Cadmium v2 Developer Manual

By Sasisekhar Govind, Román Cárdenas Rodríguez, Gabriel Wainer – February 2025

Introduction

Cadmium is a tool for Discrete-Event modeling and simulation, based on the DEVS formalism. DEVS is a discrete event system specification that allows a hierarchical and modular description of the models. This document is a developer's guide to Cadmium, and we will only focus on how to build DEVS models. 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

Cadmium is a cross-platform for modeling and simulating DEVS models implemented in C++. It is a header-only library that is used to build DEVS models and execute simulations. The tool can be found at https://devssim.carleton.ca, including an installation manual. It is assumed that developers have installed and tested the tool. This manual focuses on the development of models.

Building DEVS models with Cadmium

This section walks you through the process of building DEVS models using the Cadmium C++ library.

As you are aware, there are two types of DEVS models. Atomic and coupled. Let us first start by looking into how to define each of these models in Cadmium.

Atomic Model Definition

In Cadmium, atomic and coupled models are defined as C++ header files with the extension .hpp.
So, to define an atomic model, you must first create a .hpp file with an appropriate name. Note that throughout the document. Everything inside division operators (/`italics`/) is to be replaced with the appropriate name or value.

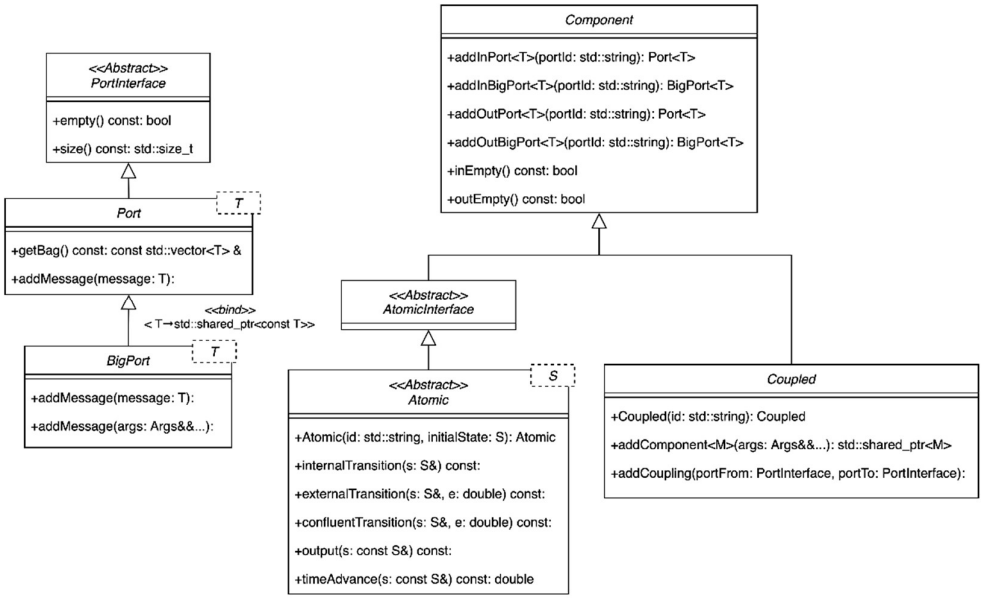


Figure 1: UML Class diagram of Cadmium

Figure 1 shows a UML class diagram of Cadmium, including all you need to develop DEVS models with Cadmium.

Going through the diagram, we see 2 parts. On the right, *Component, AtomicInterface, Atomic,* and *Coupled;* on the left, *PortInterface, Port*, and *BigPort* on the left. *Atomic* and *Coupled* are the two classes users will be directly interfacing with to create models. Users will also be able to handle the methods in *Port* and *BigPort* to deal with the ports of their models. The atomic model class definition has a list of virtual functions that the modeler can override to define the behavior of their model.

These include:

- `void externalTransition(/model name/State& state, double e) const override {}`
  - This method defines the external transition function of the model. It has access to the elapsed time, model input, and it can read and modify the state variables.
- `void output(const /model name/State& state) const override {}`
  - This method defines the output function of the atomic model. It can read state variables and modify the output port.
- `void internalTransition(/model name/State& state) const override {}`
  - This method defines the internal transition function of your model. It can read and modify the state variables of the atomic model.
- `[[nodiscard]] double timeAdvance(const /model name/State& state) const override {}`
  - This method defines the time advance function of your model. It can read state variables and returns the time expiry of that state.
- `void confluentTransition(/model name/State& state, double e) const override {}`
  - This method defines the confluent transition function of the model. It has access to the elapsed time, input, and can read and modify the state variables.

So, to create an atomic model, you must create a class that inherits this *Atomic* class and override these methods to define the behavior of your model.

The coupled model does not require any methods to be overridden, instead it provides two Application Program Interface (API) methods to help create a coupled model. These are:

- `addComponent</model name/>("/model name/")`
  - This method includes DEVS models (atomic or coupled) in a coupled model. Within the angular brackets (<>) it takes the name of a model you have previously defined, and in the parentheses, it takes the name of the model as it should be displayed in the logs. It returns a pointer to the included model.
- `addCoupling(/port from/, /port to/)`
  - This method helps define EIC, EOC, and IC of a coupled model. It takes two arguments, the first being the port from which a message bag arrives and the second being the port to which the message bag should go.

Apart from these, since both *Coupled* and *Atomic* inherit from the *Component* class, they share a few common APIs. These are:

- `addInPort</datatype/>("/name of port/")`
  - This method initializes a DEVS port as an input, it returns a pointer to a variable of type *Port*. Within the angular brackets (<>), it takes the data type of the port, and in the parenthesis, takes the name of the port as it should appear in the logs.
- `addOutPort</datatype/>("/name of port/")`
  - This method initializes a DEVS port as an output, it returns a pointer to a variable of type *Port*. Within the angular brackets (<>), it takes the data type of the port, and in the parenthesis, takes the name of the port as it should appear in the logs.
- `addInBigPort</datatype/>("/name of port/")`
  - This method is the same as *addInPort<>()*, but works on a different type of port in Cadmium called *BigPort* instead of *Port.*
- `addOutBigPort</datatype/>("/name of port/")`
  - This method is the same as *addOutPort<>()*, but works on a different type of port in Cadmium called *BigPort* instead of *Port*.
- `inEmpty()`
  - This method returns a Boolean true if a given input port is empty, else returns false.
- `outEmpty()`
  - This method returns a Boolean true if a given output port is empty, else returns false.

Finally, the *Port* class provides you with APIs that allow you to control the messages in the ports of the model. These APIs are:

- `addMessage(/value/)`
  - This method is part of the *Port* class. It takes */value/* as an input, which is then assigned to an output port.
- `getBag()`
  - This method returns the message bag at a port as an *std::vector<>*.

Conventionally, we name the .hpp file the same name as the model. In our explanations we create an atomic model called *counter*, and so the file name would be `counter.hpp`. The example counter.hpp is included in Appendix I.

The contents of the .hpp file, after including any libraries, and the Cadmium header, the atomic model definition requires you to first define the state set of the atomic model. The state of an atomic model is defined as a C++ structure.

The state structure is defined as such:

```
struct /name of model/State{
         /datatype/ /name of state variable/;
         …
         explicit /name of model/State():/name of state variable/(/initial value/), … {}
};
```

We see that the name of the state structure is */name of model/State*. This is the convention when using Cadmium, be sure to follow it. If your model is names *counter*, then the state structure should be named *counterState*.

Inside the structure, we see the definition of state variables. Here */datatype/* can be any C++ datatype, or it can be a *message structure* (explained later). Then, */name of the state variable/* is to be replaced by the appropriate state variable name. Finally, you have the struct constructor, which is used to initialize the state variables.

Notice the following line at the end of the definition:

```
explicit /name of model/State():/name of state variable/(/initial value/), … {}
```

This is the struct constructor used to initialize the values of the state variable.

Now that we have defined the state struct, the definition of the actual atomic models starts with *cadmium::Atomic<S>*. This is an abstract class in Cadmium which the atomic models will inherit from. A generic atomic model will be defined as:

```
class /atomic model name/: public Atomic</model name/State> {…};
```

This class will now define your atomic model called */atomic model name/* with the state set */model name/State*.

Let us define our ports. In Cadmium, port definitions require 2 steps: declaration, and initialization. Port declaration is done as such:

```
public Port</datatype/> /portName/;
```

Here, */datatype/* can be any C++ datatype or a *message structure*, and /*portName*/ must be a meaningful name. Note that we are using the *public* access specifier. This ensures that our ports are accessible by other models when coupled.

Once the ports have been declared, we move onto the constructor of the atomic model class. This constructor is responsible for calling the constructor of the state structure, and for initializing the ports of the atomic model. Generally, to initialize the port the method to be used is:

```
/name of the port variable/ = addInPort</datatype/>("/portname/");
/name of the port variable/ = addOutPort</datatype/>("/portname/");
```

The *portname* is what will show up in the simulation logs, name it meaningfully.

Now, with this we have defined our state set and defined our input output ports. Per the atomic model tuple, we have defined X, Y and S.

The external transition function (δ_{ext}) is defined as:

```
void externalTransition(/model name/State& state, double e) const override {
        // we can read input messages from a port like this:
        if(!/input port/->empty()){
           for (const auto& x: /input port/ -> getBag()) {
               // x is an input message!
           }
        }
        //your code block goes here
}
```

The *externalTransition* method must be overridden by the user. As its parameter, the method takes an instance of your state variable structure called */model name/State*, and the elapsed time *e*. Inside the method, you see an *if* statement that checks if the input port is empty. This is especially useful if you have multiple ports.

The inputs are in message bags, which are stored as C++ *std::vector<>*. To retrieve this *vector* from the port, Cadmium provides the *getBag*() API. So, `/input port/ -> getBag()` returns the *vector* associated with the input port. To handle individual values from the vector, you see a *for* loop written in C++11's *range based for loop* that allows you to iterate through an entire *vector*. So, for example say you are iterating through a *vector* called myVector, then this:

```
for (const auto& x: myVector) {
  // x is an element of myVector
}
```

Is equivalent to:

```
for (int i = 0; i < myVector.size(); i++) {
  const x = myVector[i];
}
```

This allows to build more readable code.

Your code block to handle your input goes inside the for loop. After this, any modifications you want to make to your state goes outside the if and for code block.

In many cases we just need to obtain one input; this corresponds to the last value in the bag. In those cases, we use the *back*() method of *std::vector*.

Here, we are setting the time advance of the new state to be the time advance of the previous state minus the elapsed time. That way, we maintain the expiry period of the state.

The output function (λ) is defined in Cadmium as:

```
void output(const /model name/State& state) const override {
  // Here, we can add message to output ports.
  /output port/->addMessage(/value/);
}
```

Here, we see that the parameter `/model name/State& state` is preceded by the `const` keyword, essentially barring you from modifying the state. Further, we see the method used to add an output to your output port.

The internal transition function (δ_{int}), it is defined as:

```
void internalTransition(/model name/State& state) const override {
    //your code block goes here
}
```

The time advance function is defined as such:

```
 double timeAdvance(const /model name/State& state) const override {
    return /expression or state variable/;
}
```

A common practice is to define a state variable called *sigma* that you return directly like:

```
double timeAdvance(const MyAtomicState& state) const override {
    return state.sigma;
}
```

And sigma is modified in the internal transition function or external transition function according to the needs of the modeler.

Finally, for the case where we need to handle simultaneous input and internal events, the confluent transition function (δ_{con}) is activated. This function is defined as:

```
void confluentTransition(/model name/State& state, double e) const override {
     //your code block goes here
}
```

Like the internal and external transition functions, you can define the behavior of the model within this function, which will be triggered when and internal and external transition occur simultaneously. In Cadmium, the confluent transition function does also come with a default behavior of:

```
void confluentTransition(MyAtomicState& state, double e) const override {
     externalTransition(internalTransition(state), 0);
}
```

When creating a new model, a good starting point is to use the blank project template found in this repo: https://github.com/Sasisekhar/blank_project_rt

Appendix I defines a complete model named *counter*, and each of the code blocks are explained in detail, including how to define a whole model.

Coupled Model Definition

Like the atomic model definition, the coupled model must also be defined in a .hpp file. The convention remains the same, */model name/.hpp*. In our explanations we will create a coupled model called *top*, and so the file will be called `top.hpp`.

Unlike the atomic model file, the coupled model is not defined in a class. Instead, it is defined in a C++ structure. This struct then inherits from the `cadmium::Coupled` class. To define a generic coupled model, you would do the following:

```
struct /name of the model/: public Coupled {…}
```

Here, */name of the model/* is to be replaced with a meaningful name. Say your model is called *top*, then, the coupled model will be defined as:

```
struct top: public Coupled {…}
```

Within the structure, there are only 2 things to be done: declaration of ports (if any), and the coupled model constructor. The constructor defines the models and the couplings that would be part of this coupled model.

Port declaration and initialization are the same as atomic model. For declaration, you do:

```
Port</datatype/> /portName/;
```

And for initialization you add the following in the constructor:

```
addInPort</datatype/>("/portname/");
addOutPort</datatype/>("/portname/");
```

Before initializing these ports and setting their directions, let us first look at the constructor. The constructor instantiates the models within this coupled model, adds the IC, EIC, and EOC, and initializes the ports. A generic coupled model constructor will look like such:

```
/name of the coupled model/(const std::string& id): Coupled(id){…}
```

Here, */name of the coupled model/* should be replaced appropriately.
NOTE: The name of the constructor **MUST** be the same as the name of the structure.

After initializing and set the directions of the ports, let us move onto model instantiation. In general, this is done as such:

```
auto /instance name/ = addComponent</model name/>("/model name/");
```

auto is a C++ shortcut that allows you to skip writing out the data type. The compiler will infer the data type of /*instance name*/ from what is on the right-hand side of the '=' sign.

The /*instance name/,* sometimes referred to as the *component id* is a unique name given to the model. This id will be used later for coupling the model. The *addComponent<>( )* method allows you to instantiate a DEVS model as part of a coupled model. Within the angular brackets (<>) is */model name/* which should be the name of the model (atomic or coupled) whose instance you are creating within this coupled model Finally, "*/model name/*" should be replaced appropriately as it provides the name for the model which will appear in the simulation logs.

Finally, adding the coupling we use the *addCoupling()* method of the Coupled class. *addCoupling()* takes two parameters: port from, and port to. This is generally defined as:

```
addCoupling(/instance name/->/output port name/, /instance name/ -> /input port name/);
```

Here, */instance name/* must be replaced with the instance whose port is to be coupled. The first parameter must be the port whose output is being coupled to the input port provided by the second parameter. This was the definition of IC. Similarly, for EIC and EOC you would define it as:

```
addCoupling(/input port of coupled model/, /instance name/ -> /input port name/);
addCoupling(/instance name/->/output port name/, /output port of coupled model/);
```

Appendix II defines a complete coupled model used to test the *counter* atomic defined earlier; the Appendix explains each of the code blocks in detail, including how to define a whole coupled model and how to execute it.

Further Appendix III defines a 2-level coupled model extended from the model in Appendix II, portraying the development of hierarchical DEVS models.

Logging in Cadmium

Now that we have looked at how to create models, the following section explains how you can set up the logger for your models

There are four distinct events that trigger logging, these are: external transition function, internal transition function, confluent transition function and output function.

In the case of the three state transition function, the logs are printed depending on how you have overridden the << C++ operator for the state struct of the model. If you observe any of the example models, you will see something similar to the code snippet below:

```
std::ostream& operator<<(std::ostream &out, const /State Struct/& state) {
  out << /However you want to print the state structure/;
  return out;
}
```

This code snippet is the generic format of overriding the << operator in relation to Cadmium. The above code snippet must be written right after the definition of the state struct. The line `std::ostream& operator<<(…)` tells the compiler how the output stream operator (<<) must be overridden to accommodate for the state struct. As parameters, it takes `std::ostream &out` which is the variable that will be the new output stream, and `const /State Struct/& state` which is a variable representing the state structure. Inside the method, you are to write out what elements of the struct you want to print out.

Looking at the *counter* model, we have:

```
std::ostream& operator<<(std::ostream &out, const counterState& state) {
    out << "{count: " << state.count << ", increment: " << state.increment << "}";
    return out;
}
```

This would result in the logs for the state transition logs of this model being printed as:

```
{count: /value of count/, increment: /value of increment/}
```

This is by convention. Following this will allow you to graphically view your output traces using the DEVS trace viewer application. So, the modeler must modify this overloading operator to log relevant state variables.

Unlike logging the state, logging the output function is solely dependent on how the << operator outputs the datatype of the port. The Cadmium loggers directly put the data presented at the port in the *data* column of the log file. However, if you wish to format it, or if using *message structures* (explained later)

you would have to overload the << operator for the datatype of the port in the format you would like the output to be in.

The two loggers native to Cadmium are the *CSVLogger*, and the *STDOUTLogger*. As the name suggests, the *CSVLogger* outputs the simulation log in the form of a Comma Separated Value (CSV) file (more information on how to read the logs can be found in Appendix II). The *STDOUTLogger* outputs the simulation log directly in the terminal, formatted as a CSV. Further, to make the output more readable, the *STDOUTLogger* color codes the logs: all state transitions are colored yellow and the output events are colored green. That way, you can differentiate the two without looking to check if the *port_name* column is missing.

Simulating DEVS models with Cadmium

Once you have defined your models as .hpp files with appropriate formatting on your logger, you can move on to simulating your model. The simulation is done in the *main.cpp* file. Simulating a DEVS model in Cadmium requires 4 main steps:

1. Initialize the model to be simulated
2. Initialize your simulator and connect the model to the simulator
3. Connect the simulator to a logger.
4. Run the simulation

First, we create a pointer of the coupled model you wish to simulate.

```
auto model = std::make_shared</coupled model struct/>("/name of top model/");
```

We use a C++ native method *std::make_shared*<>() to create this pointer. The template parameter (to be passed into <>) is the name of the top coupled model *struct*. Generally, "top model" refers to the coupled model that is highest in the hierarchy, that is, the "top model" is the coupled model that encompasses all the models you wish to simulate.

Next, the parameter(s) passed into the of *make_shared* method is the constructor argument(s) of the top model *struct*. The constructor of your top model would at least take a parameter *std::string id*, and this must be passed as a parameter to the *make_shared* method (this *id* is used by the logger). By convention, we use *model* to represent the pointer to the coupled model under simulation.

NOTE: You can only simulate coupled models. If you want to simulate a single atomic model, put it in a coupled model before simulating it.

Then, we must initialize the *RootCoordinator*, which orchestrates the entire simulation. Cadmium provides multiple root coordinators (*RealTimeRootCoordinator*, *ParallelRootCoordinator*, etc.). Here, we will use the simple *RootCoordinator*, which must be initialized as:

```
auto rootCoordinator = RootCoordinator(/pointer to top model/);
```

By convention, we use *rootCoordinator* to denote the variable pointing to the root coordinator. We then see the constructor being called *RootCoordinator*() whose parameter is the pointer to the coupled model we would like to simulate. In our case, this pointer is *model*, so:

```
auto rootCoordinator = RootCoordinator(model);
```

Then we need to initialize a *logger*, which defines how the simulation must record the status of the model. Cadmium logs state changes (δ_{int}, δ_{ext}, and δ_{con}), and outputs (λ).

```
rootCoordinator.setLogger</preferred logger/ >(/constructor parameters of logger/);
```

where *rootCoordinator* is an object of the *RootCoordinator* class. *setLogger* is a method that connects a *logger* to the object. */preferred logger/* is to be replaced with either *STDOUTLogger* or *CSVLogger*. *STDOUTLogger* logs data onto the terminal, and *CSVLogger* logs data into a CSV file.

To use *STDOUTLogger*, you would do:

```
rootCoordinator.setLogger<STDOUTLogger>(";");
```

Here ";" is the separator between the various components of the log. To use the *CSVLogger*, you would do:

```
rootCoordinator.setLogger<CSVLogger>("log_output.csv", ";");
```

Here, *"log_output.csv"* is the file name of the log output, and ";" is the separator.

Finally, to 'run the simulation' there are three sub steps: initialize the simulation, run the simulation, and terminate the simulation cleanly (C++ memory cleaning etc.).

We first call the *start*() method of *RootCoordinator*.

```
rootCoordinator.start();
```

This initializes the models, initializing the state variables with the provided default values. Then, we simulate the model for a specified time:

```
rootCoordinator.simulate(/simulation time in seconds/);
```

NOTE: The *simulate*() method expects a *double* as its parameter. So be sure to add the .0 at the end if you want to simulate for a whole number amount of time. Also note that this time value cannot be negative.

Finally, to clean up the simulation, *RootCoordinator* has a method called *stop*(), which is to be called as such:

```
rootCoordinator.stop();
```

A complete *main.cpp* file can be found in Appendix IV; the code is explained in detail.

Message Structures

Sometimes, when defining atomic models, you may find that the simple C++ datatypes may be lacking. You might want to use a complex data structure to more accurately represent the state variables or messages. This is where the use of message structures comes in. The idea of a message structure is simple: create a data type that fits your needs and tell the compiler how to print the datatype. This file is also defined as a .hpp file.

The generic definition of a message structure is this:

```
struct /message structure name/ {…};

std::ostream& operator<<(std::ostream& out, const /message structure name/& m){…}
```

Let us first look at the structure itself. This struct does not inherit from any class, it is a simple struct with name */message structure name/*. Within this struct you may define your variables, of any datatype, including the C++ STL data types like std::unordered_map<> or std::vector<>. You must also define a constructor to initialize the values of the message structure. Say you want a message structure to represent RGB values:

```
struct RGB {
  uint8_t R;
  uint8_t G;
  uint8_t B;
  RGB(): R(0), G(0), B(0) {};
```

```
};
```

Here we define each variable to be of type unsigned 8-bit integer (*uint8_t*) and initialize all of them to be 0.

Now, to inform the compiler how to print the structure (this is similar to printing the state structure), the parameter *out* must be fed the values you are interested in. For instance, to print the RGB structure, one might do (note that you are simply overloading the << operator):

```
std::ostream& operator<<(std::ostream& out, const RGB& m){
     out << "{" << m.R << ", " << m.G << ", " << m.B << "}";
  return out;
}
```

The above operator overload would make the output be printed as: *{0, 0, 0}* for black or *{255, 255, 255}* for full white, etc. Overloading "<<" is compulsory, however, you may overload other operators if convenient or essential for your use case.

Putting it all together, we get *RGB.hpp* to be:

```
#include<iostream>

struct RGB {
  uint8_t R;
  uint8_t G;
  uint8_t B;
  RGB(): R(0), G(0), B(0) {};
};

std::ostream& operator<<(std::ostream& out, const RGB& m){
  out << "{" << m.R << ", " << m.G << ", " << m.B << "}";
  return out;
}
```

Build configurations of Cadmium Projects

A Cadmium project would generally follow the following structure:

```
.
├── build_sim.sh              //script file to build the models for simulation
├── CMakeLists.txt            //CMake configuration for the project
└── main                      //main folder containing the code
    ├── CMakeLists.txt        //CMake configuration for the source code
    ├── include               //include folder containing all the models
    │   ├── coupled_model.hpp //your coupled model files
    │   │       ⋮
    │   └── atomic_model.hpp  //your atomic model files
    └── main.cpp              //main file to simulate the model
```

The `build_sim.sh` script file creates the *build* and *bin* directories and initiates the build system to start building the model files. The intermediary build files are stored in the *build* directory and the binaries will be created in the *bin* directory. This script also cleans stale builds and starts afresh. Apart from this, some projects will come with `build_rt.sh` and `build_esp.sh` which builds the models for real-time simulation and embedded execution respectively.

The *CMakeLists.txt* is the top level CMake configuration file. Projects in Cadmium make use of the CMake build system and is completely automated. For more information about the specifics of CMake, go here. Running the *cmake* command with the appropriate *CMakeLists.txt* files generate the MakeFiles for building the project, or whatever is appropriate for the platform you are on. This enables Cadmium to be platform independent.

Then, the *main* directory is where all the code lives. Within this, we define another `CMakeLists.txt file`. The build system follows a hierarchical structure, with one top level *CMakeLists.txt* file for the project configuration in the parent directory, and one *CMakeLists.txt* file within this *main* directory to set the source code and executable files. As a model developer, if you are using the *blank_project_rt* template to start your project, there is only one line in the CMakeList files that you would have to change to personalize the project. In the top level CMakeList file, you change the following line:

```
set(projectName "sample_project")
```

and replace "*sample_project*" with whatever name is most appropriate for your project. This will be the name of the binary executable that your code builds into. Everything else will build automatically.

Further, the *include* directory is where all the coupled model and atomic model files are stored. If you have additional libraries/ header files that you want to include in your project, they will also go in here.

NOTE: If you have a library with a *.cpp/ .c* file, you must define a *CMakeLists.txt* file for that library to add the source files.

Finally, in the *main* directory is the *main.cpp* file that runs the simulation.

Appendix I – Counter model

The code below defines an atomic model called *counter*. This *counter* atomic model uses 2 inputs called *direction_in* and *increment_in*, and 1 output called *count_out*. The *counter* atomic model increments a state variable called *count* per a user defined increment value received through the *increment_in* port. The user can also define the direction of counting (count up or count down) using the *direction_in* input port. The output of the counter is presented at the *count_out* port.

The code for counter, which can be found on https://github.com/Sasisekhar/DEVS_manual_example is given below, and the explanation for each code block can be found after the code.

```
#include <iostream>
#include "cadmium/modeling/devs/atomic.hpp" //Cadmium header

using namespace cadmium;         //This tells the compiler we are using with Cadmium

struct counterState {            // State Set definition
    int count;
    int increment;
    bool countUp;
    double sigma;

    explicit counterState (): count(0), increment(1), countUp (true), sigma(1.0){}
};

//This part is required to tell cadmium how to print out your state set. As you can see,
//we are asking cadmium to print {count: /count value/, increment: /increment value/}
std::ostream& operator<<(std::ostream &out, const counterState& state) {
    out  << "{count: " << state.count << ", increment: " << state.increment << "}";
    return out;
}

class counter: public Atomic<counterState> {             // counter Atomic Model definition

 public:                                                 // Input/Output port declaration
    Port<bool> direction_in; //Input to set the direction of count
    Port<int> increment_in;  //Input to set the increment value
    Port<int> count_out;     //Outputs the count value

    counter(const std::string id) : Atomic<counterState>(id, counterState()) {  // Initialize ports
      count_out = addOutPort<int>("count_out"); //Set it to output port
      direction_in = addInPort<bool>("direction_in"); //Set it to input port
      increment_in = addInPort<int>("increment_in"); //Set it to input port
    }

    // external transition: receive direction (inc/dec) and increment value
    void externalTransition(counterState& state, double e) const override {
        if(!direction_in->empty()){
          state.countUp = direction_in->getBag().back(); //We only need the last value
        }
        if(!increment_in -> empty()) {
          state.increment = increment_in->getBag().back();
        }
        state.sigma -= e;
    }

    // output function: transmit the output value
    void output(const counterState& state) const override {
        count_out->addMessage(state.count);
    }

    //Every ta(s) we increment or decrement count by the magnitude of state.increment.
    void internalTransition(counterState& state) const override {
      if(state.countUp) {
        state.count += state.increment;
      } else {
        state.count -= state.increment;
      }
      state.sigma = 1.0;
    }
```

```
    // time_advance function
    [[nodiscard]] double timeAdvance(const counterState& state) const override {
            return state.sigma;
    }
};
```

Starting from the top, the `#include` directive must be used to include the appropriate Cadmium APIs required for your model (and later, simulation). In this case we see *iostream* which is required for the "<<" operator, and *cadmium/modeling/devs/atomic.hpp* which contains all the Cadmium APIs for atomic models.

Then, `using namespace cadmium` tells the compiler that the APIs we are using are indeed that of cadmium. This helps in reducing ambiguity especially if multiple libraries were involved.

Next we see the declaration of the state set called *counterState*, with 4 state variables: an integer *count*, an integer *increment*, an Boolean *countUp*, and a double *sigma,* and the struct constructor used to initialize the values of the state variables. Here, we see *count* is initialized to 0, *increment* to 1, *countUp* to *true* and *sigma* to 1.0.

After, we define the atomic model *counter*, whose state is represented by the *counterState* structure (mentioned above). This is defined as:

```
class counter: public Atomic<counterState> {…};
```

Now that we have defined our class, let us see how to define the components of the atomic model (keep in mind that we are defining, in code, the Atomic tuple <X, Y, S, δ_{int}, δ_{ext}, δ_{con}, λ, ta>). First, we define the three ports are defined in the model: *direction_in,* a Boolean input port; *increment_in,* an integer input port, and *count_out* , an integer output port. After this, we need to initialize the ports, using the which we see being done in the constructor:

```
counter(const std::string id) : Atomic<counterState>(id, counterState()) {
  count_out = addOutPort<int>("count_out"); //Set it to output port
  direction_in = addInPort<bool>("direction_in"); //Set it to input port
  increment_in = addInPort<int>("increment_in"); //Set it to input port
}
```

NOTE: Always remember to initialize your ports. Double-check that the ports have been properly initialized and that their direction (input or output) is set correctly. While Cadmium provides error handling, multiple errors in your code can sometimes cause Cadmium (or g++) to fail to report issues accurately. As a result, it may not always be immediately apparent that port initialization was missed.

Next, we define the *externalTransition* function, which has access to the state variables in *counterState* (which is out state set) and elapsed time *e*. Implicitly, since the ports are defined as attributes of the atomic model class, the *externalTransition* function also has access to the inputs. Earlier, we saw the use of C++11's *range based for loop* to access elements of the input port bag. However, in many cases we just need to obtain one input; this corresponds to the last value in the bag. In those cases, we use the *back*() method of *std::vector*. In this case, *Direction_in -> getBag*() returns the message bag at the input port as an *std::vector<>*. Then, the *back*() method retrieves the last element from the *vector*. Therefore, in essence, we are assigning the latest value at the port to the state variable *countUp*. Similarly, in the next *if* statement, we are assigning the latest value of *increment_in* to the state variable *increment*.

Finally, apart from storing the input we do not want ta(s) to be reset when an external input arrives. To prevent this from happening, we do:

```
    state.sigma -= e;
```

Here, *sigma* is a state variable we use to keep track of the time advance for each state.

Then, we define the output function, which uses the state set *counterState*. The *const* keyword is to enforce the fact that state variables cannot be modified in the output function, only read. Within the function, we see the value of the state variable *count* be assigned to the port *count_out* using the *addMessage* method.

NOTE: The value passed into *addMessage()* must be of the same datatype as the port it is being sent to.

We then see the internal transition function, which accesses to the state set *counterState*. The value of the state variable *count* is either incremented or decremented by the value of the state variable *increment,* depending on if the value of state variable *countUp* is true or false. At the end, we restore the value of state variable *sigma* and schedule the next internal event in 1 time unit.

Finally, the time advance function simply returns the state variable *sigma*, which was modified in both the state transition functions.

Appendix – II – Counter coupled model

The code below defines a coupled model, called *counter_tester,* which tests the *counter* atomic model. To test the model, we need some way to inject input testcases into our atomic model. There are a multiple ways of doing this in Cadmium: *IEStream*, *generator*, and so on. In this example, we will use *IEStream,* a predefined atomic model in Cadmium, which produces output events based on a text file. The text files must contain the time of the event and the value of its output, separated by space. So:

```
/time instance of first event/ /value/
/time instance of next/ /value/
...
```

These models are especially useful for quick small-scale testing, since you do not have to re-build the model every time you change your input testcases. However, as your models increase in complexity, so will testcases. Handling testcases would then involve handling multiple text files, each containing 100s of testcases would quickly get out of hand. To solve this issue, the modeler can create *generators* that can procedurally generate testcases for your model. If you are interested in generators, check out the Generator-Processor-Transducer (GPT) example. (https://github.com/Sasisekhar/GPT_Example and: https://github.com/SimulationEverywhere/cadmium_v2/wiki/5.-Examples-of-DEVS-Models).

Continuing with *IEStream*, the following diagram illustrates the coupled model we are trying to create:

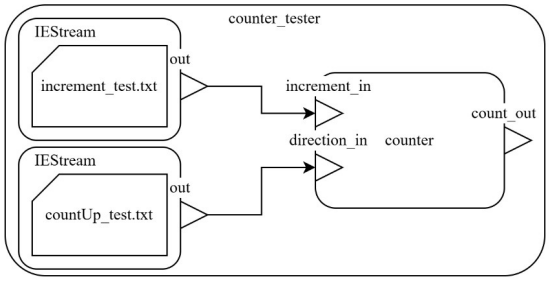


Figure 2: counter_tester coupled model

Figure 2 shows the three models and the couplings that make up the *counter_tester* coupled model. We see the *counter* atomic model with its two inputs *direction_in* and *increment_in*, and its output *count_out*. We also see the two *IEStream* models with output port *out*. The test files that the *IEStream*s take are *increment_test* and *countUp_test*, which test the *increment_in* and *direction_in* inputs of *counter*.

The code to define this model, found at in https://github.com/Sasisekhar/DEVS_manual_example is:

```
#include "cadmium/modeling/devs/coupled.hpp" //cadmium header
#include "cadmium/lib/iestream.hpp"          //iestream header
#include "counter.hpp"                       //counter atomic model file

using namespace cadmium;

struct counter_tester: public Coupled {

  counter_tester(const std::string& id): Coupled(id){
    auto counter_model   = addComponent<counter>("counter model");
    auto increment_file  = addComponent<lib::IEStream<int>>("increment file", "/absolute filepath/");
    auto countUp_file    = addComponent<lib::IEStream<bool>>("countUp file", "/absolute filepath/");

    //Internal Couplings
    addCoupling(increment_file->out, counter_model->increment_in);
    addCoupling(countUp_file->out, counter_model->direction_in);
  }
```

```
};
```

Starting from the top, we first include the appropriate Cadmium APIs required for your model (and later, simulation). In this case we see *cadmium/modeling/devs/coupled.hpp* which contains all the Cadmium APIs for coupled models, *cadmium/lib/iestream.hpp* which contains the *IEStream* atomic model, and *counter.hpp* which includes the *counter* atomic model.

Then, `using namespace cadmium` tells the compiler that the APIs we are using are indeed that of cadmium. This helps in reducing ambiguity especially if multiple libraries were involved.

Next we define the coupled model struct *counter_tester*, which inherits from the *Coupled* class.

Then, using the *addComponent* method, we add the three atomic models that are part of the *counter_tester* coupled model. As the constructor parameter, for *counter* as its constructor argument, *"counter model"* is passed. This would be the name of the model that shows up in the logs. Similarly, for the two *IEStreams*, we see the id *"increment file"* and *"countUp file"* being passed as names for the logs. However, *IEStream* has a second constructor argument, and that is the absolute file path to the input testcase file.

Finally, we use the *addCoupling*<>() method to define the internal couplings of the coupled model.

Running the model

To run the model, clone this repository: https://github.com/Sasisekhar/DEVS_manual_example. The repository contains multiple models, we are solely interested in the *counter_tester* for this section.

The text files for *IEStream* can be found in the parent directory of the repository, and they are named *increment_test.txt* and *countUp_test.txt*. Their contents by default are this:

- increment_test.txt

```
1.1 2
7.5  9
```

- countUp_test.txt

```
1.1 0

2.3 1
```

The test run in this section is based on the above values in the text files. The absolute path to these files must be provided to the constructor of *IEStream* in the *counter_tester.hpp* file. So, in this case, the *IEStream* models will be instantiated as:

```
auto increment_file = addComponent<lib::IEStream<int>>("increment file",
              "/your-filepath-here/ /DEVS_manual_example/increment_test.txt");

auto countUp_file = addComponent<lib::IEStream<bool>>("countUp file",
              "/your-filepath-here/ /DEVS_manual_example/countUp_test.txt");
```

Once you have defined the paths of your files, build the models using:

```
source build_sim.sh
```

and then run the *counter_tester* model using

```
./bin/counter_tester
```

This will generate a CSV file named *counter_test_output.csv*.

```
sep=;
time;model_id;model_name;port_name;data
0;2;countUp file;;1.1
0;3;increment file;;1.1
0;4;counter model;;{count: 0, increment: 1}
1;4;counter model;count_out;0
1;4;counter model;;{count: 1, increment: 1}
1.1;2;countUp file;out;0
1.1;2;countUp file;;1.2
1.1;3;increment file;out;2
1.1;3;increment file;;6.4
1.1;4;counter model;;{count: 1, increment: 2}
2;4;counter model;count_out;1
2;4;counter model;;{count: -1, increment: 2}
2.3;2;countUp file;out;1
2.3;2;countUp file;;inf
2.3;4;counter model;;{count: -1, increment: 2}
3;4;counter model;count_out;-1
3;4;counter model;;{count: 1, increment: 2}
4;4;counter model;count_out;1
4;4;counter model;;{count: 3, increment: 2}
5;4;counter model;count_out;3
5;4;counter model;;{count: 5, increment: 2}
6;4;counter model;count_out;5
6;4;counter model;;{count: 7, increment: 2}
7;4;counter model;count_out;7
7;4;counter model;;{count: 9, increment: 2}
7.5;3;increment file;out;9
7.5;3;increment file;;inf
7.5;4;counter model;;{count: 9, increment: 9}
8;4;counter model;count_out;9
8;4;counter model;;{count: 18, increment: 9}
9;4;counter model;count_out;18
9;4;counter model;;{count: 27, increment: 9}
10;4;counter model;count_out;27
10;4;counter model;;{count: 36, increment: 9}
10;2;countUp file;;inf
10;3;increment file;;inf
10;4;counter model;;{count: 36, increment: 9}
```

Each line in the file contains:

- *Time*: the time values when an event occurred.
- *Model_id*: an internal identifier of the model that Cadmium uses to identify these models
- *Model_name*: the name of the model. This name is what you have passed into the constructor when initializing, for example: `addComponent<counter>("counter model");` here *counter model* is printed.
- *Port_name*: the name of the port where an output was generated. The names that show up here depend on what you provide whenever you initialize the port. For instance, in `addOutPort<int>("count_out");` the *port_name* is *count_out*. This item is only populated when logging an output event. In other cases, it would be left blank.
- *Data*: This is the data of whatever event took place. If the *port_name* column is blank, then the data shown in this column is that of a state transition, else it is of an output event.

The `sep=;` line at the top allows the software to recognize the delimiter. Below this, we see the column names: `time;model_id;model_name;port_name;data`, as above. After this, the actual logs of the models begin.

Looking at the first three lines after the header, we see that the first character is 0. This implies that these three events occur at time 0. Also, *port_name* is empty for all these three, which means that they represent state transition events. In fact, since this is occurring at time 0, these logs represent the initialization of the state variables of the models. The three models that are being initialized can be seen in the *model_name* column: first *countUp file* (the IEStream model coupled to *direction_in* of the counter model), *increment file* (the IEStream model whose output is coupled to *increment_in* of the

counter model), and *counter model*. The two IEStream models log their current time advance (that is, the time of the next internal transition): a value of 1.1, meaning that the models *countUp file* and *increment file* will have an internal transition schedule in 1.1 time units. Next, we see that the data column of the *counter model* is: `{count: 0, increment: 1}`. This corresponds to the constructor that we saw earlier: `explicit counterState (): count(0), increment(1), countUp (true), sigma(1.0){}`. Hence, we see count and increment being set to 0 and 1 respectively. We do not log all the state variables to make the logs more readable, but, as discussed earlier, logs can be modified.

With that, the initialization of the models is complete. Moving on, we see that at time 1, two events occur: one output event (*port_name* is populated) and one state transition (because it has an empty *port_name*). We see that the *counter model* outputs a value of 0 at the port *count_out*. This is the value of the state variable *count* in *counter model*. We then see the state transition event of *counter model* where the state variable *count* is incremented by the value of *increment* and becomes 1 (*count* + *increment* = 0 + 1 = 1).

At time instance 1.1, we see that the *countUp file* and *increment file* models produce an output of 0 and 2 respectively. This exactly follows what was provided in the text files provided to each of these models. Then, *countUp file* transitions to a time advance of 1.2 (which would mean it would next be triggered at time 2.4) and the *increment file* transition to a time advance of 6.4 (which means it would next be triggered at time 7.5) . We also see that a transition log of *counter model* also occurs at time 1.1, and we can deduce that this is the external transition function. We can see that the value of increment has indeed changed from 1 to 2. We are not logging the value of *countUp* and so we do not know if it has changed yet (this is important, you have to decide which variables are important for you. You should ideally not log all your state variables as this would make your logs unreadable).

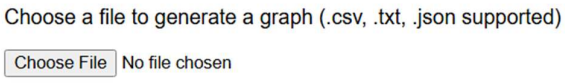
Now, at the time 2, we see that *counter model* outputs a 1, and in the next line transitions to a new state where the value of *count* is -1. This is because *countUp* indeed did change and *count* was calculated as *count* - *increment*. Further, this transition occurred at time instance 2 and not 2.1 because of the *sigma* -= *e* that was added in the external transition function. We see this continue until the end of the simulation

You may experiment with the various test files to see how your output changes. Even try commenting out one of the couplings to see how only ONE *IEStream* affects your model.

Viewing the output in the DEVS trace viewer

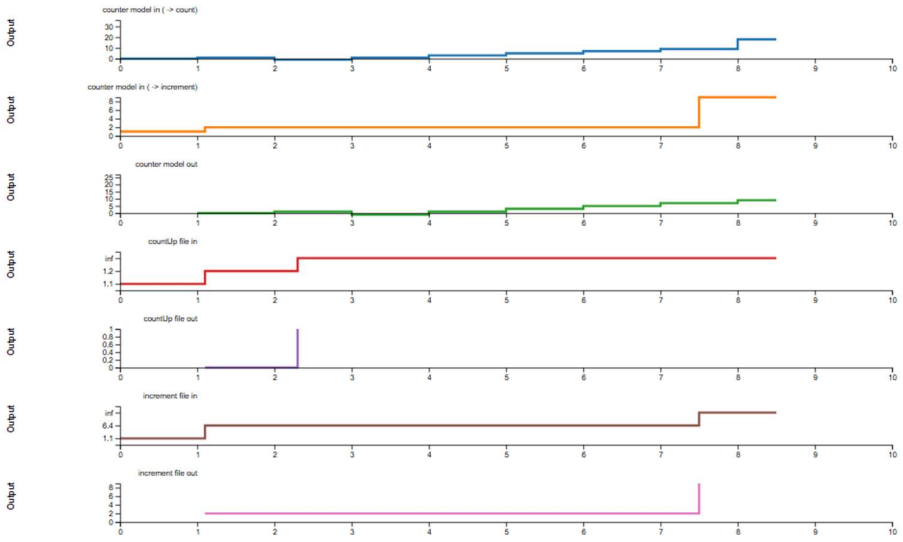
To view this output as in the trace viewer, go to this website: https://devssim.carleton.ca/graph-devs/

You will find a button to upload your .csv file:



Click on it, and choose the *counter_test_output.csv* file that we just looked at. After this, select the Graph Type from the drop-down menu. The multi-variable line graph is most appropriate for such trace files, we will look at that. Feel free to experiment. Then, proceed to generate the graph by clicking the Generate Graph method.

This trace viewer is a work in progress. It provides a simple way of seeing what your model is doing. Uploading the counter example would give you an output that looks like this:

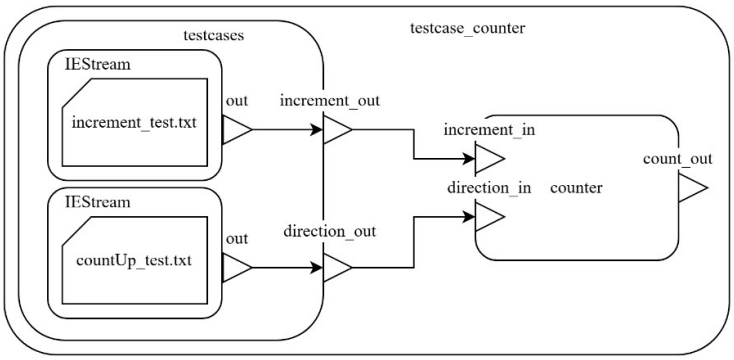


As you can see in the figure, the graphs for *counter model in (-> count)* and *counter model out* seem to be going out of the y-axis bounds, but you can clearly see the lag between the state transition of the *count* state variable (per legend: counter model (-> count)) and the output of the model (per legend: counter model out). From the figure you can also make out the values of increment from the change in the *increment* state variable in the *counter model* state transition (second from the top) and from the logs of the *increment file model* (bottom most graph).

Known bug: The graphing tool considers all logs other than output event logs (so state transition logs) to be input events, which is only true in the cases of external transition functions.

Appendix III – Two-level Coupled Model

The code snippets below define two new coupled models which extend the model defined in Appendix II. These models are organized differently to portray the modularity and hierarchy of DEVS and how it can be achieved in Cadmium. The model we are going to define can be seen in the image below:



The figure above defines a coupled model called *testcase_counter*, which consists of one coupled model called *testcases* and one atomic model called *counter*. We see that the *testcase* coupled model contains two *IEStream* atomic models that will read the *increment_test.txt* file and the *countUp_test.txt* files respectively (these are the very same ones used in Appendix II). The *testcases* coupled model has two output ports; the output ports *out* in the two *IEStream* models are connected to the *increment_out* and *direction_out* output ports.

The output ports of the *testcases* coupled model are connected to the *increment_in* and *direction_in* inputs of the *counter* atomic model. This now means that just replacing this coupled model would allow one to change the testcases being injected into the *counter* model. This is the foundation of *experimental frames* in DEVS.

The *testcases* coupled model, which can be found at under *main/include/testcases.hpp* in https://github.com/Sasisekhar/DEVS_manual_example is as follows:

```
#include "cadmium/modeling/devs/coupled.hpp" //cadmium header
#include "cadmium/lib/iestream.hpp"          //IEStream header

using namespace cadmium;

struct testcases: public Coupled {

   //The ports
   Port<int> increment_out;
   Port<bool> direction_out;

   testcases(const std::string& id): Coupled(id){
     //setting the ports as output ports
     increment_out = addOutPort<int>("increment out");
     direction_out = addOutPort<bool>("direction out");

     //adding the IEStream models
     auto increment_file = addComponent<lib::IEStream<int>>("increment file", "/absolute file path/");
     auto countUp_file = addComponent<lib::IEStream<bool>>("countUp file", "/absolute file path/");

     //performing the EOC
     addCoupling(increment_file->out, increment_out);
     addCoupling(countUp_file->out, direction_out);
   }

};
```

The start of the model is the same as in Appendix II: we include the libraries required, and then we define the coupled model struct *testcases*, which inherits from the *Coupled* class. We then declare the two Ports *increment_in* and *direction_out* as we would declare them in atomic models. Within the constructor, we initialize and set the direction of both the ports as outputs using the `addOutPort<>()` method. Then, using the *addComponent* method, we add the two atomic models that are part of the *testcases* coupled model. As the constructor parameter for the two *IEStreams*, we see the id *"increment file"* and *"countUp file"* being passed as names for the logs. However, *IEStream* has a second constructor argument, and that is the absolute file path to the input testcase file.

Finally, we use the *addCoupling*<>() method to define the external output couplings of the coupled model.

Moving onto the code for the *testcase_counter* coupled model:

```
#include "cadmium/modeling/devs/coupled.hpp"
#include "counter.hpp"
#include "testcases.hpp"

using namespace cadmium;

struct testcase_counter: public Coupled {

    testcase_counter(const std::string& id): Coupled(id){
        auto counter_model = addComponent<counter>("counter model");
        auto testcases_model = addComponent<testcases>("testcases model");

        addCoupling(testcases_model->increment_out, counter_model->increment_in);
        addCoupling(testcases_model->direction_out, counter_model->direction_in);
    }

};
```

We first include the two model definition files used in the coupled model: *counter.hpp* and *testcases.hpp*. Next, we define the coupled model struct *testcase_counter*, which inherits from the *Coupled* class.

Within the constructor, using the *addComponent* method, we add the two models that are part of the *testcase_counter* coupled model (Notice how the method for including and atomic model and a coupled model is the same). As the constructor parameter for the two models, we pass the id *"counter model"* and *"testcases model"* being passed as names for the logs.

Finally, we use the *addCoupling*<>() method to define the internal couplings of the coupled model.

Running the model

To run the model, clone this repository: https://github.com/Sasisekhar/DEVS_manual_example. The repository contains multiple models, this time we are interested in the *testcase_counter* for this section.

The text files for *IEStream* can be found in the parent directory of the repository, and they are named *increment_test.txt* and *countUp_test.txt*. Their contents by default are this:

- increment_test.txt

```
1.1 2
7.6  9
```

- countUp_test.txt

```
1.1 0

2.3 1
```

The test run in this section is based on the above values in the text files. The absolute path to these files must be provided to the constructor of *IEStream* in the *testcases.hpp* file. So, in this case, the *IEStream* models will be instantiated as:

```
auto increment_file = addComponent<lib::IEStream<int>>("increment file",
               "/your-filepath-here/ /DEVS_manual_example/increment_test.txt");

auto countUp_file = addComponent<lib::IEStream<bool>>("countUp file",
               "/your-filepath-here/ /DEVS_manual_example/countUp_test.txt");
```

Once you have defined the paths of your files, build the models using:

```
source build_sim.sh
```

and then run the *testcase_counter* model using

```
./bin/testcase_counter
```

This will generate the output logs in your terminal:

```
time;model_id;model_name;port_name;data
0;2;countUp file;;1.1
0;3;increment file;;1.1
0;4;counter model;;{count: 0, increment: 1}
1;4;counter model;count_out;0
1;4;counter model;;{count: 1, increment: 1}
1.1;2;countUp file;out;0
1.1;2;countUp file;;1.2
1.1;3;increment file;out;2
1.1;3;increment file;;6.4
1.1;4;counter model;;{count: 1, increment: 2}
2;4;counter model;count_out;1
2;4;counter model;;{count: -1, increment: 2}
2.3;2;countUp file;out;1
2.3;2;countUp file;;inf
2.3;4;counter model;;{count: -1, increment: 2}
3;4;counter model;count_out;-1
3;4;counter model;;{count: 1, increment: 2}
4;4;counter model;count_out;1
4;4;counter model;;{count: 3, increment: 2}
5;4;counter model;count_out;3
5;4;counter model;;{count: 5, increment: 2}
6;4;counter model;count_out;5
6;4;counter model;;{count: 7, increment: 2}
7;4;counter model;count_out;7
7;4;counter model;;{count: 9, increment: 2}
7.5;3;increment file;out;9
7.5;3;increment file;;inf
7.5;4;counter model;;{count: 9, increment: 9}
8;4;counter model;count_out;9
8;4;counter model;;{count: 18, increment: 9}
9;4;counter model;count_out;18
9;4;counter model;;{count: 27, increment: 9}
10;4;counter model;count_out;27
10;4;counter model;;{count: 36, increment: 9}
10;2;countUp file;;inf
10;3;increment file;;inf
10;4;counter model;;{count: 36, increment: 9}
```

You will notice that the output log of this model is the same as that of *counter_tester*. This is because at the input/output level the two models are identical. Their difference is only in structure, yet it is an important distinction.

Appendix IV – the main file

The code block given below is the *main.cpp* file used to simulate the *testcase_counter* model defined in Appendix III.

```
#include <limits> //Required for infinity
#include "include/testcase_counter.hpp"
#include "cadmium/simulation/root_coordinator.hpp"
#include "cadmium/simulation/logger/stdout.hpp"
#include "cadmium/simulation/logger/csv.hpp"

using namespace cadmium;

int main() {

    auto model = std::make_shared<testcase_counter> ("testcase counter");
    auto rootCoordinator = RootCoordinator(model);

    // rootCoordinator.setLogger<STDOUTLogger>(";");
    rootCoordinator.setLogger<CSVLogger>("display_log_output.csv", ";");

    rootCoordinator.start();
    rootCoordinator.simulate(10.1);
    rootCoordinator.stop();

    return 0;
}
```

From the top, the `#include` directive must be used to include the appropriate Cadmium APIs and models required simulation. In this case we see *limits* which is required for infinity (`std::numeric_limits<double>::infinity()`), the top model *testcase_counter.hpp*, *cadmium/simulation/root_coordinator.hpp* which contains the code for the root coordinator, and finally *cadmium/modeling/simulation/stdout.hpp* and *cadmium/modeling/simulation/csv.hpp* which contains all the Cadmium APIs for logging.

Then, `using namespace cadmium` tells the compiler that the APIs we are using are indeed that of cadmium. This helps in reducing ambiguity, especially if multiple libraries were involved.

In the *main*() function, we first create the pointer to the top coupled model, calling it *model* by convention. We then pass this *model* pointer to the *RootCoordinator* class, whose object is *rootCoordinator*. Then we move onto setting the logger. Both the code for setting *STDOUTLogger* and *CSVLogger* are present in the code, the user may uncomment any one of these lines to observe the output logs in the terminal or in a CSV file.

Finally, the *start*() method of *RootCoordinator* is called, which will initialize the simulation. Then the *simulate*() method is called with the parameter being how long the simulation runs for. And finally, the *stop*() method ends the simulation.