

Cadmium

A tool for DEVS Modeling and Simulation

User's Guide

DRAFT – 13/10/2020

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

Table of Contents

Cadmium	4
Windows - Installation and example	4
Installing Cygwin, GCC and Boost	4
Downloading and installing the Cadmium Simulator	13
Compiling and Running a Cadmium DEVS Model	14
Ubuntu - Installation and example	17
System requirements	17
Installing Boost	17
Installing g++	18
Installing Git	20
Installing the 'make' command	21
Downloading and installing the Cadmium Simulator	22
Compiling and Running a Cadmium DEVS Model	22
MacOS - Installation and example	25
System requirements	25
Installing Command Line Tools	25
Installing Homebrew and Boost	25
Downloading and installing the Cadmium Simulator	25
Compiling and Running a Cadmium DEVS Model	26
DEVS Model definition: An Example	28
Subnet: an atomic model example implemented in Cadmium	28
Unit testing the Subnet atomic model	35
A Summary on Port Definition	46
Defining the make file to compile the test	48
Simulating the complete ABP model	50
Defining the make file to compile all the test and the ABP	51
Cadmium's Services for Atomic Models	53
Declaring ports	54
Implementing atomic models: a C++ class	54
Using Atomic Models: Creating Instances from the Class	57
Cadmium's Services for Coupled Models	58
Declaring ports	58
Defining coupled models	59
Cadmium's Services to create Logs	62

Cadmium’s Services to Run the Simulation..... 64

Services to Export to Coupled Model to JSON 64

Appendix A 66

Appendix B..... 68

Appendix C..... 70

Appendix D 73

Appendix E..... 77

Appendix F..... 79

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.

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>

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.

Installing Cygwin

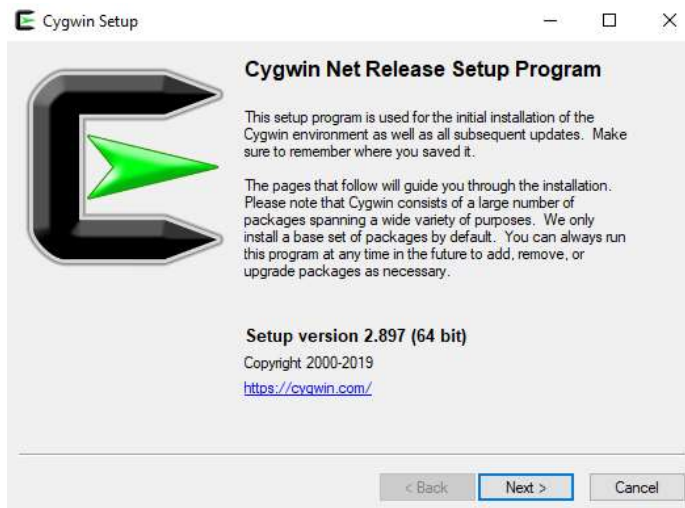
Install Cygwin by running [setup-x86_64.exe](#) (64-bit installation) or [setup-x86.exe](#) (32-bit installation)

Use the setup program to perform a [fresh install](#) or to [update](#) an existing installation.

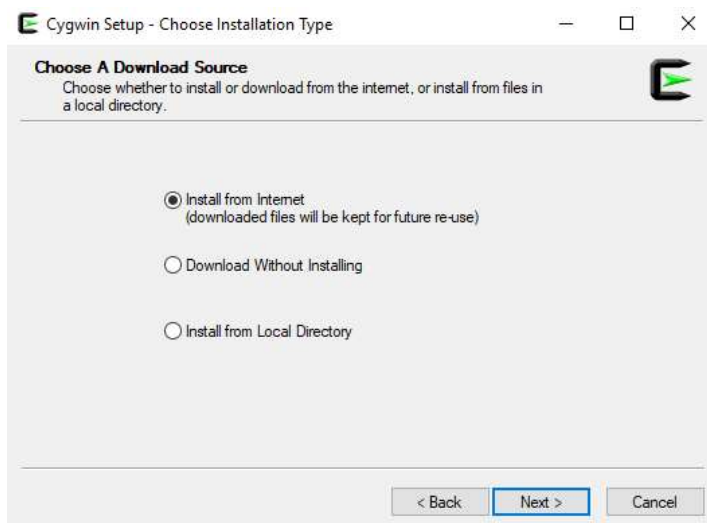
Keep in mind that individual packages in the distribution are updated separately from the DLL so the Cygwin DLL version is not useful as a general Cygwin distribution release number.

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)

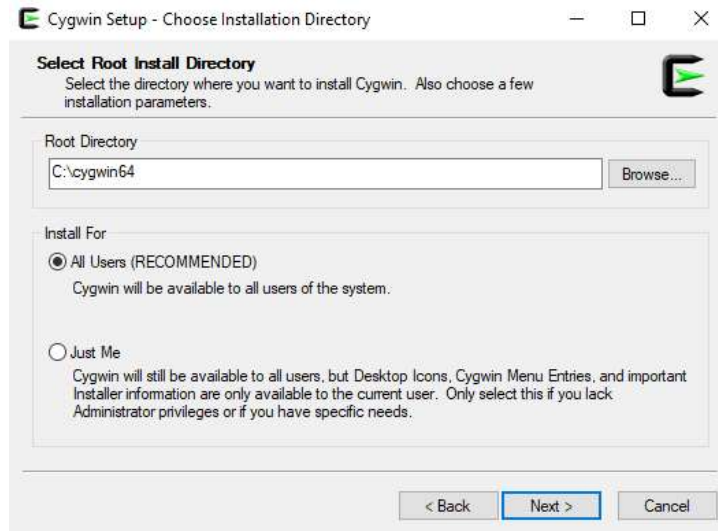
3. Execute setup-x86_64.exe (64-bit installation) or setup-x86.exe (32-bit installation) and click on "Next >". We will see the following welcome screen.



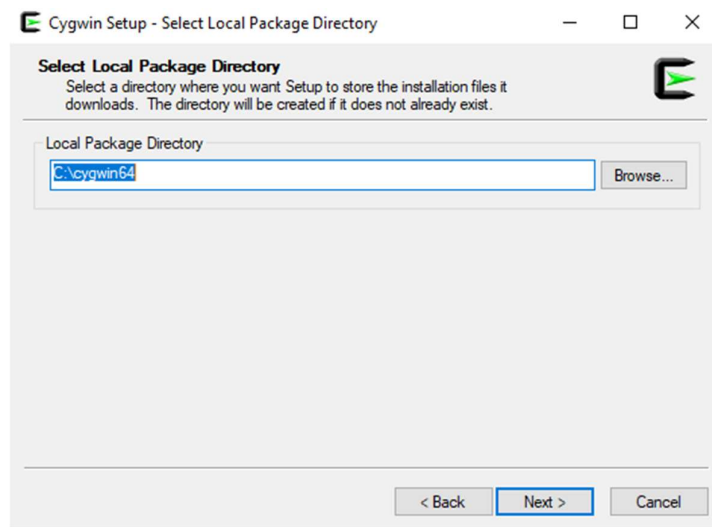
4. Select the option "Install from Internet" and click on "Next >"



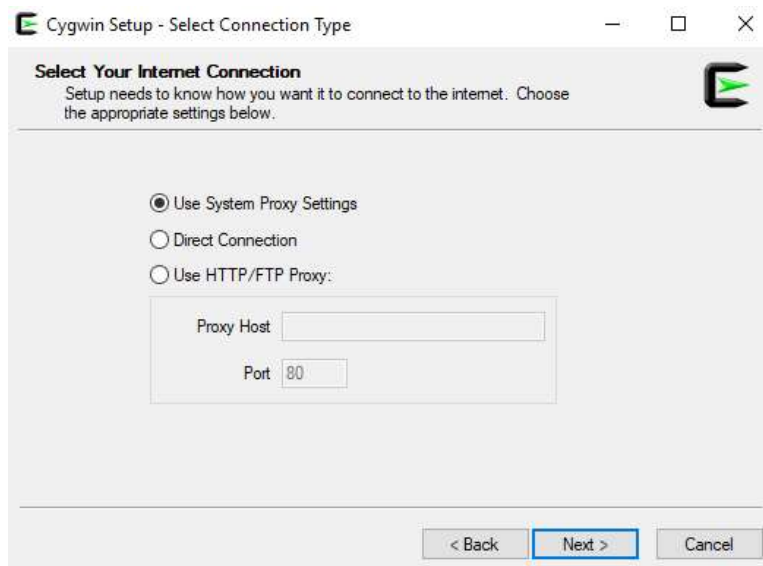
5. We need to select the Root Install directory for storage of Cygwin files. Choose the default (c:\cygwin64, as seen in the screenshot, and "All Users (RECOMMENDED)". Click on "Next >"



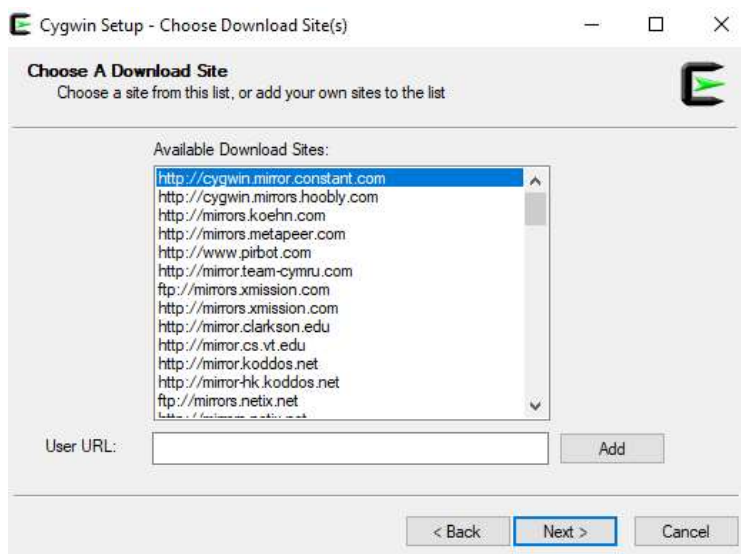
6. Choose your preferred directory for storage of Cygwin local package directory as in the screenshot (i.e. the folder we just created) and click on “Next >”



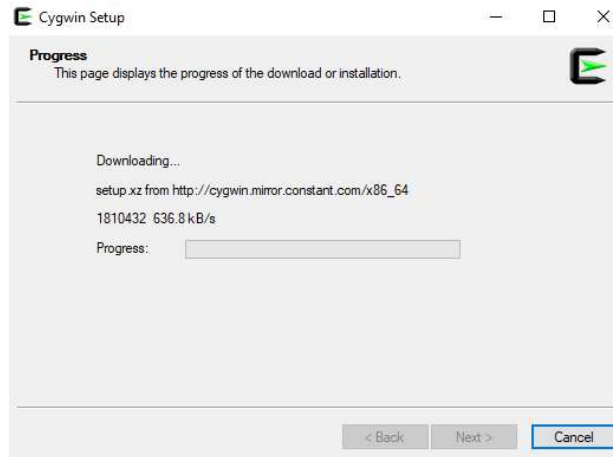
7. Select the option “Use System Proxy Setting” and click on “Next >”



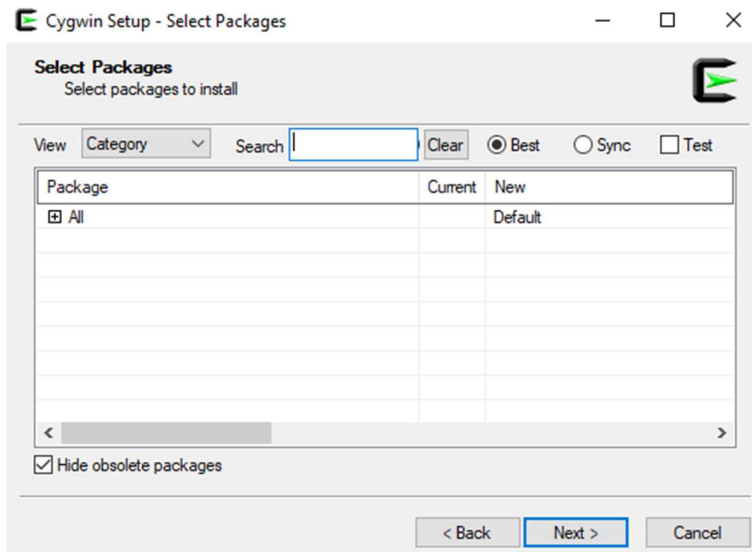
8. After a few seconds, the following window will appear. Choose a Download Site as in the screenshot. Click on "Next >" (in this case, <http://cygwin.mirror.constant.com>)



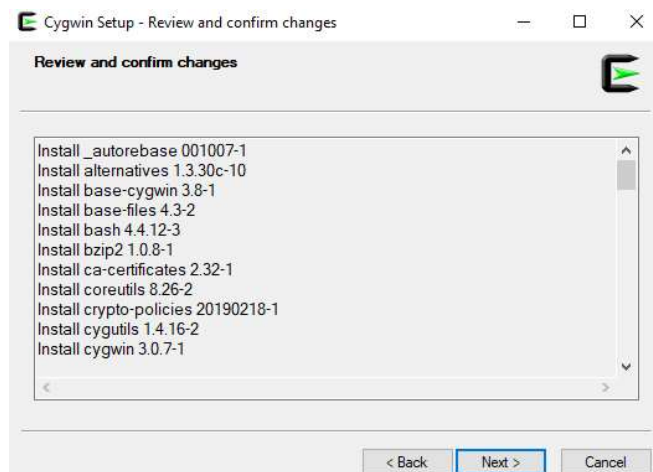
9. Cygwin will start the installation process. The following window will appear



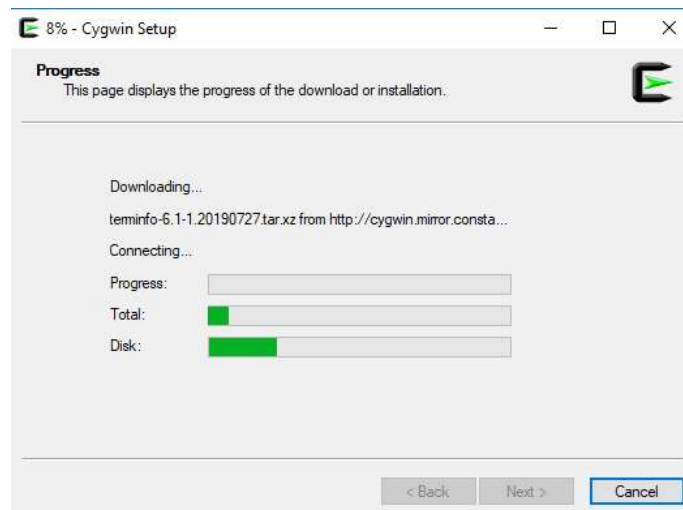
10. When we get the following window, if we click on “All”, we will see all the existing packages. Do not choose anything; simply click “Next >” leaving everything as default (as in the screenshot). This will install the default tools and libraries.



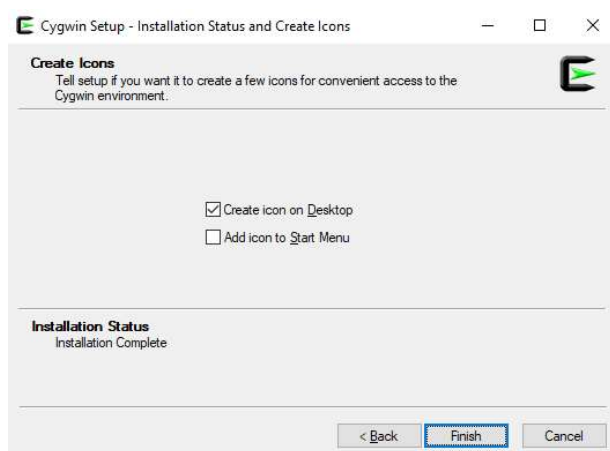
11. The following window will appear. Click on “Next >”



12. The progress window below will appear.



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



14. 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).

This PC > Local Disk (C:) > cygwin64				
Name	Date modified	Type	Size	
bin	8/8/2019 10:57 AM	File folder		
dev	8/8/2019 10:57 AM	File folder		
etc	8/8/2019 10:58 AM	File folder		
home	8/8/2019 10:56 AM	File folder		
lib	8/8/2019 10:57 AM	File folder		
sbin	8/8/2019 10:57 AM	File folder		
tmp	8/8/2019 10:57 AM	File folder		
usr	8/8/2019 10:57 AM	File folder		
var	8/8/2019 10:56 AM	File folder		
Cygwin.bat	8/8/2019 10:57 AM	Windows Batch File	1 KB	
Cygwin.ico	8/8/2019 11:02 AM	Icon	154 KB	
Cygwin-Terminal.ico	8/8/2019 11:02 AM	Icon	53 KB	
setup-x86_64.exe	8/8/2019 10:29 AM	Application	1,197 KB	

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

Type

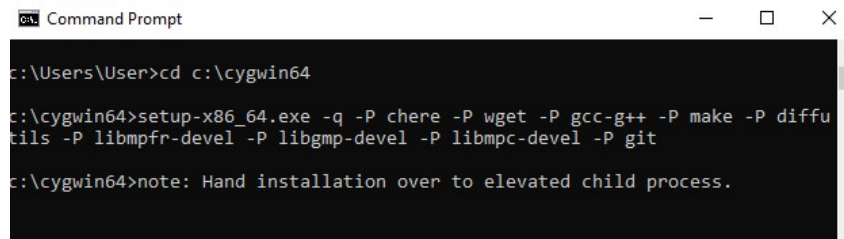
```
cd c:\cygwin64
```

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
```

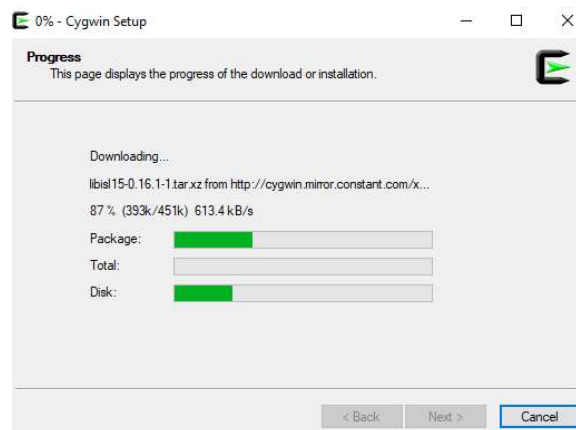
(For 32-bit installation, replace by `setup-x86.exe`)

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



```
Command Prompt  
c:\Users\User>cd c:\cygwin64  
c:\cygwin64>setup-x86_64.exe -q -P chere -P wget -P gcc-g++ -P make -P diffu  
utils -P libmpfr-devel -P libgmp-devel -P libmpc-devel -P git  
c:\cygwin64>note: Hand installation over to elevated child process.
```

16. 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.

17. 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.

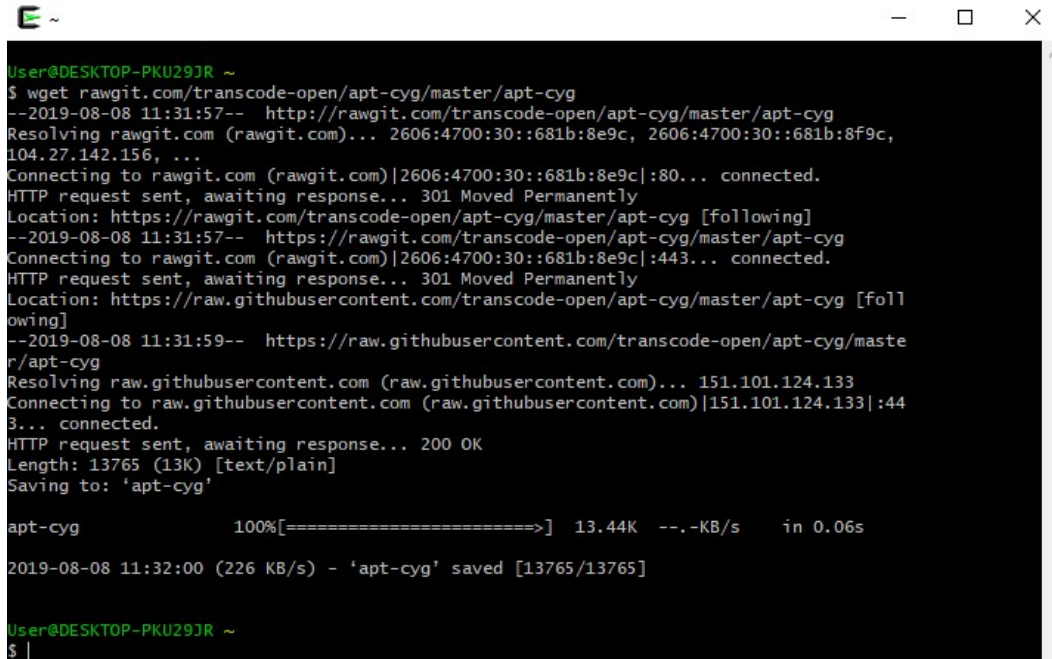
```
'./bashrc' -> '/YOURDIRECTORY/.bashrc'  
'./bash_profile' -> '/YOURDIRECTORY/.bash_profile'  
'./inputrc' -> '/YOURDIRECTORY/.inputrc'  
'./profile' -> '/YOURDIRECTORY/.profile'
```

```
YOURDIRECTORY~
```

```
$
```

18. 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
```



```
User@DESKTOP-PKU29JR ~  
$ wget rawgit.com/transcode-open/apt-cyg/master/apt-cyg  
--2019-08-08 11:31:57-- http://rawgit.com/transcode-open/apt-cyg/master/apt-cyg  
Resolving rawgit.com (rawgit.com)... 2606:4700:30::681b:8e9c, 2606:4700:30::681b:8f9c,  
104.27.142.156, ...  
Connecting to rawgit.com (rawgit.com)|2606:4700:30::681b:8e9c|:80... connected.  
HTTP request sent, awaiting response... 301 Moved Permanently  
Location: https://rawgit.com/transcode-open/apt-cyg/master/apt-cyg [following]  
--2019-08-08 11:31:57-- https://rawgit.com/transcode-open/apt-cyg/master/apt-cyg  
Connecting to rawgit.com (rawgit.com)|2606:4700:30::681b:8e9c|:443... connected.  
HTTP request sent, awaiting response... 301 Moved Permanently  
Location: https://raw.githubusercontent.com/transcode-open/apt-cyg/master/apt-cyg [following]  
--2019-08-08 11:31:59-- https://raw.githubusercontent.com/transcode-open/apt-cyg/master/apt-cyg  
Resolving raw.githubusercontent.com (raw.githubusercontent.com)... 151.101.124.133  
Connecting to raw.githubusercontent.com (raw.githubusercontent.com)|151.101.124.133|:443... connected.  
HTTP request sent, awaiting response... 200 OK  
Length: 13765 (13K) [text/plain]  
Saving to: 'apt-cyg'  
  
apt-cyg          100%[=====] 13.44K  --.-KB/s   in 0.06s  
  
2019-08-08 11:32:00 (226 KB/s) - 'apt-cyg' saved [13765/13765]  
  
User@DESKTOP-PKU29JR ~  
$ |
```

19. Type the following command and press “Enter”
- ```
install apt-cyg /bin
```

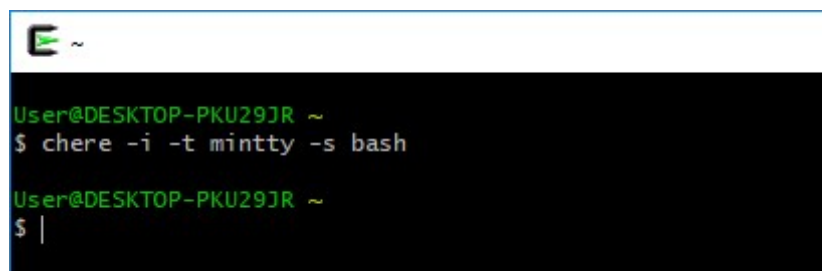


```
User@DESKTOP-PKU29JR ~
$ install apt-cyg /bin
```

"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.

20. 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.



```
User@DESKTOP-PKU29JR ~  
$ chere -i -t mintty -s bash  
  
User@DESKTOP-PKU29JR ~  
$ |
```

21. 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
```

```
User@DESKTOP-PKU29JR ~
$ apt-cyg install libboost-devel
Installing libboost-devel
--2019-08-08 11:49:37-- http://cygwin.mirror.constant.com//x86_64/release/boost/libboost-devel/libb
oost-devel-1.66.0-1.tar.xz
Resolving cygwin.mirror.constant.com (cygwin.mirror.constant.com)... 108.61.5.83
Connecting to cygwin.mirror.constant.com (cygwin.mirror.constant.com)|108.61.5.83|:80... connected.
HTTP request sent, awaiting response... 200 OK
Length: 10953344 (10M) [application/octet-stream]
Saving to: 'libboost-devel-1.66.0-1.tar.xz'

libboost-devel-1.66.0-1. 100%[=====>] 10.45M 629KB/s in 17s

2019-08-08 11:49:55 (625 KB/s) - 'libboost-devel-1.66.0-1.tar.xz' saved [10953344/10953344]

libboost-devel-1.66.0-1.tar.xz: OK
Unpacking...
Package libboost-devel requires the following packages, installing:
libboost_atomic1.63 libboost_atomic1.64 libboost_atomic1.66 libboost_chrono1.63 libboost_chrono1.64
libboost_chrono1.66 libboost_container1.63 libboost_container1.64 libboost_container1.66 libboost_co
ntext1.63 libboost_context1.64 libboost_context1.66 libboost_coroutine1.63 libboost_coroutine1.64 li
bboost_coroutine1.66 libboost_date_time1.63 libboost_date_time1.64 libboost_date_time1.66 libboost_f
iber1.64 libboost_filesystem1.63 libboost_filesystem1.64 libboost_filesystem1.66 libboost_graph1.63
libboost_graph1.64 libboost_graph1.66 libboost_iostreams1.63 libboost_iostreams1.64 libboost_iostrea
```

.....

```
3|:80... connected.
HTTP request sent, awaiting response... 200 OK
Length: 8267736 (7.9M) [application/octet-stream]
Saving to: 'libicu67-67.1-2.tar.xz'

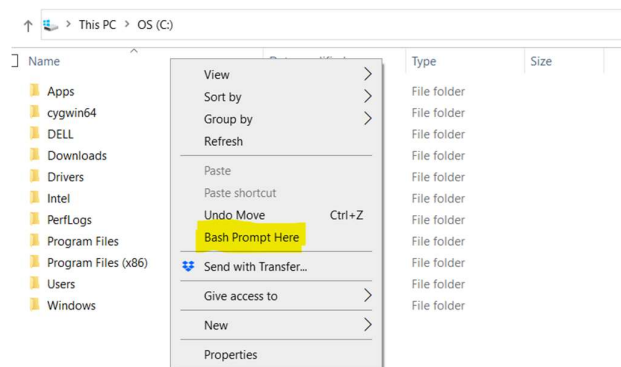
libicu67-67.1-2.tar 100%[=====>] 7.88M 593KB/s in 13s

2020-10-13 16:25:57 (620 KB/s) - 'libicu67-67.1-2.tar.xz' saved [8267736/8267736]

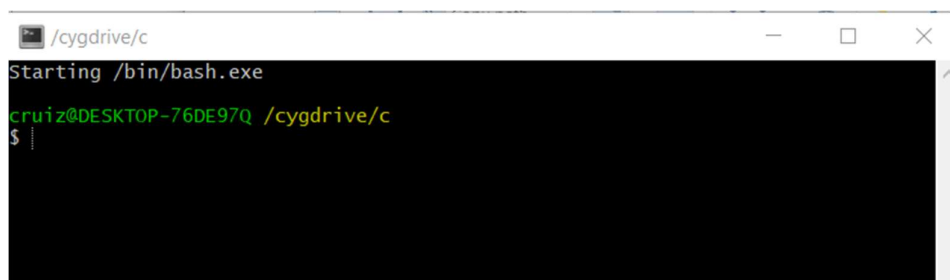
libicu67-67.1-2.tar.xz: OK
Unpacking...
Package libicu67 requires the following packages, installing:
cygwin libgcc1 libstdc++6
Package cygwin is already installed, skipping
Package libgcc1 is already installed, skipping
Package libstdc++6 is already installed, skipping
Package libstdc++6 is already installed, skipping
Package pkg-config is already installed, skipping
Running /etc/postinstall/zp_man-db.sh
Package libboost-devel installed
```

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.



2. 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).


```

/cygdrive/c/Cadmium-Simulation-Environment
Starting /bin/bash.exe

cruiz@DESKTOP-76DE97Q /cygdrive/c
$ git clone https://github.com/SimulationEverywhere/Cadmium-Simulation-Environment.git
Cloning into 'Cadmiu-Simulation-Environment'...
remote: Enumerating objects: 68, done.
remote: Counting objects: 100% (68/68), done.
remote: Compressing objects: 100% (53/53), done.
remote: Total 68 (delta 34), reused 41 (delta 15), pack-reused 0
Unpacking objects: 100% (68/68), 11.65 KiB | 63.00 KiB/s, done.

cruiz@DESKTOP-76DE97Q /cygdrive/c
$ cd Cadmium-Simulation-Environment/

cruiz@DESKTOP-76DE97Q /cygdrive/c/Cadmium-Simulation-Environment
$ git submodule update --init --recursive
Submodule 'CadmiuModelJSONExporter' (https://github.com/SimulationEverywhere/CadmiuModelJSONExporter.git) registered for path 'CadmiuModelJSONExporter'
Submodule 'DESTimes' (https://github.com/SimulationEverywhere/DESTimes.git) registered for path 'DESTimes'
Submodule 'DEVS-Models' (https://github.com/SimulationEverywhere-Models/Cadmiu-DEVS-Models.git) registered for path 'DEVS-Models'
Submodule 'RT-Cadmiu-Models' (https://github.com/SimulationEverywhere/RT-Cadmiu-Models.git) registered for path 'RT-Cadmiu-Models'
Submodule 'cadmiu' (https://github.com/SimulationEverywhere/cadmiu.git) registered for path 'cadmiu'
Cloning into '/cygdrive/c/Cadmium-Simulation-Environment/CadmiuModelJSONExporter'...
Cloning into '/cygdrive/c/Cadmium-Simulation-Environment/DESTimes'...
Cloning into '/cygdrive/c/Cadmium-Simulation-Environment/DEVS-Models'...
Cloning into '/cygdrive/c/Cadmium-Simulation-Environment/RT-Cadmiu-Models'...
Cloning into '/cygdrive/c/Cadmium-Simulation-Environment/cadmiu'...

```

```

Submodule path 'RT-Cadmiu-Models/DISCO-Demo': checked out 'e1b358606b4d5712b0f78b82486908272e5a38c7'
Submodule path 'mbed-os' (https://github.com/ARMmbed/mbed-os.git) registered for path 'RT-Cadmiu-Models/DISCO-Demo/mbed-os'
Cloning into '/cygdrive/c/Cadmium-Simulation-Environment/RT-Cadmiu-Models/DISCO-Demo/mbed-os'...
Submodule path 'RT-Cadmiu-Models/DISCO-Demo/mbed-os': checked out 'e7bc177b20427197763e1cd47261b04bf18db64e'
Submodule path 'RT-Cadmiu-Models/SeedBot': checked out '05c2690bd8b0f3bd251390fc2f51bbc7f5a4f6cc'
Submodule path 'mbed-os' (https://github.com/ARMmbed/mbed-os.git) registered for path 'RT-Cadmiu-Models/SeedBot/mbed-os'
Cloning into '/cygdrive/c/Cadmium-Simulation-Environment/RT-Cadmiu-Models/SeedBot/mbed-os'...
Submodule path 'RT-Cadmiu-Models/SeedBot/mbed-os': checked out '3801d4a1c3aa62a62211faffd24aa4d1e3795974'
Submodule path 'RT-Cadmiu-Models/SeedBot-Light-Follower': checked out '18f00779d76ba249ae5a798045ac2999fd6c75d'
Submodule path 'mbed-os' (https://github.com/ARMmbed/mbed-os.git) registered for path 'RT-Cadmiu-Models/SeedBot-Light-Follower/mbed-os'
Cloning into '/cygdrive/c/Cadmium-Simulation-Environment/RT-Cadmiu-Models/SeedBot-Light-Follower/mbed-os'...
Submodule path 'RT-Cadmiu-Models/SeedBot-Light-Follower/mbed-os': checked out '3801d4a1c3aa62a62211faffd24aa4d1e3795974'
Submodule path 'cadmiu': checked out 'b6636f791d3fbff41b6b72e1d9e34ce18152065d'
Submodule path 'cmake-modules' (https://github.com/bilke/cmake-modules.git) registered for path 'cadmiu/cmake-modules'
Cloning into '/cygdrive/c/Cadmium-Simulation-Environment/cadmiu/cmake-modules'...
Submodule path 'cadmiu/cmake-modules': checked out 'fcfc0494c45fc24fae39996db658b9bdeea4fd8'

cruiz@DESKTOP-76DE97Q /cygdrive/c/Cadmium-Simulation-Environment
$

```

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:

📁 > This PC > OS (C:) > Cadmium-Simulation-Environment

<input type="checkbox"/> Name	Date modified	Type	Size
📁 .git	2020-10-13 4:32 PM	File folder	
📁 cadmiu	2020-10-13 4:35 PM	File folder	
📁 CadmiuModelJSONExporter	2020-10-13 4:35 PM	File folder	
📁 DESTimes	2020-10-13 4:35 PM	File folder	
📁 DEVS-Models	2020-10-13 4:35 PM	File folder	
📁 RT-Cadmiu-Models	2020-10-13 4:37 PM	File folder	
📁 .DS_Store	2020-10-13 4:31 PM	DS_STORE File	7 KB
📁 .gitmodules	2020-10-13 4:31 PM	GITMODULES File	1 KB
📄 README.md	2020-10-13 4:31 PM	MD File	1 KB

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

- Open a Bash Prompt inside the folder Alternating_Bit_Protocol:
Inside the Alternating_Bit_Protocol folder, right-click + "Bash Prompt Here"
- To compile the project and the tests, type in the Bash Prompt:
`make clean; make all`

```
/cygdrive/c/Cadmium-Simulation-Environment/DEVS-Models/Alternating_Bit_Protocol
cruiz@DESKTOP-76DE97Q /cygdrive/c/Cadmium-Simulation-Environment/DEVS-Models
$ cd Alternating_Bit_Protocol/

cruiz@DESKTOP-76DE97Q /cygdrive/c/Cadmium-Simulation-Environment/DEVS-Models/Alternating_Bit_Protocol
$ make clean; make all
rm -f bin/* build/*
g++ -g -c -std=c++17 -I ../../cadmium/include -I ../../DESTimes/include -I ../../CadmiumModelJSONExporter/i
nclude_top_model/main.cpp -o build/main_top.o
g++ -g -c -std=c++17 -I ../../cadmium/include -I ../../DESTimes/include data_structures/message.cpp -o buil
d/message.o
g++ -g -o bin/ABP build/main_top.o build/message.o
g++ -g -c -std=c++17 -I ../../cadmium/include -I ../../DESTimes/include test/main_subnet_test.cpp -o build/
main_subnet_test.o
g++ -g -c -std=c++17 -I ../../cadmium/include -I ../../DESTimes/include test/main_sender_test.cpp -o build/
main_sender_test.o
g++ -g -c -std=c++17 -I ../../cadmium/include -I ../../DESTimes/include test/main_receiver_test.cpp -o buil
d/main_receiver_test.o
g++ -g -o bin/SUBNET_TEST build/main_subnet_test.o build/message.o
g++ -g -o bin/SENDER_TEST build/main_sender_test.o build/message.o
g++ -g -o bin/RECEIVER_TEST build/main_receiver_test.o build/message.o

cruiz@DESKTOP-76DE97Q /cygdrive/c/Cadmium-Simulation-Environment/DEVS-Models/Alternating_Bit_Protocol
$
```

2. Run tests

- A subfolder, called *bin*, has been created. The simulation examples we will execute are in that directory (`cd bin`).
- To run the subnet test, type in the Bash Prompt:
`./SUBNET_TEST.exe`
- To run the receiver test, type in the Bash Prompt:
`./RECEIVER_TEST.exe`
- To run the sender test, type in the Bash Prompt:
`./SENDER_TEST.exe`

```
cruiz@DESKTOP-76DE97Q /cygdrive/c/Cadmium-Simulation-Environment/DEVS-Models/Alternating_Bit_Protocol
$ cd bin

cruiz@DESKTOP-76DE97Q /cygdrive/c/Cadmium-Simulation-Environment/DEVS-Models/Alternating_Bit_Protocol/bin
$ ./SUBNET_TEST.exe

cruiz@DESKTOP-76DE97Q /cygdrive/c/Cadmium-Simulation-Environment/DEVS-Models/Alternating_Bit_Protocol/bin
$ ./RECEIVER_TEST.exe

cruiz@DESKTOP-76DE97Q /cygdrive/c/Cadmium-Simulation-Environment/DEVS-Models/Alternating_Bit_Protocol/bin
$ ./SENDER_TEST.exe

cruiz@DESKTOP-76DE97Q /cygdrive/c/Cadmium-Simulation-Environment/DEVS-Models/Alternating_Bit_Protocol/bin
$
```

- To check the output of the tests, go to the folder `../simulation_results` and open the respective files.

3. Run the top model

- Inside the subfolder *bin*, type
`./ABP.exe ../input_data/input_abp_1.txt`

```
cruiz@DESKTOP-76DE97Q /cygdrive/c/Cadmium-Simulation-Environment/DEVS-Models/Alternating_Bit_Protocol/bin
$ ./ABP.exe ../input_data/input_abp_1.txt
```

- b. 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
 - a. Create new `.txt` files with the same structure as `input_abp_0.txt` or `input_abp_1.txt` in the folder `input_data`
 - b. Run the model using the instructions in step 3
 - c. If we want to keep the output, rename `abp_output.txt`. Otherwise, it will be overwritten when we run the next simulation.

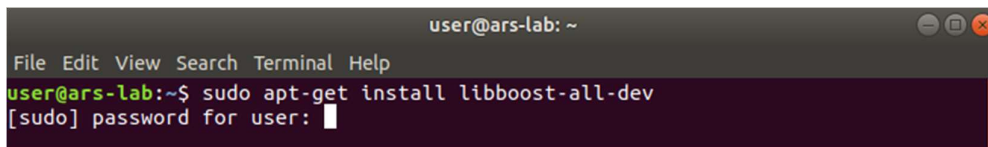
Ubuntu- Installation and example

System requirements

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

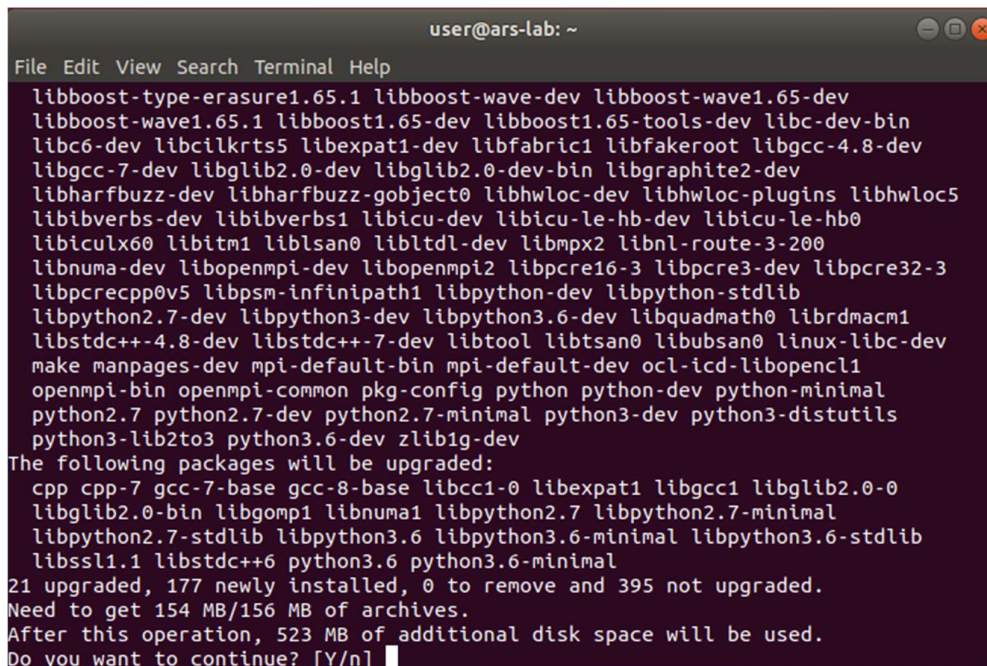
Installing Boost

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.



```
user@ars-lab: ~  
File Edit View Search Terminal Help  
user@ars-lab:~$ sudo apt-get install libboost-all-dev  
[sudo] password for user: 
```

4. 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.



```
user@ars-lab: ~  
File Edit View Search Terminal Help  
libboost-type-erasure1.65.1 libboost-wave-dev libboost-wave1.65-dev  
libboost-wave1.65.1 libboost1.65-dev libboost1.65-tools-dev libcc-dev-bin  
libc6-dev libcilkrts5 libexpat1-dev libfabric1 libfakeroot libgcc-4.8-dev  
libgcc-7-dev libglib2.0-dev libglib2.0-dev-bin libgraphite2-dev  
libharfbuzz-dev libharfbuzz-gobject0 libhwloc-dev libhwloc-plugins libhwloc5  
libibverbs-dev libibverbs1 libicu-dev libicu-le-hb-dev libicu-le-hb0  
libiculx60 libitm1 liblsan0 libltdl-dev libmpx2 libnl-route-3-200  
libnuma-dev libopenmpi-dev libopenmpi2 libpcre16-3 libpcre3-dev libpcre32-3  
libpcrecpp0v5 libpsm-infinipath1 libpython-dev libpython-stdlib  
libpython2.7-dev libpython3-dev libpython3.6-dev libquadmath0 librdmacm1  
libstdc++-4.8-dev libstdc++-7-dev libtool libtsan0 libubsan0 linux-libc-dev  
make manpages-dev mpi-default-bin mpi-default-dev ocl-icd-libopencl1  
openmpi-bin openmpi-common pkg-config python python-dev python-minimal  
python2.7 python2.7-dev python2.7-minimal python3-dev python3-distutils  
python3-lib2to3 python3.6-dev zlib1g-dev  
The following packages will be upgraded:  
  cpp cpp-7 gcc-7-base gcc-8-base libcc1-0 libexpat1 libgcc1 libglib2.0-0  
  libglib2.0-bin libgomp1 libnuma1 libpython2.7 libpython2.7-minimal  
  libpython2.7-stdlib libpython3.6 libpython3.6-minimal libpython3.6-stdlib  
  libssl1.1 libstdc++6 python3.6 python3.6-minimal  
21 upgraded, 177 newly installed, 0 to remove and 395 not upgraded.  
Need to get 154 MB/156 MB of archives.  
After this operation, 523 MB of additional disk space will be used.  
Do you want to continue? [Y/n] 
```

5. Wait until the installation is finished.

```
user@ars-lab: ~  
File Edit View Search Terminal Help  
libgcc-7-dev libglib2.0-dev libglib2.0-dev-bin libgraphite2-dev  
libharfbuzz-dev libharfbuzz-gobject0 libhwloc-dev libhwloc-plugins libhwloc5  
libibverbs-dev libibverbs1 libicu-dev libicu-le-hb-dev libicu-le-hb0  
libcudx60 libitm1 liblsan0 libltdl-dev libmpx2 libnl-route-3-200  
libnuma-dev libopenmpi-dev libopenmpi2 libpcr16-3 libpcr3-dev libpcr32-3  
libpcrcpp0v5 libpsm-infinipath1 libpython-dev libpython-stdlib  
libpython2.7-dev libpython3-dev libpython3.6-dev libquadmath0 librdmacm1  
libstdc++-4.8-dev libstdc++-7-dev libtool libtsan0 libubsan0 linux-libc-dev  
make manpages-dev mpi-default-bin mpi-default-dev ocl-icd-libopencl1  
openmpi-bin openmpi-common pkg-config python python-dev python-minimal  
python2.7 python2.7-dev python2.7-minimal python3-dev python3-distutils  
python3-lib2to3 python3.6-dev zlib1g-dev  
The following packages will be upgraded:  
cpp gcc-7-base gcc-8-base libcc1-0 libexpat1 libgcc1 libglib2.0-0  
libglib2.0-bin libgomp1 libnuma1 libpython2.7 libpython2.7-minimal  
libpython2.7-stdlib libpython3.6 libpython3.6-minimal libpython3.6-stdlib  
libssl1.1 libstdc++6 python3.6 python3.6-minimal  
21 upgraded, 177 newly installed, 0 to remove and 395 not upgraded.  
Need to get 154 MB/156 MB of archives.  
After this operation, 523 MB of additional disk space will be used.
```

....

```
user@ars-lab: ~  
File Edit View Search Terminal Help  
Setting up libboost-graph-parallel1.65-dev (1.65.1+dfsg-0ubuntu5) ...  
Setting up python3-dev (3.6.7-1~18.04) ...  
Setting up libboost-wave1.65-dev:amd64 (1.65.1+dfsg-0ubuntu5) ...  
Setting up libboost-filesystem-dev:amd64 (1.65.1.0ubuntu1) ...  
Setting up libboost-log1.65-dev (1.65.1+dfsg-0ubuntu5) ...  
Setting up libboost-python1.65-dev (1.65.1+dfsg-0ubuntu5) ...  
Setting up libboost-wave-dev:amd64 (1.65.1.0ubuntu1) ...  
Setting up libboost-graph-parallel-dev (1.65.1.0ubuntu1) ...  
Setting up libboost-mpi-dev (1.65.1.0ubuntu1) ...  
Setting up libboost-mpi-python1.65.1 (1.65.1+dfsg-0ubuntu5) ...  
Setting up libboost-python-dev (1.65.1.0ubuntu1) ...  
Setting up libboost-mpi-python1.65-dev (1.65.1+dfsg-0ubuntu5) ...  
Setting up libboost-log-dev (1.65.1.0ubuntu1) ...  
Setting up libboost-mpi-python-dev (1.65.1.0ubuntu1) ...  
Setting up libharfbuzz-dev:amd64 (1.7.2-1ubuntu1) ...  
Setting up libicu-le-hb-dev:amd64 (1.0.3+git161113-4) ...  
Setting up libicu-dev (60.2-3ubuntu3) ...  
Setting up libboost-regex1.65-dev:amd64 (1.65.1+dfsg-0ubuntu5) ...  
Setting up libboost-iostreams1.65-dev:amd64 (1.65.1+dfsg-0ubuntu5) ...  
Setting up libboost-iostreams-dev:amd64 (1.65.1.0ubuntu1) ...  
Setting up libboost-regex-dev:amd64 (1.65.1.0ubuntu1) ...  
Setting up libboost-all-dev (1.65.1.0ubuntu1) ...  
Processing triggers for libc-bin (2.27-3ubuntu1) ...  
user@ars-lab:~$
```

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


```
user@ars-lab: ~  
File Edit View Search Terminal Help  
1 Metadata [7,920 B]  
Reading package lists... Done  
E: The repository 'http://ppa.launchpad.net/jonathonf/gcc-7.1/ubuntu bionic Release' does not have a Release file.  
N: Updating from such a repository can't be done securely, and is therefore disabled by default.  
N: See apt-secure(8) manpage for repository creation and user configuration details.  
user@ars-lab:~$ sudo apt-get update  
Hit:1 http://in.archive.ubuntu.com/ubuntu bionic InRelease  
Ign:2 http://ppa.launchpad.net/jonathonf/gcc-7.1/ubuntu bionic InRelease  
Hit:3 http://in.archive.ubuntu.com/ubuntu bionic-updates InRelease  
Hit:4 http://security.ubuntu.com/ubuntu bionic-security InRelease  
Hit:5 http://in.archive.ubuntu.com/ubuntu bionic-backports InRelease  
Err:6 http://ppa.launchpad.net/jonathonf/gcc-7.1/ubuntu bionic Release  
404 Not Found [IP: 91.189.95.83 80]  
Reading package lists... Done  
E: The repository 'http://ppa.launchpad.net/jonathonf/gcc-7.1/ubuntu bionic Release' does not have a Release file.  
N: Updating from such a repository can't be done securely, and is therefore disabled by default.  
N: See apt-secure(8) manpage for repository creation and user configuration details.  
user@ars-lab:~$
```

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

```
cris@DESKTOP-76DE97Q: ~  
cris@DESKTOP-76DE97Q:~$  
cris@DESKTOP-76DE97Q:~$  
cris@DESKTOP-76DE97Q:~$ sudo apt install build-essential  
Reading package lists... Done  
Building dependency tree  
Reading state information... Done  
The following NEW packages will be installed:  
  build-essential  
0 upgraded, 1 newly installed, 0 to remove and 187 not upgraded.  
Need to get 4624 B of archives.  
After this operation, 20.5 kB of additional disk space will be used.  
Get:1 http://archive.ubuntu.com/ubuntu focal/main amd64 build-essential amd64 12.8ubuntu1 [4624 B]  
Fetched 4624 B in 1s (7141 B/s)  
Selecting previously unselected package build-essential.  
(Reading database ... 57410 files and directories currently installed.)  
Preparing to unpack .../build-essential_12.8ubuntu1_amd64.deb ...  
Unpacking build-essential (12.8ubuntu1) ...  
Setting up build-essential (12.8ubuntu1) ...  
cris@DESKTOP-76DE97Q:~$
```

5. 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`

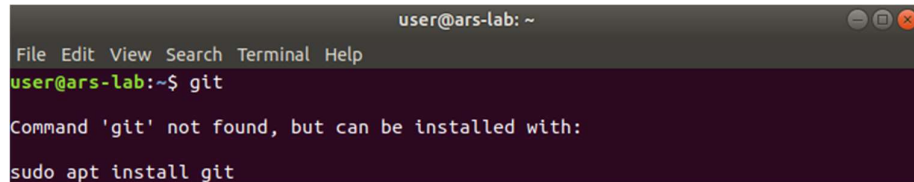
```
cris@DESKTOP-76DE97Q: ~  
cris@DESKTOP-76DE97Q:~$ g++ --version  
g++ (Ubuntu 9.3.0-10ubuntu2) 9.3.0  
Copyright (C) 2019 Free Software Foundation, Inc.  
This is free software; see the source for copying conditions. There is NO  
warranty; not even for MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  
cris@DESKTOP-76DE97Q:~$
```

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:

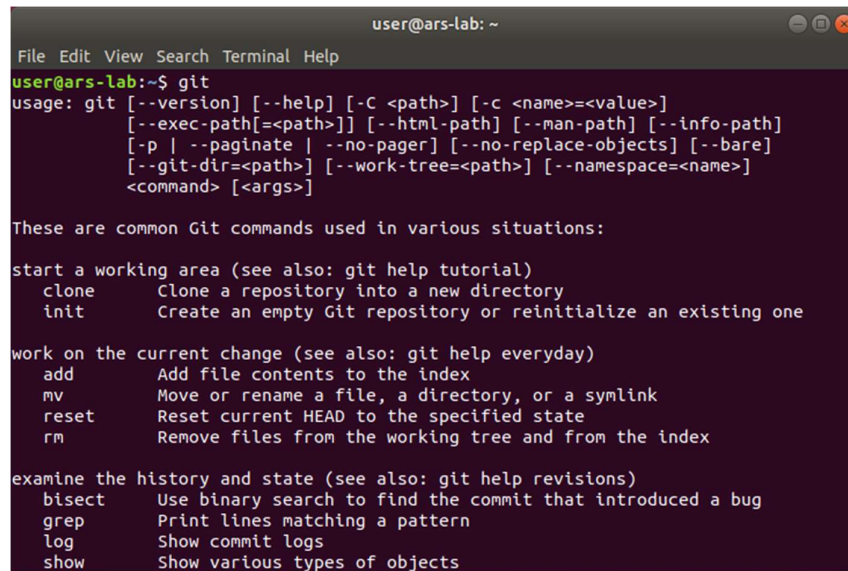
`git`

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



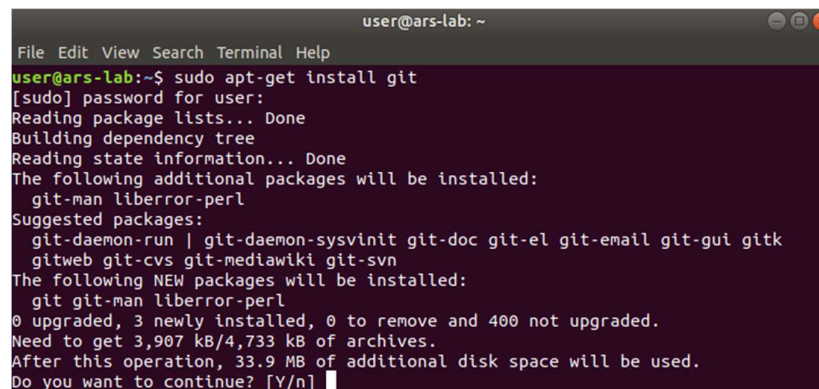
```
user@ars-lab: ~  
File Edit View Search Terminal Help  
user@ars-lab:~$ git  
  
Command 'git' not found, but can be installed with:  
  
sudo apt install git
```

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



```
user@ars-lab: ~  
File Edit View Search Terminal Help  
user@ars-lab:~$ git  
usage: git [--version] [--help] [-C <path>] [-c <name>=<value>]  
       [--exec-path[=<path>]] [--html-path] [--man-path] [--info-path]  
       [-p | --paginate | --no-pager] [--no-replace-objects] [--bare]  
       [--git-dir=<path>] [--work-tree=<path>] [--namespace=<name>]  
       <command> [<args>]  
  
These are common Git commands used in various situations:  
  
start a working area (see also: git help tutorial)  
  clone      Clone a repository into a new directory  
  init       Create an empty Git repository or reinitialize an existing one  
  
work on the current change (see also: git help everyday)  
  add        Add file contents to the index  
  mv         Move or rename a file, a directory, or a symlink  
  reset      Reset current HEAD to the specified state  
  rm         Remove files from the working tree and from the index  
  
examine the history and state (see also: git help revisions)  
  bisect     Use binary search to find the commit that introduced a bug  
  grep       Print lines matching a pattern  
  log        Show commit logs  
  show       Show various types of objects
```

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



```
user@ars-lab: ~  
File Edit View Search Terminal Help  
user@ars-lab:~$ sudo apt-get install git  
[sudo] password for user:  
Reading package lists... Done  
Building dependency tree  
Reading state information... Done  
The following additional packages will be installed:  
  git-man liberror-perl  
Suggested packages:  
  git-daemon-run | git-daemon-sysvinit git-doc git-el git-email git-gui gitk  
  gitweb git-cvs git-mediawiki git-svn  
The following NEW packages will be installed:  
  git git-man liberror-perl  
0 upgraded, 3 newly installed, 0 to remove and 400 not upgraded.  
Need to get 3,907 kB/4,733 kB of archives.  
After this operation, 33.9 MB of additional disk space will be used.  
Do you want to continue? [Y/n]
```

5. 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.

```
user@ars-lab: ~  
File Edit View Search Terminal Help  
user@ars-lab:~$ sudo apt-get install git  
[sudo] password for user:  
Reading package lists... Done  
Building dependency tree  
Reading state information... Done  
The following additional packages will be installed:  
  git-man liberror-perl  
Suggested packages:  
  git-daemon-run | git-daemon-sysvinit git-doc git-el git-email git-gui gitk  
  gitweb git-cvs git-mediawiki git-svn  
The following NEW packages will be installed:  
  git git-man liberror-perl  
0 upgraded, 3 newly installed, 0 to remove and 400 not upgraded.  
Need to get 3,907 kB/4,733 kB of archives.  
After this operation, 33.9 MB of additional disk space will be used.  
Do you want to continue? [Y/n] y  
Get:1 http://in.archive.ubuntu.com/ubuntu bionic-updates/main amd64 git amd64 1:  
2.17.1-1ubuntu0.4 [3,907 kB]  
88% [1 git 3,614 kB/3,907 kB 92%]  
The following NEW packages will be installed:  
  git git-man liberror-perl  
0 upgraded, 3 newly installed, 0 to remove and 400 not upgraded.  
Need to get 3,907 kB/4,733 kB of archives.  
After this operation, 33.9 MB of additional disk space will be used.  
Do you want to continue? [Y/n] y  
Get:1 http://in.archive.ubuntu.com/ubuntu bionic-updates/main amd64 git amd64 1:  
2.17.1-1ubuntu0.4 [3,907 kB]  
Fetched 582 kB in 7s (79.0 kB/s)  
Selecting previously unselected package liberror-perl.  
(Reading database ... 184045 files and directories currently installed.)  
Preparing to unpack .../liberror-perl_0.17025-1_all.deb ...  
Unpacking liberror-perl (0.17025-1) ...  
Selecting previously unselected package git-man.  
Preparing to unpack .../git-man_1%3a2.17.1-1ubuntu0.4_all.deb ...  
Unpacking git-man (1:2.17.1-1ubuntu0.4) ...  
Selecting previously unselected package git.  
Preparing to unpack .../git_1%3a2.17.1-1ubuntu0.4_amd64.deb ...  
Unpacking git (1:2.17.1-1ubuntu0.4) ...  
Setting up git-man (1:2.17.1-1ubuntu0.4) ...  
Setting up liberror-perl (0.17025-1) ...  
Processing triggers for man-db (2.8.3-2ubuntu0.1) ...  
Setting up git (1:2.17.1-1ubuntu0.4) ...  
user@ars-lab:~$
```

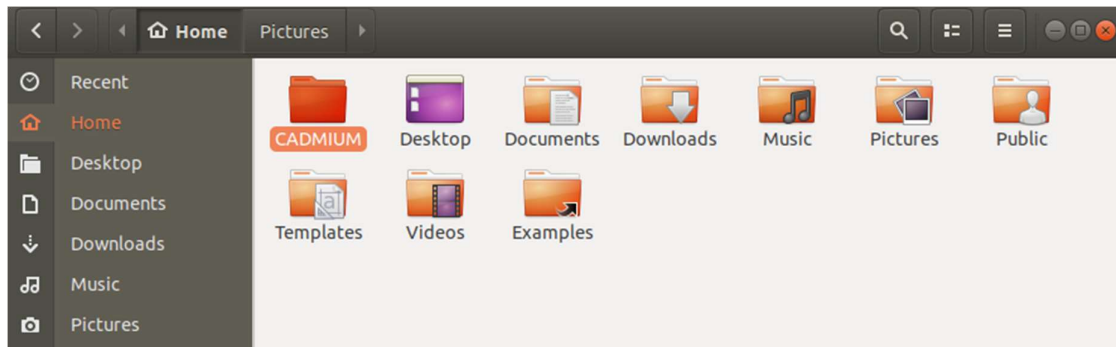
Installing the 'make' command

1. Open the terminal (Press CTRL + Alt + t) and the type following command:
`sudo apt-get install make`
2. Enter the administrative password if we are asked. Wait until the installation is finished

```
user@ars-lab: ~  
File Edit View Search Terminal Help  
user@ars-lab:~$ sudo apt-get install make  
Reading package lists... Done  
Building dependency tree  
Reading state information... Done  
make is already the newest version (4.1-9.1ubuntu1).  
0 upgraded, 0 newly installed, 0 to remove and 398 not upgraded.  
user@ars-lab:~$
```


Downloading and installing the Cadmium Simulator

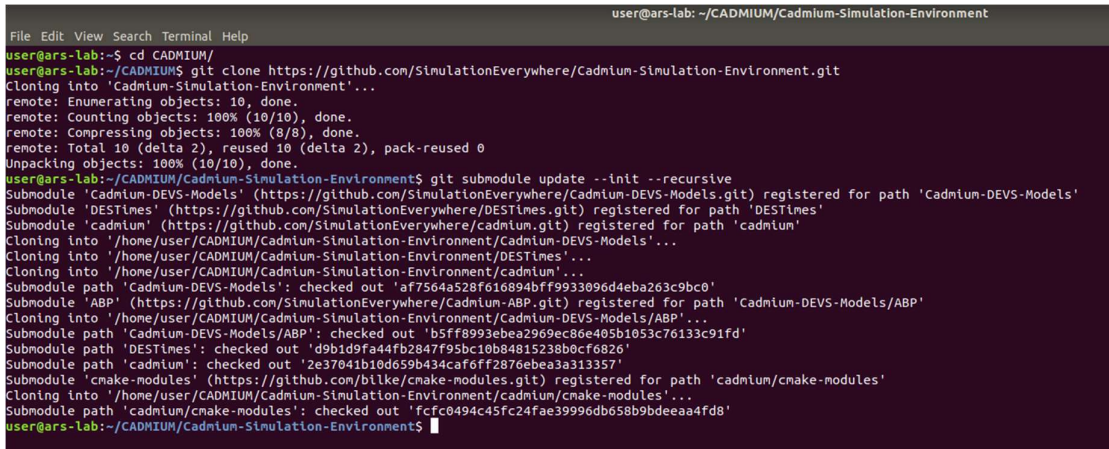
1. Create a new folder in the Home directory and name it as "CADMIUM".



2. 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 terminal window showing the execution of git commands. The user is in the directory ~/CADMIUM/Cadmium-Simulation-Environment. The commands executed are: git clone https://github.com/SimulationEverywhere/Cadmium-Simulation-Environment.git, cd Cadmium-Simulation-Environment, and git submodule update --init --recursive. The output shows the cloning of the repository and the initialization and recursive update of submodules: 'Cadmium-DEVS-Models', 'DESTimes', 'cadmium', 'Cadmium-DEVS-Models/ABP', 'Cadmium-DEVS-Models/ABP', 'DESTimes', 'cadmium', and 'cnake-modules'.

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

```
cris@DESKTOP-76DE97Q:/mnt/c/Cadmium-Simulation-Environment/DEVS-Models/Alternating_Bit_Protocol$ make clean; make all
rm -f bin/* build/*
g++ -g -c -std=c++17 -I ../../cadmium/include -I ../../DESTimes/include -I ../../CadmiumModelJSONExporter/include top_model/main.cpp -o build/main_top.o
g++ -g -c -std=c++17 -I ../../cadmium/include -I ../../DESTimes/include data_structures/message.cpp -o build/message.o
g++ -g -o bin/ABP build/main_top.o build/message.o
g++ -g -c -std=c++17 -I ../../cadmium/include -I ../../DESTimes/include test/main_subnet_test.cpp -o build/main_subnet_test.o
g++ -g -c -std=c++17 -I ../../cadmium/include -I ../../DESTimes/include test/main_sender_test.cpp -o build/main_sender_test.o
g++ -g -c -std=c++17 -I ../../cadmium/include -I ../../DESTimes/include test/main_receiver_test.cpp -o build/main_receiver_test.o
g++ -g -o bin/SUBNET_TEST build/main_subnet_test.o build/message.o
g++ -g -o bin/SENDER_TEST build/main_sender_test.o build/message.o
g++ -g -o bin/RECEIVER_TEST build/main_receiver_test.o build/message.o
cris@DESKTOP-76DE97Q:/mnt/c/Cadmium-Simulation-Environment/DEVS-Models/Alternating_Bit_Protocol$
```

2. 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

```
cris@DESKTOP-76DE97Q:/mnt/c/Cadmium-Simulation-Environment/DEVS-Models/Alternating_Bit_Protocol$ cd bin
cris@DESKTOP-76DE97Q:/mnt/c/Cadmium-Simulation-Environment/DEVS-Models/Alternating_Bit_Protocol/bin$ ./SUBNET_TEST
cris@DESKTOP-76DE97Q:/mnt/c/Cadmium-Simulation-Environment/DEVS-Models/Alternating_Bit_Protocol/bin$ ./RECEIVER_TEST
cris@DESKTOP-76DE97Q:/mnt/c/Cadmium-Simulation-Environment/DEVS-Models/Alternating_Bit_Protocol/bin$ ./SENDER_TEST
cris@DESKTOP-76DE97Q:/mnt/c/Cadmium-Simulation-Environment/DEVS-Models/Alternating_Bit_Protocol/bin$
```

5. 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 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

```
cris@DESKTOP-76DE97Q:/mnt/c/Cadmium-Simulation-Environment/DEVS-Models/Alternating_Bit_Protocol/bin$ ./ABP ../input_data/input_abp_1.txt
cris@DESKTOP-76DE97Q:/mnt/c/Cadmium-Simulation-Environment/DEVS-Models/Alternating_Bit_Protocol/bin$
```

3. 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

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:
 - a. Use the keyboard shortcut "*Command + Space*" to open Spotlight Search.
 - b. Type in "terminal".
 - c. 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).

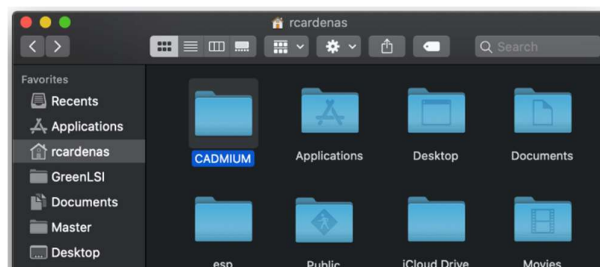
Installing Homebrew and Boost

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

1. Open a terminal:
 - a. Use the keyboard shortcut "*Command + Space*" to open Spotlight Search.
 - b. Type in "terminal".
 - c. 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".



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



```
rcardenas@hyrule:~ > cd CADMIUM/  
rcardenas@hyrule:CADIUM > git clone https://github.com/SimulationEverywhere/Cadmium-Simulation-Environment.git  
Cloning into 'Cadmium-Simulation-Environment'...  
remote: Enumerating objects: 31, done.  
remote: Counting objects: 100% (31/31), done.  
remote: Compressing objects: 100% (26/26), done.  
remote: Total 31 (delta 13), reused 18 (delta 5), pack-reused 0  
Unpacking objects: 100% (31/31), done.  
rcardenas@hyrule:CADIUM > cd Cadmium-Simulation-Environment/  
rcardenas@hyrule:Cadmium-Simulation-Environment > git submodule update --init --recursive  
Submodule 'DESTimes' (https://github.com/SimulationEverywhere/DESTimes.git) registered for path 'DESTimes'  
Submodule 'DEVS-Models' (https://github.com/SimulationEverywhere/Cadmium-DEVS-Models.git) registered for path 'DEVS-Models'  
Submodule 'cadmium' (https://github.com/SimulationEverywhere/cadmium.git) registered for path 'cadmium'  
Cloning into '/Users/rcardenas/CADIUM/Cadmium-Simulation-Environment/DESTimes'...  
Cloning into '/Users/rcardenas/CADIUM/Cadmium-Simulation-Environment/DEVS-Models'...  
Cloning into '/Users/rcardenas/CADIUM/Cadmium-Simulation-Environment/cadmium'...  
Submodule path 'DESTimes': checked out 'd9b1d9fa44fb2847f95bc10b84815238b0cf6826'  
Submodule path 'DEVS-Models': checked out 'f3a29d1adabfed666769f26ba2c227ae3a76ef39'  
Submodule 'ABP' (https://github.com/SimulationEverywhere/Cadmium-ABP.git) registered for path 'DEVS-Models/ABP'  
Cloning into '/Users/rcardenas/CADIUM/Cadmium-Simulation-Environment/DEVS-Models/ABP'...  
Submodule path 'DEVS-Models/ABP': checked out '42b59474135de9b20e86608dacade8b02f001b8b'  
Submodule path 'cadmium': checked out 'b6636f791d3fbff41b6b72e1d9e34ce18152065d'  
Submodule 'cmake-modules' (https://github.com/bilke/cmake-modules.git) registered for path 'cadmium/cmake-modules'  
Cloning into '/Users/rcardenas/CADIUM/Cadmium-Simulation-Environment/cadmium/cmake-modules'...  
Submodule path 'cadmium/cmake-modules': checked out 'fcfc0494c45fc24fae39996db658b9bdeea4fd8'  
rcardenas@hyrule:Cadmium-Simulation-Environment >
```

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:
 - i. Use the keyboard shortcut “*Command + Space*” to open Spotlight Search.
 - ii. Type in “terminal”
 - iii. 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:
 - i. Use the keyboard shortcut "*Command + Space*" to open Spotlight Search.
 - ii. Type in "terminal"
 - iii. 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:
 - i. Use the keyboard shortcut "*Command + Space*" to open Spotlight Search.
 - ii. Type in "terminal".
 - iii. 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.

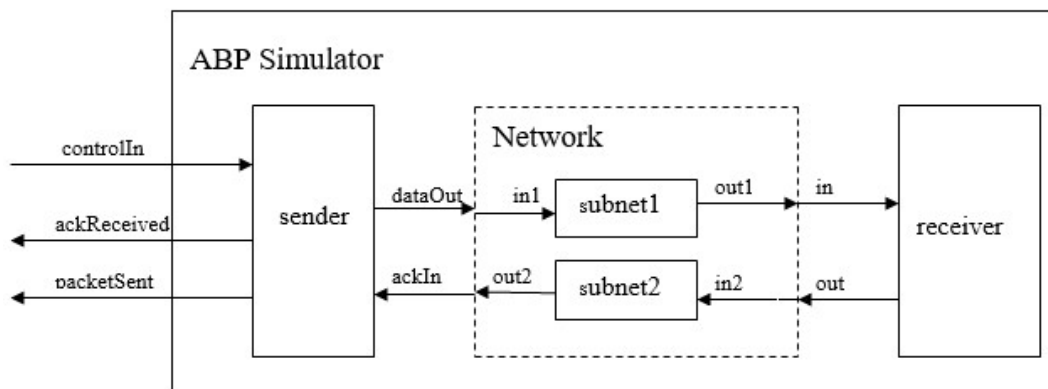


Figure 1 ABP Simulator coupled model

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 ostream& operator<<(ostream& 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: transmitting_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:

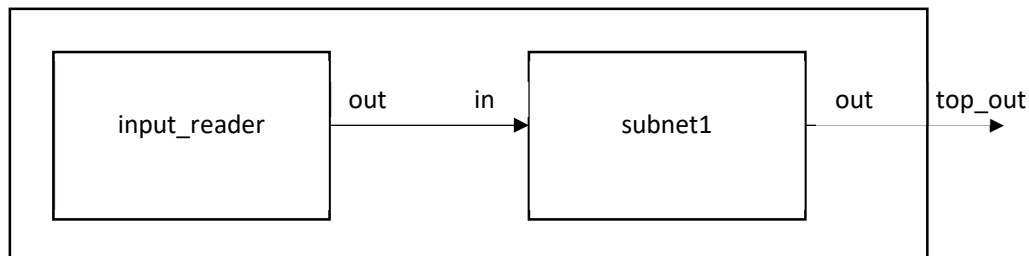


Figure 5. Coupled model for testing the subnet atomic model

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 (`istream.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 EMPTY 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 class `iestream_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 `istream_input` (defined in `<cadmium/basic_model/pdevs/istream.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 `istream_input`. We instantiate `istream_input` with `Message_t` and we leave the time as a template parameter (`istream_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 (`istream_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, `oport 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, `istream_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
[istream_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
[istream_input_defs<Message_t>::out: {15 1}] generated by model input_reader
00:02:40:000
[istream_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.

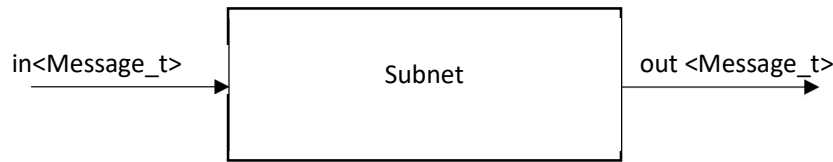


Figure 10. Atomic model Subnet with the ports and the message type on each port

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>::type` would 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 (`istream_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. from – 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” (`istream_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 directory with 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 acknowledgment (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 ostream& operator << (ostream& 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, `inputs_coupled_name`) to store all the input ports as follows:

```
dynamic::modeling::Ports inputs_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 "Cadmiun Model JSON Exporter" automatically generates this JSON file from the implementation of the model in Cadmium. "Cadmiun Model JSON Exporter" is included as a submodule when you download "Cadmiun-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 "Cadmiun 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 first 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.

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 ostream& operator << (ostream& 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 ostream& operator<<(ostream& 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::ostream& operator<<(std::ostream& 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::ostream& operator<<(std::ostream& 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:      {}],
Sender_defs::dataOut: {}] generated by model sender1
00:15:00:000
[istream_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:      {}],
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:      {}],
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
```