared\_ptr<dynamic::modeling::model>** name\_atomic\_model\_instance **= dynamic::translate::make\_dynamic\_atomic\_model**< atomic\_model\_name, **TIME> ("**instance\_name**");**

To store the instances of atomic models, Cadmium uses one advanced C++ data type: shared\_ptr<>, defined in the standard library. shared\_ptr<> is a smart pointer that allows shared ownership of an object through a pointer.

Cadmium’s Services for Coupled Models

Coupled models are defined inside the main function in a cpp file (along with the instances of the atomic models, the logger and the runner).

Coupled models are defined using C++ functions and data types defined in the simulator. The functions were built following the formal definitions for DEVS coupled models. Therefore, each of the components defined formally for DEVS coupled models can be included.

# Declaring ports

Port declaration for coupled models is done using the same method as for atomic models.

//Port definition

**struct** model\_name\_ports\_defs{

**struct** out\_port\_name1 **: public out\_port<**message\_type\_1**> {};**

**struct** out\_port\_name2 **: public out\_port<**message\_type\_2**> {};**

**struct** in\_port\_name1 **: public in\_port<**message\_type\_3**> {};**

**struct** in\_port\_name2 **: public in\_port<**message\_type\_4**> {};**

**};**

In coupled models, we can omit grouping all the ports in a single structure and declare them as follows:

**struct** out\_port\_name1 **: public out\_port<**message\_type\_1**> {};**

**struct** out\_port\_name2 **: public out\_port<**message\_type\_2**> {};**

**struct** in\_port\_name1 **: public in\_port<**message\_type\_3**> {};**

**struct** in\_port\_name2 **: public in\_port<**message\_type\_4**> {};**

If there are two **different** coupled models using the same port type (i.e., with the same name and message type), we declare the port type once and we use it for both models. However, the **same** coupled model cannot have two ports with the same name (i.e. in our C++ implementation, they cannot be the same type).

# Defining coupled models

### Coupled model ports

To assign the input and output ports we already declared to a coupled model, Cadmium use the data type **Ports. Ports** is a data type used to define input and output ports. It is defined in <cadmium/modeling/dynamic\_model.hpp> under the namespace dynamic::modeling. It is a vector that takes as elements the typeid of the port structure declaration. To provide the type of a port, we use the method typeid() defined in the std C++ library (typeid() takes a data type as input).

**Input ports**

We need to create a variable of type Ports (in this generic example, **iports**\_coupled\_name **)** to store all the input ports as follows:

**dynamic::modeling::Ports iports**\_coupled\_name **= {   
 typeid(**model\_name\_ports\_defs::in\_port\_name1**),**

**typeid(**model\_name\_ports\_defs::in\_port\_name2**)**

**};**

**Output ports**

The output ports are also stored inside a variable of type Ports in the same way.

**dynamic::modeling::Ports oports**\_coupled\_name **= {  
 typeid(**model\_name\_ports\_defs::out\_port\_name1**),**

**typeid(**model\_name\_ports\_defs::out\_port\_name2**)**

**};**

### Submodels

Submodels are stored inside a variable of type **Models**. As already explained, **Models** is used to define the components of a coupled model. It is defined in <cadmium/modeling/dynamic\_model.hpp> under the namespace dynamic::modeling. It is a vector that takes as elements pointers to models (shared\_ptr<dynamic::modeling::model>>)

In this generic case, the name of the variable is **submodels**\_coupled\_name. It contains the instances of submodels inside the coupled model. In this generic case, name\_component\_instance1 and name\_component\_instance2. It does not matter the order we use to specify the components of the top model.

**dynamic::modeling::Models submodels**\_coupled\_name **=**

**{** name\_component\_instance1**,**

name\_component\_instance2**};**

name\_ component \_instance\_x are the names given to the variables that store the instance of the components of the coupled model. They can be instances of atomic models or coupled models.

### External Input Couplings (EICs)

Cadmium provides a data type and a method to define EICs.

The **EICs** data type is used to define the set of external input couplings. The set is stored as a vector with elements of type **EIC,** another data type to define each external input. It is implemented as a structure with two elements: the name of the submodel connected to the external input (a string), and a link that represents the external input (a shared\_ptr<>). Both **EICs** and **EIC** are defined in <cadmium/modeling/dynamic\_model.hpp> under the namespace dynamic::modeling.

**make\_EIC<>()** is used to create an EIC structure. It is defined in <cadmium/modeling/dynamic\_model\_translator.hpp>, and it returns an element of type EIC. It takes template parameters of the types of the input ports of the coupled model and the submodel inside the coupled model, in this specific order (i.e. form – to). It uses as parameter a string with the name of the submodel.

**dynamic::modeling::EICs eics\_**\_coupled\_name **= {**

**dynamic::translate::make\_EIC<**model\_name\_ports\_defs::in\_port\_name1**,**

component\_port\_name**>("**instance\_name**"),**

**dynamic::translate::make\_EIC<** model\_name\_ports\_defs::in\_port\_name2**,**

component\_port\_name2**>("**instance\_name2**")**

**};**

In this generic case, the name of the variable is **eics**\_coupled\_name. It contains two EICs: (1) the input port “in\_port\_name1” of the coupled model is connected to the input port “component\_port\_name” of the subcomponent “instance\_name”; (2) the input port “in\_port\_name2” of the coupled model is connected to the input port “component\_port\_name2” of the subcomponent “instance\_name2”.

instance\_name1(and 2) are unique names given to each instance of the components of the coupled model. They can be instances of atomic models or coupled models.

### External Output Couplings (EOCs)

Cadmium provides a data type and a method to define EOCs.

**EOCs** is a data type similar to **EICs** above, but for the External Output Couplings.

**make\_EOC<>()** is a method similar to **make\_EIC<>()** above, but for the External Output Couplings. It returns an element of type EOC, using the types of the output port of the submodel in the coupled model, and the output port of the coupled model, in this specific order (i.e. form – to). The parameter of the method is a string with the name of the submodel.

**dynamic::modeling::EOCs eocs\_**\_coupled\_name **= {**

**dynamic::translate::make\_EOC<**component\_port\_name, model\_name\_ports\_defs::

out\_port\_name1**>("**instance\_name**"),**

**dynamic::translate::make\_EOC<**component\_port\_name2, model\_name\_ports\_defs::

in\_port\_name2**>("**instance\_name2**")**

**};**

In this generic case, the name of the variable is **eocs**\_coupled\_name. It contains two EOCs: (1) the output port “component\_port\_name” of the subcomponent “instance\_name” is connected to the output port “out\_port\_name1” of the coupled model; (2) the output port “component\_port\_name2” of the subcomponent “instance\_name2” is connected to the output port “out\_port\_name2” of the coupled model.

instance\_name1(and 2) are unique names given to each instance of the components of the coupled model. They can be instances of atomic models or coupled models.

### Internal Couplings (ICs)

Cadmium provides a data type and a method to define the ICs.

**ICs** is a data type to define internal couplings. It is stored as a vector that takes elements of type **IC,** used to define each internal connection. It is implemented as a structure with three elements: (1) the name of the “from” component (i.e. a string), (2) the name of the “to” component (i.e. a string), and (3) a link to connect the output port of one component with the input port of the other component. They are defined in <cadmium/modeling/dynamic\_model.hpp> under the namespace dynamic::modeling.

**make\_IC<>()** is used to create the internal couplings, i.e., elements of type IC. It is defined in <cadmium/modeling/dynamic\_model\_translator.hpp>. It uses the type of the output port of the submodel “from” and the type of the input port of the submodel “to”, in this specific order (i.e. from output port– to input port). The parameters of the method are two strings, the first one with the name of the “from” submodel and the second one with the name of the “to” submodel (i.e. from submodel name – to submodel name).

**dynamic::modeling::ICs ics\_**\_coupled\_name **= {**

**dynamic::translate::make\_EIC<**component\_port\_name\_out1, component\_port\_name\_in1**>("**instance\_name\_out1**","**instance\_name\_in1**"),**

**dynamic::translate::make\_EIC<**component\_port\_name\_out2, component\_port\_name\_in2**>("**instance\_name\_out2**","**instance\_name\_in2**")**

**};**

In this generic case, the name of the variable is **ics**\_coupled\_name. It contains two ICs: (1) the output port “component\_port\_name\_out1” of the subcomponent “instance\_name\_out1” is connected to the input port “component\_port\_name\_in1” of the subcomponent “instance\_name\_in1”; (2) the output port “component\_port\_name\_out2” of the subcomponent “instance\_name\_out2” is connected to the input port “component\_port\_name\_in2” of the subcomponent “instance\_name\_in2”.

instance\_name\_in/out\_1 (and 2) are unique names given to each instance of the components of the coupled model. They can be instances of atomic models or coupled models.

### Coupled model variable

Cadmium defines the class **coupled<TIME>** to define coupled models. We use it to create coupled models’ instances. It is defined in <cadmium/modeling/dynamic\_coupled.hpp> under the namespace dynamic::modeling. The class uses seven variables: (1) a string with the model name, (2) a variable of type Models representing the subcomponents, (3) a variable of type Ports for the input ports, (4) a variable of type Ports for the output ports, (5) a variable of type EICs for external input couplings, (6) a variable of type EOCs for external output couplings and (7) a variable of type ICs for internal couplings. The constructor of this class takes all these parameters in this specific order.

To create the coupled model, we need to define all the elements explained in this section (i.e. input ports, output ports, submodels, EICs, EOCs, and ICs).

We declare the variable where the coupled model will be stored, in this generic case, coupled\_name\_variable. The variable where the coupled model is stored is of the data type shared\_ptr<dynamic::modeling::coupled<TIME>> defined in the simulator.

To create an instance of the coupled model, we use the C++ method make\_shared<>(). make\_shared<>() is a method that allows creating a shared\_ptr<>. It uses as a template parameter the data type that will be stored in the pointer, and as function parameters the constructor parameters for the data type. In Cadmium, to create coupled models, the function parameters are the ones used in the constructor of the class **coupled<TIME>:** (1) model name, (2) components, (3) input ports, (4) output ports, (5) EICs, (6) EOCs and (7) ICs in this specific order as in the following general example:

**shared\_ptr<dynamic::modeling::coupled<TIME>>** coupled\_name\_variable **= make\_shared<dynamic::modeling::coupled<TIME>>(**

**"**coupled\_name**", submodels\_**coupled\_name**, iports\_**coupled\_name**,**

**oports\_**coupled\_name**, eics\_**coupled\_name**, eocs\_**coupled\_name**, ics\_**coupled\_name

**);**

coupled\_name is a unique name given to the coupled model.

The resulting coupled model can be used inside other coupled models.

Cadmium’s Services to create Logs

Cadmium also provides services for generating logs of the simulation. There are two basic logs: (1) messages generated on the output ports and (2) state of the atomic model.

The logs are defined as follows:

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* Loggers \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

static ofstream out\_messages("../simulation\_results/messages\_log.txt");

struct oss\_sink\_messages{

static ostream& sink(){

return out\_messages;

}

};

static ofstream out\_state("../simulation\_results/output\_state\_log.txt");

struct oss\_sink\_state{

static ostream& sink(){

return out\_state;

}

};

using state=logger::logger<logger::logger\_state, dynamic::logger::formatter<TIME>, oss\_sink\_state>;

using log\_messages=logger::logger<logger::logger\_messages, dynamic::logger::formatter<TIME>, oss\_sink\_messages>;

using global\_time\_mes=logger::logger<logger::logger\_global\_time, dynamic::logger::formatter<TIME>, oss\_sink\_messages>;

using global\_time\_sta=logger::logger<logger::logger\_global\_time, dynamic::logger::formatter<TIME>, oss\_sink\_state>;

using logger\_top=logger::multilogger<state, log\_messages, global\_time\_mes, global\_time\_sta>;

First, we need to define the file where we will output the message log. To do so, we create a variable (out\_messages) of type ofstream. We initialize out\_messages with the path to the output file for the message log ("../simulation\_results/messages\_log.txt").

We then define the structure oss\_sink\_messages to tell the simulator where we will save the output log. The structure uses a method (sink) that returns a pointer to out\_messages. We use oss\_sink\_messages to declare the message logger.

We need to do the same for the state variable log. To do so, we define a variable (out\_state) of type ofstream. We initialize out\_state with the path to the output file for the state log ( "../simulation\_results/state\_log.txt").

Finally, we define the structure (oss\_sink\_state) to tell the simulator where to save the state log. The structure has a method (sink) that returns a pointer to out\_state. We will use oss\_sink\_state to declare the state logger.

To define the logger, we need to include the following declarations:

**using state = logger::logger<logger::logger\_state, dynamic::logger::formatter<TIME>, oss\_sink\_state>;**

It defines the state logger. We instantiate the logger with: (1) the logger we are using, in this case **logger\_state** (defined in <cadmium/logger/logger.hpp>), (2) the formatter (defined in <cadmium/logger/ dynamic\_common\_loggers.hpp>), and (3) the sink we just defined (i.e. oss\_sink\_state).

All logs are defined in the same way. Only the first and third template parameters changes because they are the ones that specify which log we are using and where we generate the log.

**using log\_messages = logger::logger<logger::logger\_messages, dynamic::logger::formatter<TIME>, oss\_sink\_messages>;**

It defines the message logger. As in the previous case, we instantiate (1) the logger we are using, in this case, **logger\_messages** (defined in <cadmium/logger/logger.hpp>), (2) the formatter (defined in <cadmium/logger/ dynamic\_common\_loggers.hpp>), and (3) the sink just defined (oss\_sink\_messages).

In order to include the global time of the simulation inside the state and message log, we need to declare a new logger: global\_time. In this specific case, we need two: one for the messages and one for the states because the logs are generated on different files.

**using global\_time\_mes = logger::logger<logger::logger\_global\_time, dynamic::logger::formatter<TIME>, oss\_sink\_messages>;**

It defines the global time for the message logger. As in the previous case, we instantiate with (1) the logger we are using, in this case **logger\_global\_time** (defined in <cadmium/logger/logger.hpp>), (2) the formatter (defined in <cadmium/logger/ dynamic\_common\_loggers.hpp>), and (3) the sink (oss\_sink\_messages).

**using global\_time\_sta = logger::logger<logger::logger\_global\_time, dynamic::logger::formatter<TIME>, oss\_sink\_state>;**

It defines the global time for the state logger as in the previous cases.

Once we have declared all the loggers we need, we have to combine them, so our simulation generates all the logs at the same time. For this purpose, we use the **multilogger** structure defined in <cadmium/logger/logger.hpp> instantiated with the above log definitions (i.e. **state, log\_messages, global\_time\_mes, global\_time\_sta)** as template parameters:

**using logger\_top = logger::multilogger<state, log\_messages, global\_time\_mes, global\_time\_sta>;**

Cadmium’s Services to Run the Simulation

Cadmium provides a templated class to execute the model: runner. The runner class defined in <cadmium/engine/pdevs\_dynamic\_runner.hpp> under the namespace dynamic::engine:: takes two template parameters: the class used for the time (in this example, NDTime) and a logger (in this case, logger\_top). The parameters for the class constructor are the name of the top model (TOP in this generic case) and the initial time for the simulation (usually 0).

dynamic::engine::runner<NDTime, logger\_top> r(TOP, {0});

To define the end time of the simulation, we have two options: (1) run the simulation until a specific time or (2) run the simulation until all models are passivated.

To run the simulation until a specific time we use the runner method run\_until(). This method takes as parameter the end time of the simulation.

r.run\_until(TIME("04:00:00:000"));

To run the simulation until all models are passivated, we use the runner method run\_until\_passivate(). This method does not take any parameter.

r.run\_until\_passivate();

Services to Export Coupled Models to JSON

In order to visualize the simulation results using DEVS Viewer, (<https://staubibr.github.io/arslab-prd/app-simple/index.html>), you need to write a DEVS coupled model using JSON.

The library “Cadmium Model JSON Exporter" automatically generates this JSON file from the implementation of the model in Cadmium. “Cadmium Model JSON Exporter" is included as a submodule when you download “Cadmium-Simulation-Environment” following the instructions provided at the beginning of this manual.

It is not necessary to understand the content of the JSON file generated. You do not even need to open it, just keep it handy if you are going to use DEVS Viewer to visualize your simulation results.

### How do we generate the JSON file?

To use the services defined in “Cadmium Model JSON Exporter" to generate the JSON file, you need to follow these steps:

1. Locate your main file (i.e. the one you used to define the coupled model) and place the following include statement at the top of the file (see Appendix D). This will allow you to use the functions defined in “Cadmium Model JSON Exporter" library:

//Json exporter header

#include <dynamic\_json\_exporter.hpp>

2. You need to call the function dynamic\_export\_model\_to\_json(), which transforms a DEVS coupled model implemented in Cadmium to JSON and stores it in the specified file. To do so, you just need to add the following code at the end of your main file (see Appendix D). In the Appendix D, we called the function after calling the runner, just before the return statement.

static ofstream out\_JSON("ABP\_json.json");   
dynamic\_export\_model\_to\_json(out\_JSON, TOP);

In the first line, we define out\_JSON variable that stores the location of the JSON file where the output will be stored. "ABP\_json.json" is the relative path and the name for the json output file. Make sure to use .json as extension.

In the second line, we call the function dynamic\_export\_model\_to\_json. The function takes two input parameters. The fist parameter is the out\_JSON variable we defined above. The second parameter is the variable where you stored your top coupled model, usually TOP. This is the same variable you use in the runner.

To generate the JSON file, you will need to make the following additions (highlighted in red) in your makefile:

INCLUDECADMIUM=-I ../../cadmium/include

INCLUDEJSON=-I ../../CadmiumModelJSONExporter/include

INCLUDEDESTIMES=-I ../../DESTimes/include

[ ... ]

main\_top.o: top\_model/main.cpp

$(CC) -g -c $(CFLAGS) $(INCLUDECADMIUM) $(INCLUDEDESTIMES) $(INCLUDEJSON) top\_model/main.cpp -o build/main\_top.o

[ ... ]\

The JSON file will be generated in your "bin" folder (*Author: Françoise Blanc*).

### Additional requirements to use the viewer

To use the DEVS viewer, the logs for the state and messages must be defined using a specific format.

**Specific format for state output.**

The definition of the << operator for the structure state\_type in atomic must follow the following format:

< state.var1, state.var2, …., state.varN>

This will be generically implemented as follows:

**friend ostringstream& operator << (ostringstream& os, const typename** atomic\_model\_name**<TIME>::state\_type& state) {**

os << "<" << state.var1 << ", " << state.var2 << ">";

**return os;**

}

For a specific example for the ABP protocol, check Appendix B.

**Specific format for messages output.**

**The message output** (see figure 4 in section “DEVS Model definition: An Example”) should follow the same format as the state.

< msg.var1, msg.var2, …., msg.varN>

The following code snippet represents how to implement this format for the for the ABP model.

//Output the content of the structure

ostream& operator<<(ostream& os, const Message\_t& msg) {

os << "<" << msg.packet << ", " << msg.bit << ">";

return os;

}

Appendix A

Template for the definition of an atomic model.

**#ifndef** ATOMIC\_MODEL\_NAME\_HPP

**#define** ATOMIC\_MODEL\_NAME\_HPP

//Include simulator headers

**#include <cadmium/modeling/ports.hpp>**

**#include <cadmium/modeling/message\_bag.hpp>**

//Include other headers needed for the C++ implementation of the model

#include <limits>

#include <math.h>

#include <assert.h>

//Include the relative path to the message types

#include "../data\_structures/message.hpp"

**using namespace cadmium;**

**using namespace std;**

//Port definition

**struct** model\_name\_ports\_defs{

**struct** out\_port\_name1 **: public out\_port<**message\_type\_1**> {};**

**struct** out\_port\_name2 **: public out\_port<**message\_type\_2**> {};**

**struct** in\_port\_name1 **: public in\_port<**message\_type\_3**> {};**

**struct** in\_port\_name2 **: public in\_port<**message\_type\_4**> {};**

**};**

//Atomic model class

**template<typename TIME>** **class** model\_name {

**public:**

//Ports definition

**using input\_ports = tuple<typename** model\_name\_ports\_defs**::** in\_port\_name1**,**   
 **typename** model\_name\_ports\_defs**::** in\_port\_name2**>;**

**using output\_ports = tuple<typename** model\_name\_ports\_defs**::** out\_port\_name1**,**   
 **typename** model\_name\_ports\_defs**::** out\_port\_name2**>;**

//Model parameters to be overwritten during instantiation

**struct state\_type{**

//Declare the state variables here

**};**

**state\_type state;**

//Default constructor without parameters

model\_name **() noexcept{**

**//Define the default constructor here**

}

//Constructor with parameters if needed

**void internal\_transition() {**

//Define internal transition here

**}**

**void external\_transition(TIME e, typename make\_message\_bags<input\_ports>::type mbs){**

//Define external transition here

}

**void confluence\_transition(TIME e,typename make\_message\_bags<input\_ports>::type mbs){**

//Define confluence transition here

//Default definition

internal\_transition();

external\_transition(TIME(), std::move(mbs));

**}**

**typename make\_message\_bags<output\_ports>::type output() const {**

**typename make\_message\_bags<output\_ports>::type bags;**

//Define output function here

**return bags;**

**}**

**TIME time\_advance() const {**

**TIME next\_internal;**

//Define time advance function here

**return next\_internal;**

**}**

**friend ostringstream& operator<<(ostringstream& os, const typename Subnet<TIME>::state\_type& state) {**

//Define how to log the state here

**return os;**

}

};

**#endif //**ATOMIC\_MODEL\_NAME\_HPP

Appendix B

Implementation of the receiver atomic class

#ifndef \_\_RECEIVER\_HPP\_\_

#define \_\_RECEIVER\_HPP\_\_

#include <cadmium/modeling/ports.hpp>

#include <cadmium/modeling/message\_bag.hpp>

#include <limits>

#include <assert.h>

#include <string>

#include "../data\_structures/message.hpp"

using namespace cadmium;

using namespace std;

//Port definition

struct Receiver\_defs{

struct out : public out\_port<Message\_t> { };

struct in : public in\_port<Message\_t> { };

};

template<typename TIME> class Receiver{

public:

//Parameters to be overwritten when instantiating the atomic model

TIME preparationTime;

// default constructor

Receiver() noexcept{

preparationTime = TIME("00:00:10");

state.ackNum = 0;

state.sending = false;

}

// state definition

struct state\_type{

int ackNum;

bool sending;

};

state\_type state;

// ports definition

using input\_ports=std::tuple<typename Receiver\_defs::in>;

using output\_ports=std::tuple<typename Receiver\_defs::out>;

// internal transition

void internal\_transition() {

state.sending = false;

}

// external transition

void external\_transition(TIME e, typename make\_message\_bags<input\_ports>::type mbs) {

if(get\_messages<typename Receiver\_defs::in>(mbs).size()>1)

assert(false && "one message per time unit");

vector<Message\_t> message\_port\_in;

message\_port\_in = get\_messages<typename Receiver\_defs::in>(mbs);

state.ackNum = message\_port\_in[0].bit;

state.sending = true;

}

// confluence transition

void confluence\_transition(TIME e, typename make\_message\_bags<input\_ports>::type mbs) {

internal\_transition();

external\_transition(TIME(), std::move(mbs));

}

// output function

typename make\_message\_bags<output\_ports>::type output() const {

typename make\_message\_bags<output\_ports>::type bags;

Message\_t out\_aux;

out\_aux = Message\_t(0, state.ackNum);

get\_messages<typename Receiver\_defs::out>(bags).push\_back(out\_aux);

return bags;

}

// time\_advance function

TIME time\_advance() const {

TIME next\_internal;

if (state.sending) {

next\_internal = preparationTime;

}else {

next\_internal = std::numeric\_limits<TIME>::infinity();

}

return next\_internal;

}

friend std::ostringstream& operator<<(std::ostringstream& os, const typename Receiver<TIME>::state\_type& i) {

os << "< " << i.ackNum << "> ";

return os;

}

};

#endif // \_\_RECEIVER\_HPP\_\_

Appendix C

Implementation of the sender atomic class

#ifndef \_\_SENDER\_HPP\_\_

#define \_\_SENDER\_HPP\_\_

#include <cadmium/modeling/ports.hpp>

#include <cadmium/modeling/message\_bag.hpp>

#include <limits>

#include <assert.h>

#include <string>

#include <random>

#include "../data\_structures/message.hpp"

using namespace cadmium;

using namespace std;

//Port definition

struct Sender\_defs{

struct packetSentOut : public out\_port<int> { };

struct ackReceivedOut : public out\_port<int> {};

struct dataOut : public out\_port<Message\_t> { };

struct controlIn : public in\_port<int> { };

struct ackIn : public in\_port<Message\_t> { };

};

template<typename TIME> class Sender{

public:

//Parameters to be overwritten when instantiating the atomic model

TIME preparationTime;

TIME timeout;

// default constructor

Sender() noexcept{

preparationTime = TIME("00:00:10");

timeout = TIME("00:00:20");

state.alt\_bit = 0;

state.next\_internal = std::numeric\_limits<TIME>::infinity();

state.model\_active = false;

}

// state definition

struct state\_type{

bool ack;

int packetNum;

int totalPacketNum;

int alt\_bit;

bool sending;

bool model\_active;

TIME next\_internal;

};

state\_type state;

// ports definition

using input\_ports=std::tuple<typename Sender\_defs::controlIn, typename Sender\_defs::ackIn>;

using output\_ports=std::tuple<typename Sender\_defs::packetSentOut, typename Sender\_defs::ackReceivedOut, typename Sender\_defs::dataOut>;

// internal transition

void internal\_transition() {

if (state.ack){

if (state.packetNum < state.totalPacketNum){

state.packetNum ++;

state.ack = false;

state.alt\_bit = (state.alt\_bit + 1) % 2;

state.sending = true;

state.model\_active = true;

state.next\_internal = preparationTime;

} else {

state.model\_active = false;

state.next\_internal = std::numeric\_limits<TIME>::infinity();

}

} else{

if (state.sending){

state.sending = false;

state.model\_active = true;

state.next\_internal = timeout;

} else {

state.sending = true;

state.model\_active = true;

state.next\_internal = preparationTime;

}

}

}

// external transition

void external\_transition(TIME e, typename make\_message\_bags<input\_ports>::type mbs) {

if((get\_messages<typename Sender\_defs::controlIn>(mbs).size()+get\_messages<typename Sender\_defs::ackIn>(mbs).size())>1)

assert(false && "one message per time unit");

for(const auto &x : get\_messages<typename Sender\_defs::controlIn>(mbs)){

if(state.model\_active == false){

state.totalPacketNum = x;

if (state.totalPacketNum > 0){

state.packetNum = 1;

state.ack = false;

state.sending = true;

state.alt\_bit = 0; //set initial alt\_bit

state.model\_active = true;

state.next\_internal = preparationTime;

}else{

if(state.next\_internal != std::numeric\_limits<TIME>::infinity()){

state.next\_internal = state.next\_internal - e;

}

}

}

}

for(const auto &x : get\_messages<typename Sender\_defs::ackIn>(mbs)){

if(state.model\_active == true) {

if (state.alt\_bit == x.bit) {

state.ack = true;

state.sending = false;

state.next\_internal = TIME("00:00:00");

}else{

if(state.next\_internal != std::numeric\_limits<TIME>::infinity()){

state.next\_internal = state.next\_internal - e;

}

}

}

}

}

// confluence transition

void confluence\_transition(TIME e, typename make\_message\_bags<input\_ports>::type mbs) {

internal\_transition();

external\_transition(TIME(), std::move(mbs));

}

// output function

typename make\_message\_bags<output\_ports>::type output() const {

typename make\_message\_bags<output\_ports>::type bags;

Message\_t out;

if (state.sending){

out.packet = state.packetNum;

out.bit = state.alt\_bit;

get\_messages<typename Sender\_defs::dataOut>(bags).push\_back(out);

get\_messages<typename Sender\_defs::packetSentOut>(bags).push\_back(state.packetNum);

}else{

if (state.ack){

get\_messages<typename Sender\_defs::ackReceivedOut>(bags).push\_back(state.alt\_bit);

}

}

return bags;

}

// time\_advance function

TIME time\_advance() const {

return state.next\_internal;

}

friend std::ostringstream& operator<<(std::ostringstream& os, const typename Sender<TIME>::state\_type& i) {

os << "packetNum: " << i.packetNum << " & totalPacketNum: " << i.totalPacketNum;

return os;

}

};

#endif // \_\_SENDER\_HPP\_\_

Appendix D

Implementation of the ABP coupled model

//Cadmium Simulator headers

#include <cadmium/modeling/ports.hpp>

#include <cadmium/modeling/dynamic\_model.hpp>

#include <cadmium/modeling/dynamic\_model\_translator.hpp>

#include <cadmium/engine/pdevs\_dynamic\_runner.hpp>

#include <cadmium/logger/common\_loggers.hpp>

//Json exporter header

#include <dynamic\_json\_exporter.hpp>

//Time class header

#include <NDTime.hpp>

//Messages structures

#include "../data\_structures/message.hpp"

//Atomic model headers

#include <cadmium/basic\_model/pdevs/iestream.hpp> //Atomic model for inputs

#include "../atomics/subnet.hpp"

#include "../atomics/sender.hpp"

#include "../atomics/receiver.hpp"

//C++ headers

#include <iostream>

#include <chrono>

#include <algorithm>

#include <string>

using namespace std;

using namespace cadmium;

using namespace cadmium:: and cadmium::basic\_models::pdevs;

using TIME = NDTime;

/\*\*\*\*\* Define input port for coupled models \*\*\*\*\*/

struct inp\_control : public in\_port<int>{};

struct inp\_1 : public in\_port<Message\_t>{};

struct inp\_2 : public in\_port<Message\_t>{};

/\*\*\*\*\* Define output ports for coupled model \*\*\*\*\*/

struct outp\_ack : public out\_port<int>{};

struct outp\_1 : public out\_port<Message\_t>{};

struct outp\_2 : public out\_port<Message\_t>{};

struct outp\_pack : public out\_port<int>{};

/\*\*\*\*\*\* Input Reader atomic model declaration \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

template<typename T>

class InputReader\_Int : public iestream\_input<int,T> {

public:

InputReader\_Int() = default;

InputReader\_Int(const char\* file\_path) : iestream\_input<int,T>(file\_path) {}

};

int main(int argc, char \*\* argv) {

if (argc < 2) {

cout << "Program used with wrong parameters. The program must be invoked as follow:";

cout << argv[0] << " path to the input file " << endl;

return 1;

}

/\*\*\*\*\*\* Input Reader atomic model instantiation \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

string input = argv[1];

const char \* i\_input = input.c\_str();

shared\_ptr<dynamic::modeling::model> input\_reader = dynamic::translate::make\_dynamic\_atomic\_model<InputReader\_Int, TIME, const char\* >("input\_reader" , move(i\_input));

/\*\*\*\*\*\* Sender atomic model instantiation \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

shared\_ptr<dynamic::modeling::model> sender1 = dynamic::translate::make\_dynamic\_atomic\_model<Sender, TIME>("sender1");

/\*\*\*\*\*\* Receiver atomic model instantiation \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

shared\_ptr<dynamic::modeling::model> receiver1 = dynamic::translate::make\_dynamic\_atomic\_model<Receiver, TIME>("receiver1");

/\*\*\*\*\*\* Subnet atomic models instantiation \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

shared\_ptr<dynamic::modeling::model> subnet1 = dynamic::translate::make\_dynamic\_atomic\_model<Subnet, TIME>("subnet1");

shared\_ptr<dynamic::modeling::model> subnet2 = dynamic::translate::make\_dynamic\_atomic\_model<Subnet, TIME>("subnet2");

/\*\*\*\*\*\*\*NETWORKS COUPLED MODEL\*\*\*\*\*\*\*\*/

dynamic::modeling::Ports iports\_Network = {typeid(inp\_1),typeid(inp\_2)};

dynamic::modeling::Ports oports\_Network = {typeid(outp\_1),typeid(outp\_2)};

dynamic::modeling::Models submodels\_Network = {subnet1, subnet2};

dynamic::modeling::EICs eics\_Network = {

dynamic::translate::make\_EIC<inp\_1, Subnet\_defs::in>("subnet1"),

dynamic::translate::make\_EIC<inp\_2, Subnet\_defs::in>("subnet2")

};

dynamic::modeling::EOCs eocs\_Network = {

dynamic::translate::make\_EOC<Subnet\_defs::out,outp\_1>("subnet1"),

dynamic::translate::make\_EOC<Subnet\_defs::out,outp\_2>("subnet2")

};

dynamic::modeling::ICs ics\_Network = {};

shared\_ptr<dynamic::modeling::coupled<TIME>> NETWORK;

NETWORK = make\_shared<dynamic::modeling::coupled<TIME>>(

"Network", submodels\_Network, iports\_Network, oports\_Network, eics\_Network, eocs\_Network, ics\_Network

);

/\*\*\*\*\*\*\*ABP SIMULATOR COUPLED MODEL\*\*\*\*\*\*\*\*/

dynamic::modeling::Ports iports\_ABP = {typeid(inp\_control)};

dynamic::modeling::Ports oports\_ABP = {typeid(outp\_ack),typeid(outp\_pack)};

dynamic::modeling::Models submodels\_ABP = {sender1, receiver1, NETWORK};

dynamic::modeling::EICs eics\_ABP = {

cadmium::dynamic::translate::make\_EIC<inp\_control, Sender\_defs::controlIn> ("sender1")

};

dynamic::modeling::EOCs eocs\_ABP = {

dynamic::translate::make\_EOC<Sender\_defs::packetSentOut,outp\_pack>("sender1"),

dynamic::translate::make\_EOC<Sender\_defs::ackReceivedOut,outp\_ack>("sender1")

};

dynamic::modeling::ICs ics\_ABP = {

dynamic::translate::make\_IC<Sender\_defs::dataOut, inp\_1> ("sender1","Network"),

dynamic::translate::make\_IC<outp\_2, Sender\_defs::ackIn> ("Network","sender1"),

dynamic::translate::make\_IC<Receiver\_defs::out, inp\_2> ("receiver1","Network"),

dynamic::translate::make\_IC<outp\_1, Receiver\_defs::in> ("Network","receiver1")

};

shared\_ptr<dynamic::modeling::coupled<TIME>> ABP;

ABP = make\_shared<dynamic::modeling::coupled<TIME>>(

"ABP", submodels\_ABP, iports\_ABP, oports\_ABP, eics\_ABP, eocs\_ABP, ics\_ABP

);

/\*\*\*\*\*\*\*TOP COUPLED MODEL\*\*\*\*\*\*\*\*/

dynamic::modeling::Ports iports\_TOP = {};

dynamic::modeling::Ports oports\_TOP = {typeid(outp\_pack),typeid(outp\_ack)};

dynamic::modeling::Models submodels\_TOP = {input\_reader, ABP};

dynamic::modeling::EICs eics\_TOP = {};

dynamic::modeling::EOCs eocs\_TOP = {

dynamic::translate::make\_EOC<outp\_pack,outp\_pack>("ABP"),

dynamic::translate::make\_EOC<outp\_ack,outp\_ack>("ABP")

};

dynamic::modeling::ICs ics\_TOP = {

dynamic::translate::make\_IC<iestream\_input\_defs<int>::out, inp\_control> ("input\_reader","ABP")

};

shared\_ptr<cadmium::dynamic::modeling::coupled<TIME>> TOP;

TOP = make\_shared<dynamic::modeling::coupled<TIME>>(

"TOP", submodels\_TOP, iports\_TOP, oports\_TOP, eics\_TOP, eocs\_TOP, ics\_TOP

);

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* Loggers \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

static ofstream out\_messages("../simulation\_results/ABP\_output\_messages.txt");

struct oss\_sink\_messages{

static ostream& sink(){

return out\_messages;

}

};

static ofstream out\_state("../simulation\_results/ABP\_output\_state.txt");

struct oss\_sink\_state{

static ostream& sink(){

return out\_state;

}

};

using state=logger::logger<logger::logger\_state, dynamic::logger::formatter<TIME>, oss\_sink\_state>;

using log\_messages=logger::logger<logger::logger\_messages, dynamic::logger::formatter<TIME>, oss\_sink\_messages>;

using global\_time\_mes=logger::logger<logger::logger\_global\_time, dynamic::logger::formatter<TIME>, oss\_sink\_messages>;

using global\_time\_sta=logger::logger<logger::logger\_global\_time, dynamic::logger::formatter<TIME>, oss\_sink\_state>;

using logger\_top=logger::multilogger<state, log\_messages, global\_time\_mes, global\_time\_sta>;

/\*\*\*\*\*\*\*\*\*\*\*\*\*\* Runner call \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

dynamic::engine::runner<NDTime, logger\_top> r(TOP, {0});

r.run\_until\_passivate();

/\*\*\*\*\*\*\*\*\*\*\*\*\*\* JSON Exporter call \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

static ofstream out\_JSON("ABP\_json.json");   
 dynamic\_export\_model\_to\_json(out\_JSON, TOP);

return 0;

}

Appendix E

Message log for the ABP simulation

00:00:00:000

[iestream\_input\_defs<int>::out: {}] generated by model input\_reader

00:00:10:000

[iestream\_input\_defs<int>::out: {5}] generated by model input\_reader

00:00:20:000

[Sender\_defs::packetSentOut: {1}, Sender\_defs::ackReceivedOut: {}, Sender\_defs::dataOut: {1 0}] generated by model sender1

00:00:23:000

[Subnet\_defs::out: {1 0}] generated by model subnet1

00:00:33:000

[Receiver\_defs::out: {0 0}] generated by model receiver1

00:00:36:000

[Subnet\_defs::out: {0 0}] generated by model subnet2

00:00:36:000

[Sender\_defs::packetSentOut: {}, Sender\_defs::ackReceivedOut: {0}, Sender\_defs::dataOut: {}] generated by model sender1

00:00:46:000

[Sender\_defs::packetSentOut: {2}, Sender\_defs::ackReceivedOut: {}, Sender\_defs::dataOut: {2 1}] generated by model sender1

00:00:49:000

[Subnet\_defs::out: {2 1}] generated by model subnet1

00:00:59:000

[Receiver\_defs::out: {0 1}] generated by model receiver1

00:01:02:000

[Subnet\_defs::out: {0 1}] generated by model subnet2

00:01:02:000

[Sender\_defs::packetSentOut: {}, Sender\_defs::ackReceivedOut: {1}, Sender\_defs::dataOut: {}] generated by model sender1

00:01:12:000

[Sender\_defs::packetSentOut: {3}, Sender\_defs::ackReceivedOut: {}, Sender\_defs::dataOut: {3 0}] generated by model sender1

00:01:15:000

[Subnet\_defs::out: {3 0}] generated by model subnet1

00:01:25:000

[Receiver\_defs::out: {0 0}] generated by model receiver1

00:01:28:000

[Subnet\_defs::out: {0 0}] generated by model subnet2

00:01:28:000

[Sender\_defs::packetSentOut: {}, Sender\_defs::ackReceivedOut: {0}, Sender\_defs::dataOut: {}] generated by model sender1

00:01:38:000

[Sender\_defs::packetSentOut: {4}, Sender\_defs::ackReceivedOut: {}, Sender\_defs::dataOut: {4 1}] generated by model sender1

00:01:41:000

[Subnet\_defs::out: {4 1}] generated by model subnet1

00:01:51:000

[Receiver\_defs::out: {0 1}] generated by model receiver1

00:01:54:000

[Subnet\_defs::out: {0 1}] generated by model subnet2

00:01:54:000

[Sender\_defs::packetSentOut: {}, Sender\_defs::ackReceivedOut: {1}, Sender\_defs::dataOut: {}] generated by model sender1

00:02:04:000

[Sender\_defs::packetSentOut: {5}, Sender\_defs::ackReceivedOut: {}, Sender\_defs::dataOut: {5 0}] generated by model sender1

00:02:07:000

[Subnet\_defs::out: {5 0}] generated by model subnet1

00:02:17:000

[Receiver\_defs::out: {0 0}] generated by model receiver1

00:02:20:000

[Subnet\_defs::out: {0 0}] generated by model subnet2

00:02:20:000

[Sender\_defs::packetSentOut: {}, Sender\_defs::ackReceivedOut: {0}, Sender\_defs::dataOut: {}] generated by model sender1

00:15:00:000

[iestream\_input\_defs<int>::out: {3}] generated by model input\_reader

00:15:10:000

[Sender\_defs::packetSentOut: {1}, Sender\_defs::ackReceivedOut: {}, Sender\_defs::dataOut: {1 0}] generated by model sender1

00:15:13:000

[Subnet\_defs::out: {1 0}] generated by model subnet1

00:15:23:000

[Receiver\_defs::out: {0 0}] generated by model receiver1

00:15:26:000

[Subnet\_defs::out: {0 0}] generated by model subnet2

00:15:26:000

[Sender\_defs::packetSentOut: {}, Sender\_defs::ackReceivedOut: {0}, Sender\_defs::dataOut: {}] generated by model sender1

00:15:36:000

[Sender\_defs::packetSentOut: {2}, Sender\_defs::ackReceivedOut: {}, Sender\_defs::dataOut: {2 1}] generated by model sender1

00:15:39:000

[Subnet\_defs::out: {2 1}] generated by model subnet1

00:15:49:000

[Receiver\_defs::out: {0 1}] generated by model receiver1

00:15:52:000

[Subnet\_defs::out: {0 1}] generated by model subnet2

00:15:52:000

[Sender\_defs::packetSentOut: {}, Sender\_defs::ackReceivedOut: {1}, Sender\_defs::dataOut: {}] generated by model sender1

00:16:02:000

[Sender\_defs::packetSentOut: {3}, Sender\_defs::ackReceivedOut: {}, Sender\_defs::dataOut: {3 0}] generated by model sender1

00:16:22:000

[Sender\_defs::packetSentOut: {}, Sender\_defs::ackReceivedOut: {}, Sender\_defs::dataOut: {}] generated by model sender1

00:16:32:000

[Sender\_defs::packetSentOut: {3}, Sender\_defs::ackReceivedOut: {}, Sender\_defs::dataOut: {3 0}] generated by model sender1

00:16:35:000

[Subnet\_defs::out: {3 0}] generated by model subnet1

00:16:45:000

[Receiver\_defs::out: {0 0}] generated by model receiver1

00:16:48:000

[Subnet\_defs::out: {0 0}] generated by model subnet2

00:16:48:000

[Sender\_defs::packetSentOut: {}, Sender\_defs::ackReceivedOut: {0}, Sender\_defs::dataOut: {}] generated by model sender1

Appendix F

State log for the ABP simulation

00:00:00:000

State for model input\_reader is next time: 00:00:00:000

State for model sender1 is packetNum: 0 & totalPacketNum: 0

State for model receiver1 is ackNum: 0

State for model subnet1 is index: 0 & transmitting: 0

State for model subnet2 is index: 0 & transmitting: 0

00:00:00:000

State for model input\_reader is next time: 00:00:10:000

State for model sender1 is packetNum: 0 & totalPacketNum: 0

State for model receiver1 is ackNum: 0

State for model subnet1 is index: 0 & transmitting: 0

State for model subnet2 is index: 0 & transmitting: 0

00:00:10:000

State for model input\_reader is next time: 00:14:50:000

State for model sender1 is packetNum: 1 & totalPacketNum: 5

State for model receiver1 is ackNum: 0

State for model subnet1 is index: 0 & transmitting: 0

State for model subnet2 is index: 0 & transmitting: 0

00:00:20:000

State for model input\_reader is next time: 00:14:50:000

State for model sender1 is packetNum: 1 & totalPacketNum: 5

State for model receiver1 is ackNum: 0

State for model subnet1 is index: 1 & transmitting: 1

State for model subnet2 is index: 0 & transmitting: 0

00:00:23:000

State for model input\_reader is next time: 00:14:50:000

State for model sender1 is packetNum: 1 & totalPacketNum: 5

State for model receiver1 is ackNum: 0

State for model subnet1 is index: 1 & transmitting: 0

State for model subnet2 is index: 0 & transmitting: 0

00:00:33:000

State for model input\_reader is next time: 00:14:50:000

State for model sender1 is packetNum: 1 & totalPacketNum: 5

State for model receiver1 is ackNum: 0

State for model subnet1 is index: 1 & transmitting: 0

State for model subnet2 is index: 1 & transmitting: 1

00:00:36:000

State for model input\_reader is next time: 00:14:50:000

State for model sender1 is packetNum: 1 & totalPacketNum: 5

State for model receiver1 is ackNum: 0

State for model subnet1 is index: 1 & transmitting: 0

State for model subnet2 is index: 1 & transmitting: 0

00:00:36:000

State for model input\_reader is next time: 00:14:50:000

State for model sender1 is packetNum: 2 & totalPacketNum: 5

State for model receiver1 is ackNum: 0

State for model subnet1 is index: 1 & transmitting: 0

State for model subnet2 is index: 1 & transmitting: 0

00:00:46:000

State for model input\_reader is next time: 00:14:50:000

State for model sender1 is packetNum: 2 & totalPacketNum: 5

State for model receiver1 is ackNum: 0

State for model subnet1 is index: 2 & transmitting: 1

State for model subnet2 is index: 1 & transmitting: 0

00:00:49:000

State for model input\_reader is next time: 00:14:50:000

State for model sender1 is packetNum: 2 & totalPacketNum: 5

State for model receiver1 is ackNum: 1

State for model subnet1 is index: 2 & transmitting: 0

State for model subnet2 is index: 1 & transmitting: 0

00:00:59:000

State for model input\_reader is next time: 00:14:50:000

State for model sender1 is packetNum: 2 & totalPacketNum: 5

State for model receiver1 is ackNum: 1

State for model subnet1 is index: 2 & transmitting: 0

State for model subnet2 is index: 2 & transmitting: 1

00:01:02:000

State for model input\_reader is next time: 00:14:50:000

State for model sender1 is packetNum: 2 & totalPacketNum: 5

State for model receiver1 is ackNum: 1

State for model subnet1 is index: 2 & transmitting: 0

State for model subnet2 is index: 2 & transmitting: 0

00:01:02:000

State for model input\_reader is next time: 00:14:50:000

State for model sender1 is packetNum: 3 & totalPacketNum: 5

State for model receiver1 is ackNum: 1

State for model subnet1 is index: 2 & transmitting: 0

State for model subnet2 is index: 2 & transmitting: 0

00:01:12:000

State for model input\_reader is next time: 00:14:50:000

State for model sender1 is packetNum: 3 & totalPacketNum: 5

State for model receiver1 is ackNum: 1

State for model subnet1 is index: 3 & transmitting: 1

State for model subnet2 is index: 2 & transmitting: 0

00:01:15:000

State for model input\_reader is next time: 00:14:50:000

State for model sender1 is packetNum: 3 & totalPacketNum: 5

State for model receiver1 is ackNum: 0

State for model subnet1 is index: 3 & transmitting: 0

State for model subnet2 is index: 2 & transmitting: 0

00:01:25:000

State for model input\_reader is next time: 00:14:50:000

State for model sender1 is packetNum: 3 & totalPacketNum: 5

State for model receiver1 is ackNum: 0

State for model subnet1 is index: 3 & transmitting: 0

State for model subnet2 is index: 3 & transmitting: 1

00:01:28:000

State for model input\_reader is next time: 00:14:50:000

State for model sender1 is packetNum: 3 & totalPacketNum: 5

State for model receiver1 is ackNum: 0

State for model subnet1 is index: 3 & transmitting: 0

State for model subnet2 is index: 3 & transmitting: 0

00:01:28:000

State for model input\_reader is next time: 00:14:50:000

State for model sender1 is packetNum: 4 & totalPacketNum: 5

State for model receiver1 is ackNum: 0

State for model subnet1 is index: 3 & transmitting: 0

State for model subnet2 is index: 3 & transmitting: 0

00:01:38:000

State for model input\_reader is next time: 00:14:50:000

State for model sender1 is packetNum: 4 & totalPacketNum: 5

State for model receiver1 is ackNum: 0

State for model subnet1 is index: 4 & transmitting: 1

State for model subnet2 is index: 3 & transmitting: 0

00:01:41:000

State for model input\_reader is next time: 00:14:50:000

State for model sender1 is packetNum: 4 & totalPacketNum: 5

State for model receiver1 is ackNum: 1

State for model subnet1 is index: 4 & transmitting: 0

State for model subnet2 is index: 3 & transmitting: 0

00:01:51:000

State for model input\_reader is next time: 00:14:50:000

State for model sender1 is packetNum: 4 & totalPacketNum: 5

State for model receiver1 is ackNum: 1

State for model subnet1 is index: 4 & transmitting: 0

State for model subnet2 is index: 4 & transmitting: 1

00:01:54:000

State for model input\_reader is next time: 00:14:50:000

State for model sender1 is packetNum: 4 & totalPacketNum: 5

State for model receiver1 is ackNum: 1

State for model subnet1 is index: 4 & transmitting: 0

State for model subnet2 is index: 4 & transmitting: 0

00:01:54:000

State for model input\_reader is next time: 00:14:50:000

State for model sender1 is packetNum: 5 & totalPacketNum: 5

State for model receiver1 is ackNum: 1

State for model subnet1 is index: 4 & transmitting: 0

State for model subnet2 is index: 4 & transmitting: 0

00:02:04:000

State for model input\_reader is next time: 00:14:50:000

State for model sender1 is packetNum: 5 & totalPacketNum: 5

State for model receiver1 is ackNum: 1

State for model subnet1 is index: 5 & transmitting: 1

State for model subnet2 is index: 4 & transmitting: 0

00:02:07:000

State for model input\_reader is next time: 00:14:50:000

State for model sender1 is packetNum: 5 & totalPacketNum: 5

State for model receiver1 is ackNum: 0

State for model subnet1 is index: 5 & transmitting: 0

State for model subnet2 is index: 4 & transmitting: 0

00:02:17:000

State for model input\_reader is next time: 00:14:50:000

State for model sender1 is packetNum: 5 & totalPacketNum: 5

State for model receiver1 is ackNum: 0

State for model subnet1 is index: 5 & transmitting: 0

State for model subnet2 is index: 5 & transmitting: 1

00:02:20:000

State for model input\_reader is next time: 00:14:50:000

State for model sender1 is packetNum: 5 & totalPacketNum: 5

State for model receiver1 is ackNum: 0

State for model subnet1 is index: 5 & transmitting: 0

State for model subnet2 is index: 5 & transmitting: 0

00:02:20:000

State for model input\_reader is next time: 00:14:50:000

State for model sender1 is packetNum: 5 & totalPacketNum: 5

State for model receiver1 is ackNum: 0

State for model subnet1 is index: 5 & transmitting: 0

State for model subnet2 is index: 5 & transmitting: 0

00:15:00:000

State for model input\_reader is next time: inf

State for model sender1 is packetNum: 1 & totalPacketNum: 3

State for model receiver1 is ackNum: 0

State for model subnet1 is index: 5 & transmitting: 0

State for model subnet2 is index: 5 & transmitting: 0

00:15:10:000

State for model input\_reader is next time: inf

State for model sender1 is packetNum: 1 & totalPacketNum: 3

State for model receiver1 is ackNum: 0

State for model subnet1 is index: 6 & transmitting: 1

State for model subnet2 is index: 5 & transmitting: 0

00:15:13:000

State for model input\_reader is next time: inf

State for model sender1 is packetNum: 1 & totalPacketNum: 3

State for model receiver1 is ackNum: 0

State for model subnet1 is index: 6 & transmitting: 0

State for model subnet2 is index: 5 & transmitting: 0

00:15:23:000

State for model input\_reader is next time: inf

State for model sender1 is packetNum: 1 & totalPacketNum: 3

State for model receiver1 is ackNum: 0

State for model subnet1 is index: 6 & transmitting: 0

State for model subnet2 is index: 6 & transmitting: 1

00:15:26:000

State for model input\_reader is next time: inf

State for model sender1 is packetNum: 1 & totalPacketNum: 3

State for model receiver1 is ackNum: 0

State for model subnet1 is index: 6 & transmitting: 0

State for model subnet2 is index: 6 & transmitting: 0

00:15:26:000

State for model input\_reader is next time: inf

State for model sender1 is packetNum: 2 & totalPacketNum: 3

State for model receiver1 is ackNum: 0

State for model subnet1 is index: 6 & transmitting: 0

State for model subnet2 is index: 6 & transmitting: 0

00:15:36:000

State for model input\_reader is next time: inf

State for model sender1 is packetNum: 2 & totalPacketNum: 3

State for model receiver1 is ackNum: 0

State for model subnet1 is index: 7 & transmitting: 1

State for model subnet2 is index: 6 & transmitting: 0

00:15:39:000

State for model input\_reader is next time: inf

State for model sender1 is packetNum: 2 & totalPacketNum: 3

State for model receiver1 is ackNum: 1

State for model subnet1 is index: 7 & transmitting: 0

State for model subnet2 is index: 6 & transmitting: 0

00:15:49:000

State for model input\_reader is next time: inf

State for model sender1 is packetNum: 2 & totalPacketNum: 3

State for model receiver1 is ackNum: 1

State for model subnet1 is index: 7 & transmitting: 0

State for model subnet2 is index: 7 & transmitting: 1

00:15:52:000

State for model input\_reader is next time: inf

State for model sender1 is packetNum: 2 & totalPacketNum: 3

State for model receiver1 is ackNum: 1

State for model subnet1 is index: 7 & transmitting: 0

State for model subnet2 is index: 7 & transmitting: 0

00:15:52:000

State for model input\_reader is next time: inf

State for model sender1 is packetNum: 3 & totalPacketNum: 3

State for model receiver1 is ackNum: 1

State for model subnet1 is index: 7 & transmitting: 0

State for model subnet2 is index: 7 & transmitting: 0

00:16:02:000

State for model input\_reader is next time: inf

State for model sender1 is packetNum: 3 & totalPacketNum: 3

State for model receiver1 is ackNum: 1

State for model subnet1 is index: 8 & transmitting: 0

State for model subnet2 is index: 7 & transmitting: 0

00:16:22:000

State for model input\_reader is next time: inf

State for model sender1 is packetNum: 3 & totalPacketNum: 3

State for model receiver1 is ackNum: 1

State for model subnet1 is index: 8 & transmitting: 0

State for model subnet2 is index: 7 & transmitting: 0

00:16:32:000

State for model input\_reader is next time: inf

State for model sender1 is packetNum: 3 & totalPacketNum: 3

State for model receiver1 is ackNum: 1

State for model subnet1 is index: 9 & transmitting: 1

State for model subnet2 is index: 7 & transmitting: 0

00:16:35:000

State for model input\_reader is next time: inf

State for model sender1 is packetNum: 3 & totalPacketNum: 3

State for model receiver1 is ackNum: 0

State for model subnet1 is index: 9 & transmitting: 0

State for model subnet2 is index: 7 & transmitting: 0

00:16:45:000

State for model input\_reader is next time: inf

State for model sender1 is packetNum: 3 & totalPacketNum: 3

State for model receiver1 is ackNum: 0

State for model subnet1 is index: 9 & transmitting: 0

State for model subnet2 is index: 8 & transmitting: 1

00:16:48:000

State for model input\_reader is next time: inf

State for model sender1 is packetNum: 3 & totalPacketNum: 3

State for model receiver1 is ackNum: 0

State for model subnet1 is index: 9 & transmitting: 0

State for model subnet2 is index: 8 & transmitting: 0

00:16:48:000

State for model input\_reader is next time: inf

State for model sender1 is packetNum: 3 & totalPacketNum: 3

State for model receiver1 is ackNum: 0

State for model subnet1 is index: 9 & transmitting: 0

State for model subnet2 is index: 8 & transmitting: 0