Proceedings of the 2022 Winter Simulation Conference
B. Feng, G. Pedrielli, Y. Peng, S. Shashaani, E. Song, C.G. Corlu, L.H. Lee, E.P. Chew, T. Roeder, and P. Lendermann, eds.

DEFINING DEVS MODELS USING THE CADMIUM TOOLKIT

Gabriel Wainer
Cristina Ruiz Martin

Department of Systems and Computer Engineering
Carleton University
1125 Colonel By Drive
Ottawa, ON K1S 5B6, CANADA

ABSTRACT

Discrete Event System Specification (DEVS) is a mathematical formalism to model and simulate discrete-event dynamic systems. Using DEVS for modeling and simulation has numerous advantages, which include a rigorous formal definition of models, a well-defined mechanism for modular composition, and separation of concerns between the model definition and the simulation of the model, among others. In this tutorial, we will explain DEVS and present how to develop DEVS models using one of the multiple DEVS simulators: Cadmium. Cadmium is a DEVS simulator based on C++17. We will discuss the tool's Application Programming Interface and we will present a model for the Rock-Paper-Scissors game as an example to explain how to define models in DEVS and implement them in Cadmium.

1 INTRODUCTION

Modeling and Simulation (M&S) has become an essential tool in science and engineering. Its ability to represent problems in several disciplines and perform scientific exploration has increased its popularity. There are many methodologies to develop M&S solutions, and some of them allow defining the models formally, which has a few advantages such as being able to discuss and show the model to stakeholders prior to the implementation.

In particular, the Discrete Event System Specification (DEVS) (Zeigler et al. 2000) provides a theoretical framework to develop discrete-event M&S and it was used in many applications since its creation. More specifically, DEVS 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.

Since the introduction of the DEVS formalism, several DEVS simulators have been developed. The use of DEVS simulators allows us to implement and simulate DEVS models without being worried about implementing, verifying, and validating the simulation algorithm. We can focus our efforts on the verification and validation of the model itself. Cadmium (Belloli et al. 2019) is one of those DEVS simulators. Cadmium allows to implement advanced DEVS models and has several advantages over other DEVS simulators such as advanced features that help with model verification and reduce the probability of running a simulation with a bug. It implements model checks for both atomic and coupled models. Some of them are in compilation time and others in execution time, but all of them before a simulation starts. Cadmium is implemented as a library written in C++, compliant with the C++17 and the Boost library coding standard. It supports multiple data types for the Time, and Messages, and compiles in multiple platforms, including Linux, Windows, and Mac OS. Cadmium is freely available on GitHub (https://github.com/SimulationEverywhere/cadmium).

2022 Winter Simulation Conference (WSC) | 978-1-6654-7661-4/22/$31.00 ©2022 IEEE | DOI: 10.1109/WSC57314.2022.10015492

In this tutorial, we will explain the DEVS formalism along with the key features of the Cadmium simulator and how to use it to build DEVS models. Section 2 introduces the DEVS formalism, its advantages, DEVS-Graphs, and the current state of the art in DEVS simulators. In section 3, we present the Cadmium API and in Section 4, we present a case study to show how to define and execute DEVS models.

2 THE DEVS FORMALISM

The DEVS formalism (*Discrete Event System Specification*) is a formal discrete-event modeling and simulation methodology (Zeigler et al. 2000). It is derived from Systems Theory, and it allows one to define hierarchical modular models that can be easily reused. In DEVS, an atomic model defines the behavior of a component. An atomic model is defined by describing the set of states the model goes through, an internal and an external transition function, an output function, and a state duration function. At every point in time, the model is in a given state that has an associated duration for that state. When the time elapses, an output is transmitted, and the internal transition takes place to change the model state. A state can also change when an external event is received, which triggers the execution of the external transition function. DEVS models can be put together by linking the outputs of a model to inputs of other models to form coupled models, which can also be coupled together.

The use of DEVS provides several advantages. It allows developing hierarchical models in a modular fashion, improving model reuse. The model definition, implementation, and experimentation are defined by independent formal specification and execution apparatus. The same model can be implemented on different platforms facilitating the reliability of models and results. Moreover, the DEVS simulation algorithm has been formally verified, ensuring the correctness of execution of atomic and coupled models.

DEVS can also be seen as a common denominator for many other formalisms (Vangheluwe 2000), as seen in Figure 1 where we show a Formalism Transformation Graph. This allows using DEVS as a common denominator that can be used to compose models defined in different formalisms.

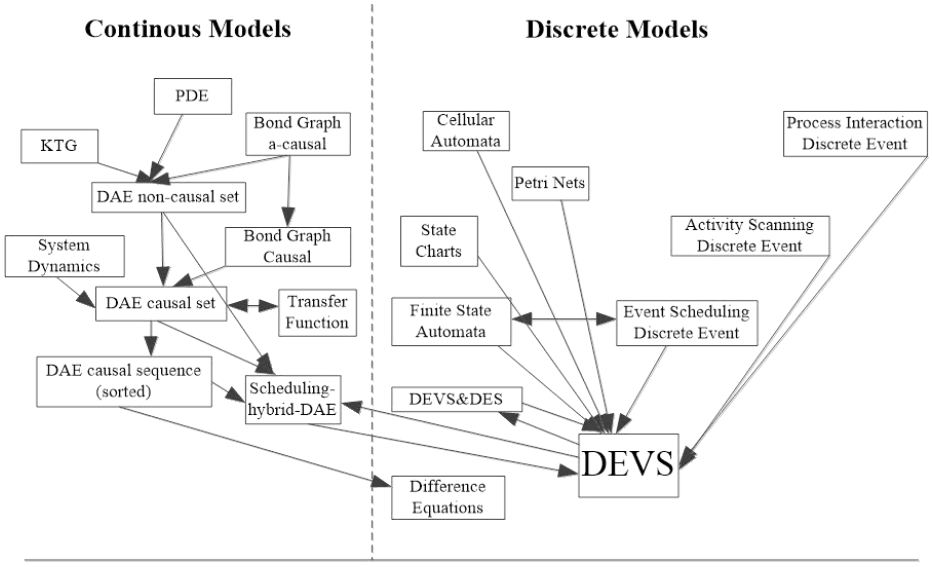

Figure 1: Formalism Transformation Graph. Adapted from (Vangheluwe 2000).

2.1 DEVS Atomic Models

Atomic models are the basic building blocks of all DEVS models: they are the lowest level indivisible models that encapsulate a portion of the modeled systems' behavior. Equation (1) shows the formal specification of DEVS atomic models (Zeigler et al. 2000).

$$M = \langle X, Y, S, \delta_{ext}, \delta_{int}, \delta_{conf}, \lambda, ta \rangle \quad (1)$$

The above specification consists of a number of individual elements:
$X = \{(p, v)|p \in IPorts, v \in X_p\}$ is the set of input events, where *IPorts* represents the set of input ports and X_p represents the set of values for the input ports;
$Y = \{(p, v)|p \in OPorts, v \in Y_p\}$ is the set of output events, where *OPorts* represents the set of output ports and Y_p represents the set of values for the output ports;
S: is the set of sequential states;
δ_{ext}: $Q \times X^b \rightarrow S$ is the external state transition function, with $Q = \{(s, e)|s \in S, e \in [0, ta(s)]\}$ and e is the elapsed time since the last state transition;
δ_{int}: $S \rightarrow S$ is the internal state transition function;
δ_{conf}: $Q \times X^b \rightarrow S$ is the confluent transition function;
λ: $S \rightarrow Y^b$ is the output function; and
ta: $S \rightarrow R^+_0 \cup \infty$ is the time advance function.

Once a model enters a state, it will stay there until one of two conditions are met:

1. An external event occurs causing the model to transition into a new state, depending on the events received and the time of receipt, as specified by δ_{ext}.
2. The elapsed time has reached the value of the time advance function in the current state, necessitating an internal transition to occur according to δ_{int}.

The confluence function determines the behavior of the model when both an internal and external transition are scheduled at the same time.

When an internal transition occurs, a bag of outputs is generated through the output ports allowing the model to interact with other models. After each state change, a new time advance is calculated, and the next internal event is scheduled. If the time advance is infinite the model is said to be passive, i.e., awaiting external inputs indefinitely.

2.2 DEVS Coupled Models

DEVS coupled models are comprised of either DEVS atomic models or other DEVS coupled models. This way, components can be built up hierarchically by coupling simple models together to form more complex models. Equation (2) shows the formal specification of DEVS coupled models (Zeigler et al. 2000).

$$CM = \langle X, Y, D, \{M_d | d \in D|\}, EIC, EOC, IC \rangle \quad (2)$$

The above specification consists of a number of individual elements:
X: $\{(p, v)|p \in IPorts, v \in X_p\}$ is the set of input events, where *IPorts* represents the set of input ports and Xp represents the set of values for the input ports;
Y: $\{(p, v)|p \in OPorts, v \in Y_p\}$ is the set of output events, where *OPorts* represents the set of output ports and Y_p represents the set of values for the output ports;
D: is the set of the component names and for each d ∈ D;
M_d: is a DEVS basic (i.e., atomic or coupled) model;
EIC: is the set of external input couplings, $EIC \subseteq \{((Self, in_{Self}), (j, in_j))|in_{Self} \in IPorts, j \in D, in_j \in IPorts_j\}$;
EOC: is the set of external output couplings, $EOC \subseteq \{((Self, out_{Self}), (j, out_j))|out_{Self} \in IPorts, j \in D, out_j \in OPorts_j\}$;
IC: is the set of internal couplings, $IC \subseteq \{((Self, out_i), (j, in_j))|i, j \in D, out_i \in OPorts_i, in_j \in IPorts_j\}$.

The purpose of the coupling elements of the formal specification is to define how each of the sub-models in the coupled model (Md) connect to the inputs of the coupled model (through the elements of EIC), send outputs from the coupled model (through the elements of EOC), and connect to each other (through elements of IC). All these couplings illustrate the structure of the model and, when combined with all the sub-model definitions, determine how the model will behave. It has been proven that for any given

coupled model an equivalent atomic model can be created, this is called the closure under coupling property. The advantage is that through the decomposition of models we can create and design complex models through simple components

2.3 Graphical Specification of DEVS Models

A graphical specification (DEVS-graphs) can be used in specifying DEVS models (Praehofer and Pree, 1993). The main benefit of the DEVS-graphs notation is that it allows for greater interaction and clearer communication with stakeholders who may have limited knowledge of the formalisms, mathematical notation, and programming skills. Additionally, it assists the modeler with visualizing the system. These benefits result in a developed system that accurately represents the stakeholders' needs.

Notations exist for modeling both atomic and coupled DEVS models. For atomic models (Figure 2), this notation is similar to a finite state machine: there are nodes (representing the states) connected by directed edges (representing the transitions), though several key extensions have been made.

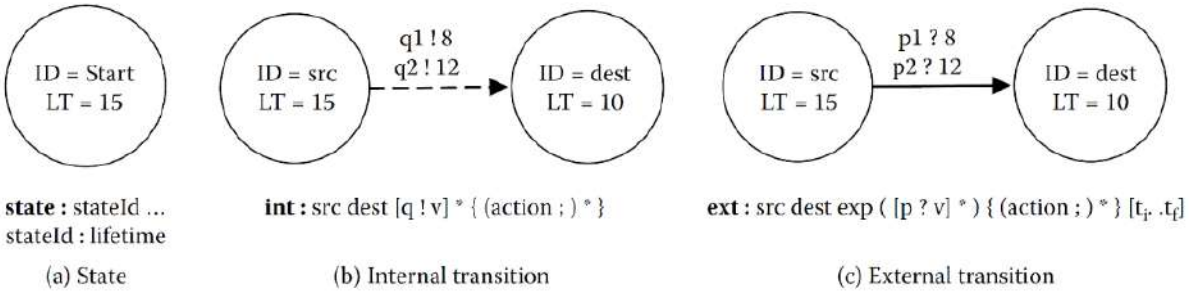

Figure 2: DEVS Atomic Models using DEVS-Graphs (Wainer 2009).

As shown in Figure 2, the edges connecting nodes can be of two types: a dashed line to represent an internal transition or a solid line to represent an external transition. Some additional annotations are required to fully define the model:

1. Each state must be labeled with an ID and a time advance value (LT) associated to the state.
2. Each internal transition must be labeled with any outputs to ports and the associate output values that are generated when the internal transition fires.
3. Each external transition must be labeled with the input port and value pair that must be received in order for the transition to occur.

Coupled models can then be built up by connecting the inputs and outputs from atomic models to each other. It is useful in this circumstance to hide the structure of the atomic model inside a black box (Figure 3), and focus on the interfaces of the model when constructing coupled models. Atomic and coupled models are then connected together using arrows, the ends of which specify the sending or receiving port.

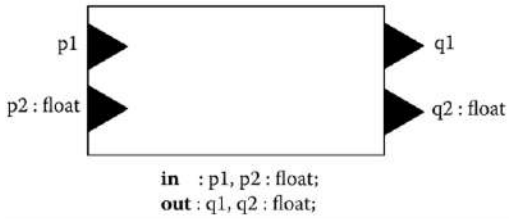

Figure 3: DEVS Coupled Models using DEVS-Graphs (Wainer 2009).

2.4 DEVS Simulators

There are different DEVS Simulators, each of them with its advantages and disadvantages. Van Tendeloo and Vangheluwe (2017) provide an overview of the current state of eight different DEVS simulators, selected based on their functionality, DEVS compliance, and performance. In their study, they specifically evaluate ADEVS (a Discrete Event Simulator) (Nutaro 2014), PythonPDEVS (Van Tendeloo and Vangheluwe 2014), CD++ (Wainer 2002), VLE (Virtual Laboratory Environment) (Quesnel et al. 2007), MS4 Me (Seo et al. 2013), DEVS-Suite (Kim et al. 2009), PowerDEVS (Bergero and Kofman 2011), and X-S-Y (Hwang 2012).

As of 2017, ADEVS was the fastest well-established DEVS framework evaluated so far (Van Tendeloo and Vangheluwe 2017). *ADEVS* is implemented as a lightweight C++ library and both atomic and coupled models are defined using C++ programming language. ADEVS has evolved during the last 15 years, has over 29 releases, and supports high performance by relying on optimized data structures.

The performance of PythonPDEVS is close to *ADEVS*. The main advantage is that it offers a variety of schedulers (sorted list, activity map, and a heapset) for speedup. PythonPDEVS supports dynamic typing; therefore, all the messages are not required to be of the same type. PythonPDEVS has sequential and distributed variants (Van Tendeloo and Vangheluwe 2014b) but both focus on computational activity information to reduce simulation time. New features (such as pause, resume, and step the simulation) have been added to help debugging (Van Mierlo et al. 2017).

Other DEVS simulators have also relevant features. For example, CD++ allows to implement both DEVS and Cell-DEVS models and couple them together. VLE couples multiple simulators within a DEVS-Bus architecture and uses DEVS for coordination. MS4Me allows building models using a custom natural-like language called DNL combined with Java. DEVS-Suite is the successor of DEVSJava (Sarjoughian and Zeigler 1998) and it includes additional features such as visualization of coupled model simulation. PowerDEVS contains a graphical modeling environment. X-S-Y is written in Python and its more relevant feature is the verification of finite and deterministic DEVS models.

A more recent DEVS simulator is CD-Boost (Vicino et al. 2015) a new DEVS simulator that introduced a sequential architecture and an effective implementation of the abstract simulator presented in (Chow et al. 1994). The authors evaluated the performance by comparing it to *adevs* using DEVStone, and it showed that no new overhead was introduced. It outperforms *adevs* when there are numerous simultaneous events. However, CD-Boost had several disadvantages such as the lack of multiple input/output ports, or different types of messages for different models. This resulted a lack of use by the community because the complexity implementing the models, the difficulties in verification of the models, and slow compilation (and massive use of memory). In (Belloli et al. 2019), we presented Cadmium, an evolution of CD-Boost that solves the two main disadvantages of CD-Boost simulator: (1) lack of ports and (2) the restriction of using a single type of data for all the messages. Additionally, we implemented other advanced features that help with model verification and reduces the probability of running a simulation with a bug. Cadmium implements model checks for both atomic and coupled models. Some of them are in compilation time and others in execution time, but all of them before a simulation starts. For atomic models, it checks that all the methods of the atomic model are defined, that the model port types are correct, and that the model has a state defined. For coupled models, Cadmium checks that there are no connections between ports with different message types, that there are no invalid connections based on the model EIC, EOC or IC link structure, and that all the submodels inside a couple are valid (i.e. it performs atomic model checks). Cadmium is implemented as a library written in C++, compliant with the C++17 and the Boost library coding standard. It supports multiple data types for the Time, and Messages, and compiles in multiple platforms, including Linux, Windows, and Mac OS. Cadmium is available on GitHub (https://github.com/SimulationEverywhere/cadmium).

3 CADMIUM APPLICATION PROGRAMMING INTERFACE

Cadmium provides three main functionalities: an interface for defining atomic models, an interface for defining coupled models, and methods to run the simulation and define the structure and contents of simulation logs. Figure 4 shows the interface used for defining atomic models.

```
1 struct AtomicName_defs{        //Input&Output Port declaration
2    struct input_port1 : public in_port< MSGi1> {};
3    struct input_portn : public in_port< MSGin> {};
4    struct output_port1 : public out_port< MSGo1> {};
5    struct output_portn : public out_port< MSGon> {};   };
6
7 template<typename TIME> class AtomicName{
8    using defs=AtomicName_defs;   //port definition in context
9  public:
10  // Model parameters
11  struct state_type{ //Define your state variables };
12  state_type state;
13  AtomicName() noexcept { //parameters/initial state values}
14
15  //DEVS functions
16  void internal_transition() {
17  // Define internal transition function }
18  void external_transition(TIME e, typename make_message_bags
19     <input_ports>::type mbs) {
20  // Define external function       }
21  void confluence_transition(TIME e, typename
22          make_message_bags <input_ports>::type mbs) {
23   // Define confluent function        }
24  typename make_message_bags<output_ports>::type output() const {
25   // Output function
26    typename make_message_bags<output_ports>::type bags;
27    //Define your output function. Fill bags
28    return bags;             }
29 TIME time_advance() const {
30      //Define time advance function       }
31 };
```

Figure 4: DEVS atomic model implementation using Cadmium.

As seen in Figure 4, we first declare the model ports as structures (lines 1-5) using `in_port` and `out_port` classes defined in the simulation tool. The atomic model is defined as a template class of type *Time* (lines 7-31) in order to allow atomic models to be instantiated with different data types for time. This gives flexibility in the data structures chosen for the definition of simulated time. Each atomic model class can contain as many parameters as needed (line 10). It has a set of state variables grouped together in a structure called `state_type` (lines 11-12). It also has a model constructor to instantiate the model parameters and initialize the state (line 13). There may be additional constructors with different input parameters, but the default constructor must also be present. Then, we implement all the DEVS functions (internal, external, confluence, output, and time advance) using the template provided in lines 15-31) in C++. The code in bold cannot be modified as it is part of the simulator.

Once atomic models are defined, the coupled models are implemented using the template provided in Figure 5, which is an implementation of the coupled model shown in Figure 6.

```
1 //*****INSTANTIATE ATOMICS *******//
2 template<typename TIME>
3 class iestream_int : public iestream_input<int,TIME> {
4    public:
5      iestream_int(): iestream_input<int,TIME>
6      ("inputs/test_filterNetworks.txt") {};   };
7 //*****DEFINE COUPLED *******//
8  struct inp_in_1 : public in_port<int>{};
9  struct outp_out_1 : public out_port<double>{};
10 using iports_C1 = std::tuple< inp_in_1>;
11 using oports_C1 = std::tuple< outp_out_1>;
12 using submodels_C1=models_tuple<filterNet, iestream_int> ;
13 using eics_C1=tuple<EIC
14      <inp_in_1,iestream_int, >;
15 using eocs_C1 =tuple< EOC
16     < filterNet, filterNet _defs::out, outp_out_1> >;
17 using ics_C1=tuple<IC
18     <iestream_int,iestream_defs::out,
19           filterNet, fiterNet _defs::in> >;
20
21 using C1=coupled_model <TIME,iports_C1,oports_C1,
22           submodels_C1,eics_C1,eocs_C1,ics_C1>;
```

Figure 5: DEVS coupled model implementation using Cadmium.

After instantiating all atomic models (lines 1-6), we declare the coupled model ports (lines 8-9) as structures using `in_port` and `out_port` classes defined by the simulation tool (as we did with the atomic models in lines 2-5). Then, we assign the input and output ports we just declared to the model as a tuple (lines 10-11). Then, we create a tuple with all the submodels (line 12). The external input couplings (line 13-14), external output couplings (line 15-16), and internal couplings (line 17-19) are also defined as tuples. For example, the external input couplings is defined as a tuple of the coupled model input port (e.g., `inp_in_1`, the name of the submodel (e.g., `iestream_int`) and the input port of the submodel (e.g., `iestream_defs::in`). The internal couplings and the external output couplings are defined similarly. The coupled model (line 21-22) is defined as a tuple of all these components.

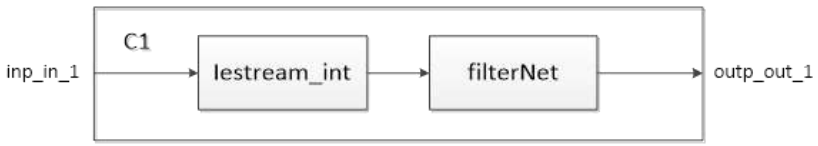

Figure 6: Example of a DEVS coupled model.

Once the top model is defined, we need to define loggers to store the simulation results, and define parameters to run the simulation. This is done as defined in Figure 7. The logger is defined by declaring the name of an output file (line 1). Then, we need to define the structure `oss_sink_messages` (lines 2-5) to tell define where to save the output log. In lines 7-9, we specify which elements we want to log. In this case, we define how the messages between the models are generated (lines 7-8). We also define how the state of the models is logged (line 9). In this specific case, we decided to log the messages output by the different models along with the global simulation time when each message was generated. Finally, we need to create the runner (line 11) and call the `run_until` method (line 12) with the simulation end time. The runner is created using the time class and the logger as template parameters and the top model name as parameter. A detailed manual for the Cadmium simulation tool can be found in (Ruiz-Martin and Wainer 2019).

```
1  static std::ofstream out_data("output_file_name.txt");
2    struct oss_sink_provider {
3       static std::ostream& sink()
4       return out_data;
5   };
6
7 using log_messages=cadmium::logger::logger<cadmium::logger::logger_messages,
8      cadmium::logger::formatter<TIME>, oss_sink_provider>;
9 using logger_top=cadmium::logger::multilogger<log_messages, global_time>;
10
11 cadmium::engine::runner<NDTime, logger_top> r(TOP, {0});
12 r.run_until(NDTime("04:00:00:000"));
```

Figure 7: Logger definition and simulator call in Cadmium.

4 ROCK-PAPER-SCISSORS GAME DEVS IMPLEMENTATION

In this section we use a simple modeling example to show how to define DEVS models and implement them using Cadmium. The model defines a discrete-event specification of the "rock-paper-scissors" game with two players. Each player chooses between rock, paper and scissors at the same time. The winner is calculated according to the following rules: rock beat scissors; scissors beat paper, paper beat rock.

4.1 Rock-Paper-Scissors Game DEVS Model Definition

The rock-paper-scissors game can be modeled using DEVS as a coupled model with 3 components (Figure 8): Player1, Player2, and Comparer. Each player is further decomposed into two atomic models: RequestReceiver to model the decision to play, and ActionMaker to choose one of the three options of the game. The Comparer determines which player won the round. RequestReceiver, ActionMaker and Comparer are atomic models.

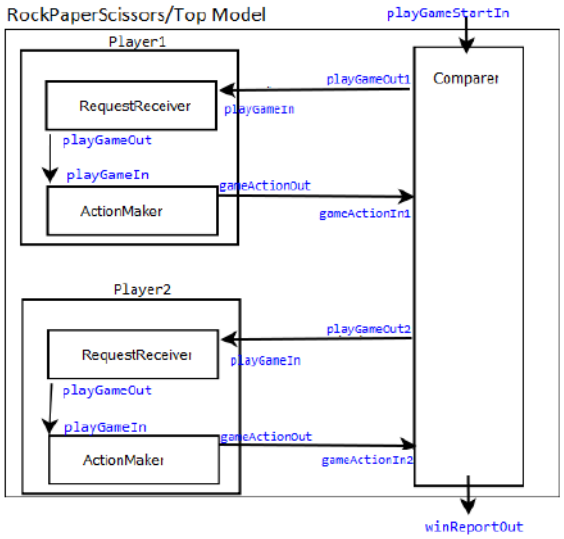

Figure 8: Rock-Paper-Scissors Coupled model.

RequestReceiver

This atomic model imitates the human instinct of receiving a request of doing something, pondering about the decision, and deciding to act. The formal definition of the atomic model is defined in equation (3) and the DEVS-Graph notation of the model is shown in Figure 9.

$$\text{RequestReceiver} = < X, Y, S, \delta int, \delta ext, \lambda, ta > \quad (3)$$

X = {(playGameIn ∈ {true, false})};
Y = {(playGameOut ∈ {true, false})};
S ={active, passive};
δint(S) = passive
δext(S, e, x){
if (x==true AND S == false) then S = active}
λ(active) = true
ta(active) = 20sec; ta(passive)= INFINITY

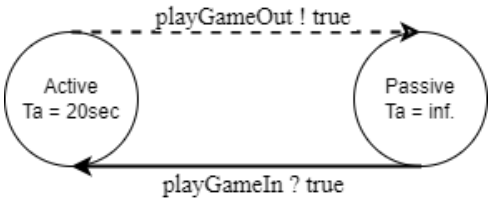

Figure 9: DEVS Graph of the request receiver atomic model.

ActionMaker

This atomic model defines the decision making process for choosing a motion (Rock, Paper or Scissors), and present this final decision. The formal definition of the atomic model is shown in (4).

$$\text{ActionMaker} = < X, Y, S, \delta int, \delta ext, \lambda, ta > \quad (4)$$

X = {playGameIn ∈ {true, false}};
Y = {gameActionOut ∈ {"R", "P", "S"}};
S = {(active ∈ {true, false}), (choice ∈ {"None", "R", "P", "S"})};
δint(active = true){
active = false
choice = "None" }
δext(s, e, x){
if (active == false)
active = true;
choice = random("R", "P", "S") // choose at random }
λ(active = true){ Send choice }
ta(active = true) = 3sec; ta(active = false) = infinity

Comparer

The Comparer atomic model begins the game upon receiving a trigger signal input of type from the playGameStartIn input port. From there, it waits 5 seconds before triggering the output function to alert the players that the game has begun. Once the model receives the players' choices, the model decides which player won based on their actions. The formal definition of the atomic model is shown in equation (5).

$$\text{Comparer} = < X, Y, S, \delta int, \delta ext, \lambda, ta > \quad (5)$$

X = {(playGameStartIn ∈ {true, false}), (gameActionIn1 ∈ {"R", "P", "S"}),
(gameActionIn2 ∈ {{"R", "P", "S"})};
Y = {(playGameOut1 ∈ {true, false}), (playGameOut2 ∈ {true, false}), (winReportOut ∈ {0 ,1, 2})};
S = {(active ∈ {true, false}), (playerResult1 ∈ {{"None", "R", "P", "S"}),
(playerResult2 ∈ {"None", "R", "P", "S"}), (received1 ∈ {true, false}), (received2 ∈{true, false},

```
    (playerIDWin ∈{-1,0,1,2}), (winnerTracker ∈ {[{0,1,2}],[{0,1,2}],…})}, (leading ∈ {0,1,2})}}
δint(s){
        active = false;
        received1 = false;
        received2 = false;
        playerIDWin = -1;
        playerResult1 = "None";
        playerResult2 = "None";            }
δext(s, e, x) {
        if (x.playGameStartIn == true)  // Start game
            if (active == false)   active = true;
        if (x.gameActionIn1)    //receiving player input
            if (active == true AND received1 == false)
                playerResult1 = x.gameActionIn1   // store the action of the player
                received1 = true;
        if (x.gameActionIn2)
            if (active == true AND received2 == false)
                playerResult2 = x.gameActionIn2   // store the action of the player
                received2 = true;
        //Evaluate who won if possible and update the leading board
        if (active == true AND received1 == true AND received2 == true)
            (playerResult1 == playerResult2)  ? playerIDWin = 0;
            (playerResult1 == "R" AND playerResult2 == "S") ? playerIDWin = 1;
            (playerResult1 == "S" AND playerResult2 == "P") ? playerIDWin = 1;
            (playerResult1 == "P" AND playerResult2 == "R") ? playerIDWin = 1;
            else playerIDWin = 2;
        Add the winner to winnerTracker vector
        //count who is leading the game
        if count 1s in winnerTracker > count 2s in winnerTracker:  leading = 1
        else if count 1s in winnerTracker = count 2s in winnerTracker: leading = 0
        else leading = 2                                                    }
λ(s) {
        if (active == true AND received1 == true AND received2 == true AND playerIDWin != -1)
            send playerIDWin through port winReportOut
        else if (active == true AND received1 == false)
           send true through port playGameOut1
        else if (active == true AND received2 == false)
            send true through port playGameOut2                          }
ta(s) {
        if (active == true AND (state.received1 == false OR state.received2 == false))
            5 sec //referee time to alert players
        else if (active == true AND received1 == true AND received2 == true)
            15 sec//referee time to make winning decision
        else
            infinity                }
```

4.2 Rock-Paper-Scissors Game Implementation in Cadmium

To show how to use Cadmium to implement the DEVS models, we will focus on the definition of one of the atomic models (ActionMaker) and the top coupled model. The complete model implementation and the

instruction on how to run the simulations are available on GitHub: `https://github.com/SimulationEverywhere-Models/RockPaperScissors`.

Atomic models are implemented in Cadmium in an hpp file as shown in Figure 10. We first declare the ports with the type of message that they will manage. In this case, the output port is a string (we need to represent `S`, `R` and `P`) and the input port it is a Boolean. Then, we define the atomic model as a class, in this case `ActionMaker`. We first define the model parameters (in this case, `thinkingTime`). Then, we define the input and output ports as tuples and then the state, the constructor/s and all the DEVS functions.

The model state is defined as a struct called `state_type` which includes as many attributes as needed. In this case, we use two: `active` (a Boolean representing if a decision must be notified) and `choice` (a string representing the decision, i.e., `S`, `R`, `P` or `None`). For the constructor, we must define a default constructor that does not take any parameters, but we can add additional constructors if needed. In the constructor, we define the initial state of the model. In this case, the model in not active and the choice is `None`.

For implementing the DEVS functions, we just need to translate the formal definition described in the previous section into C++ using the Cadmium interface.

```
struct ActionMaker_defs{   //Port declaration
   struct gameActionOut : public out_port<string> {};
   struct playGameIn : public in_port<bool> {};
};

template<typename TIME> class ActionMaker{
   TIME thinkingTime = TIME("00:00:03:000")

   public:

   // ports definition
   using input_ports=tuple<typename ActionMaker_defs::playGameIn>;
   using output_ports=tuple<typename ActionMaker_defs::gameActionOut>;

   struct state_type {    // state definition
      bool active; //triggered to true when playGameIn received
      string choice; //player choice
   };

   state_type state;

    ActionMaker() {      // default constructor
      state.active = false; // set to true once game request received
      state.choice = "None"; // means no choice
   }

   void internal_transition() {    // internal transition function
      state.active = false;
      state.choice = -1;
   }

   void external_transition(TIME e,typename make_message_bags<input_ports>::type mbs){
      if (state.active == false) {
         state.active = true;
         std::random_device rand;
         std::mt19937 generate(rand());
         std::uniform_int_distribution<> distribute(1, 3);
         value = distribute(generate);     //generate a value between 1-3
         if (value == 1) state.choice = "R";
         else if (value == 2) state.choice = "P";
```

```
        else if (value == 3) state.choice = "S";
      }
   }

   void confluence_transition(TIME e,typename make_message_bags<input_ports>::type
                              mbs) {    // confluent transition function
      internal_transition();
      external_transition(TIME(), move(mbs));
   }

   typename make_message_bags<output_ports>::type output() const {    // output function
      typename make_message_bags<output_ports>::type bags;
      if (state.active == true) {
         vector<string> gameChoice;             //output choice
         gameChoice.push_back(state.choice);
         get_messages<typename ActionMaker_defs::gameActionOut>(bags) = gameChoice;
      }
      return bags;
   }

   TIME time_advance() const {             // time_advance function
      if   (state.active)  return thinkingTime
      else return numeric_limits<TIME>::infinity();
   }
};
```

Figure 10: Cadmium code snippet: ActionMaker atomic model.

Once all the atomic models are defined, we need to instantiate them and implement the coupled model as we show in Figure 11. We first specify the time type for the model (we can use a float or any other class). In this case, we use a custom data type, `NDTime`. We then declare the ports for all the coupled models. In this case, the output port of the top model `winReportOutP` of type `out_port` (a class provided by the simulator) and using `int` messages. We do the same for the ports of the Person coupled model. Then, we specialize all the atomic models that are defined using a template system. In this case, we specialize a model that parses the input file (`iestream_input`) as `GameTrigger` with the message type `bool`. Finally, we define the main function including the logger definition, the atomic models' instantiation, the implementation of the coupled models using the template provided in Figure 5. We use the standard logger definition and the atomic models are instantiated with the command `cadmium::translate::make_dynamic_atomic_model`, which includes the atomic model identifier (i.e. a unique name such as `play_game_gen` to create an instance of the `GameTrigger`, and the parameters for the constructor. In the case of `play_game_gen`, we include the input file from where to read the inputs. All atomic models are instantiated in a similar way. For each coupled model, we need to declare, as arrays, the sub-models, input ports, output ports, external input couplings, external output coupling, and internal coupling. For example, Player1 has two submodules defined as `modeling::Models submodels_Player1 = {requestReceiver1, actionMaker1};` one input port defined as `modeling::Ports iports_Player1 = {typeid(playGameIn)};` one output port defined as `modeling::Ports oports_Player1 = {typeid(gameActionOut)}`; one external input coupling (EIC) connected to the `RequestReciver` model where `playGameIn` is the input port of the Player coupled model and `RequestReceiver_defs::playGameIn is the` port of the RequestReceiver atomic model; one external output coupling (EOC) connected to the ActionMaker atomic model where `ActionMaker_defs::gameActionOut` is the output port of the ActionMaker atomic model and `gameActionOut is the output` port of the Player coupled model; and one internal input (IC) coupling connecting the output port of RequestReceiver with the input port of ActionMaker. The coupled models are defined with the command `std::make_shared<cadmium::modeling::coupled<TIME>>` as an array of the six previous elements preceded by the coupled model identifier. Player2 and the top model are defined in a similar way.

```
using TIME = NDTime; //Specify the time class
struct winReportOutP : public out_port<int> {}; // Declare output ports for coupled model

// Declare output ports for Player Coupled Model
struct gameActionOut : public out_port<string> {};
struct playGameIn : public in_port<bool> {};

// Specialize the atomic parser to send inputs to the top model
template<typename T> class GameTrigger : public iestream_input<bool, T> {
   public:
      GameTrigger() = default;
      GameTrigger (const char* file_path) : iestream_input<bool, T>(file_path) {}
};

int main(int argc, char ** argv) {  // Loggers
  static std::ofstream out_data("rock_sissors_paper_out.txt"); //define output file name
    struct oss_sink_provider{ static std::ostream& sink(){  return out_data; }         };
  using log_messages=cadmium::logger::logger<cadmium::logger::logger_messages,
        cadmium::logger::formatter<TIME>, oss_sink_provider>;

  string input_data = argv[1];    // Instantiate GameTrigger
  const char * i_input_data = input_data.c_str();
  std::shared_ptr<cadmium::modeling::model> play_game_gen =
    cadmium::translate::make_dynamic_atomic_model<ApplicationGen, TIME, const
    char* >(" play_game_gen " , std::move(i_input_data));

  shared_ptr<modeling::model> comparer;   // Instantiate Comparer
  comparer = translate::make_dynamic_atomic_model<Comparer, TIME>("comparer");

  // Player 1 Coupled Model definition
  shared_ptr<modeling::model> requestReceiver1 =
    translate::make_dynamic_atomic_model<RequestReceiver,TIME>("requestReceiver1");

  shared_ptr<modeling::model> actionMaker1 =
    translate::make_dynamic_atomic_model<ActionMaker,TIME>("actionMaker1");

  modeling::Ports iports_Player1 = {typeid(playGameIn)};
  modeling::Ports oports_Player1 = {typeid(gameActionOut)};
  modeling::Models submodels_Player1 = {requestReceiver1, actionMaker1};
  modeling::EICs eics_Player1 = {
    translate::make_EIC<playGameIn,RequestReceiver_defs::playGameIn>("requestReceiver1") };
  modeling::EOCs eocs_Player1 = {
   translate::make_EOC<ActionMaker_defs::gameActionOut,gameActionOut>("actionMaker1") };
  modeling::ICs ics_Player1 = {
     translate::make_IC<RequestReceiver_defs::playGameOut, ActionMaker_defs::playGameIn>
          ("requestReceiver1","actionMaker1")
};

  shared_ptr<modeling::coupled<TIME>> Player1 = make_shared<modeling::coupled<TIME>>
       ("Player1",submodels_Player1, iports_Player1, oports_Player1, eics_Player1,
            eocs_Player1, ics_Player1);
...

  /*******TOP MODEL********/
    modeling::Ports iports_TOP = {};
    modeling::Ports oports_TOP = {typeid(winReportOutP)};
    modeling::Models submodels_TOP = {play_game_gen, Player1, Player2, comparer};
    modeling::EICs eics_TOP = {};
    modeling::EOCs eocs_TOP = {
```

```
        translate::make_EOC <Comparer_defs::winReportOut, winReportOutP>("comparer")  };
    modeling::ICs ics_TOP = {
        translate::make_IC<iestream_input_defs<bool>::out,Comparer_defs::playGameStartIn>
            ("play_game_gen","comparer"),translate::make_IC<Comparer_defs::playGameOut1,
                Player1_defs::playGameIn>("comparer1","Player1"),

      translate::make_IC<Comparer_defs::playGameOut2,Player2_defs::playGameIn>
              ("comparer1","Player2"),
      translate::make_IC<gameActionOut,Comparer_defs::gameActionIn1>("Player1","comparer"),
      translate::make_IC<gameActionOut,Comparer_defs::gameActionIn2>("Player2","comparer")
    };

    shared_ptr<modeling::coupled<TIME>> TOP = make_shared<modeling::coupled<TIME>>("TOP",
             submodels_TOP, iports_TOP, oports_TOP, eics_TOP, eocs_TOP, ics_TOP);

    // Call Runner
    engine::runner<NDTime, logger_top> r(TOP,{0});
    r.run_until(NDTime("00:10:00:000"));
}
```

Figure 11: Rock-Paper-Scissors coupled model definition.

After implementing all the coupled models, we define the runner as `cadmium::engine::runner<NDTime, logger_top> r(TOP, {0})`, where `NDTime` represents the time type, `logger_top` is the logger we have selected, TOP is the name of the variable where the Top Model is defined and {0} is the simulation starting time. Finally, we call the runner with the simulation finishing time as a parameter as explained in Figure 7.

4.3 Simulating the Rock-Paper-Scissors Game

In Figure 12, we show a simulation log for a scenario where the game is played for one round. As we can see in the log file, at time 2 min, the signal to trigger the game is generated. 5 seconds after, the comparer notifies the players that an action must be generated. 20sec after (at time 2min 25sec) the receiver sends the request to perform the action. At time 2min 28sec action maker 1 (i.e., player 1) generates the decision `P` (i.e., paper) and player 2 `S` (i.e., scissors). At time 2min 43 sec the comparer generates an output stating that player 2 won the round.

```
00:02:00:000 [iestream_input_defs::out: {true}] generated by model play_game_gen
00:02:05:000 [Comparer_defs::playGameOut1: {true}, Comparer_defs::playGameOut2: {true}]
                  generated by model comparer1
00:02:25:000 [RequestReceiver_defs::playGameOut: {true}] generated by model requestReceiver1
             [RequestReceiver_defs::playGameOut: {true}] generated by model requestReceiver2
00:02:28:000 [ActionMaker_defs::gameActionOut: {"P"}] generated by model actionMaker1
             [ActionMaker_defs::gameActionOut: {"S"}] generated by model actionMaker2
00:02:43:000 [Comparer_defs::winReportOut: {2}] generated by model comparer1
```

Figure 12: Simulation log snipped.

The simulation log files can be processed using different techniques to identify, for example, what is the percentage of games won by each player or the percentage a player choose a given action.

5 CONCLUSIONS

In this tutorial, we have reviewed the DEVS formalism, the DEVS-Graph notation, and some of the DEVS simulators available to the community. We have focused on one of those simulators, Cadmium. Through an example, we have explained how to define both atomic and coupled models using the DEVS formalisms and how to implement them using Cadmium. A comprehensive list of DEVS models and their

implementation in Cadmium can be found on our GitHub repository: https://github.com/SimulationEverywhere/Cadmium-Simulation-Environment

REFERENCES

Belloli, L., D. Vicino, C. Ruiz-Martin, and G. Wainer. 2019 "Building Devs Models with the Cadmium Tool", In *Proceedings of the 2019 Winter Simulation Conference (WSC)*, National Harbour, Meriland, USA, , edited by N. Mustafee, K.-H.G. Bae, S. Lazarova-Molnar, M. Rabe, C. Szabo, P. Haas, and Y-J. Son, 45-59. Piscataway, New Jersey: Institute of Electrical and Electronics Engineers.

Bergero F, and E. Kofman. 2011 "PowerDEVS: a Tool for Hybrid System Modeling and Real-Time Simulation". *Simulation*; 87: 113–132.

Chow, A.C., B. P. Zeigler, and D. H. Kim. 1994, "Abstract Simulator for the Parallel DEVS Formalism". *Proc. of the Fifth Conference on AI, Simulation, and Planning in High Autonomy Systems,* Gainesville, FL

Hwang, MH. 2012 X-S-Y. https://code.google.com/p/x-s-y/. Accessed 26th April 2022.

Kim, S., H. Sarjoughian, and V. Elamvazhuthi. 2009. "DEVS-Suite: a Simulator Supporting Visual Experimentation Design and Behavior Monitoring". *In Proceedings of the 2009 Spring Simulation Multiconference*. Art. No. 161.

Nutaro, J. 2011 *Building Software for Simulation: Theory and Algorithms, with Applications in C++*. Hoboken, NJ: Wiley.

Nutaro, J. 2014. *A Discrete EVent System Simulator.* http://web.ornl.gov/~1qn/adevs/adevs-docs/manual.pdf. Accessed 22nd April 2022.

Quesnel,G., R. Duboz, E. Ramat, and M. K. Traore. 2007 "VLE: a Multimodeling and Simulation Environment" in *Proceedings of the 2007 Summer Computer Simulation Conference*, San Diego, CA.

Ruiz-Martin, C., and G. Wainer. 2019. *Cadmium. A Tool for DEVS Modeling and Simulation. User's Guide*. http://www.sce.carleton.ca/courses/sysc-5104/lib/exe/fetch.php?media=cadmium.pdf. Access 26th April 2022.

Sarjoughian, H., and B.P. Zeigler. 1998 "DEVSJava: Basis for a DEVS-based Collaborative M&S Environment". *Simulation* 30: 29–36

Seo, C., B.P. Zeigler, R. Coop, and D. Kim. 2013. "DEVS Modeling and Simulation Methodology with MS4Me Software". In *Proceedings of the 2013 Spring Simulation Multiconference*. pp. 33:1–33:8.

Vangheluwe, H. 2000. "DEVS as a Common Denominator for Multi-Formalism Hybrid Systems Modelling". *In CACSD. Conference Proceedings. IEEE International Symposium on Computer-Aided Control System Design*. IEEE, pp. 129–134.

Van Mierlo, S., Y. Van Tendeloo, and H. Vangheluwe. 2017 "Debugging Parallel DEVS" *Simulatio,* 93(4): 285-306.

Van Tendeloo, Y., and H. Vangheluwe. 2014 "The Modular Architecture of the Python (P)DEVS Simulation Kernel". In *Proceedings of the Symposium on Theory of Modeling & Simulation*, Tampa, FL, USA

Van Tendeloo, Y., and H. Vangheluwe. 2014b "Activity in PythonDEVS" *ITM Web of Conferences,* vol. 3, p. 01002.

Van Tendeloo, Y., and H. Vangheluwe. 2017 "An Evaluation of DEVS Simulation Tools" *Simulation* 93(2): 103-121.

Vicino, D., D. Niyonkuru, and G. Wainer. 2015 "Sequential DEVS Architecture" *Proceedings of the Symposium on Theory of Modeling and Simulation*, Alexandria, VA, USA

Wainer G. 2002 "CD++: A Toolkit to Develop DEVS Models". *Software: Practice and Experience* 32(13): 1261–1306.

Wainer, G. 2009. *Discrete-Event Modeling and Simulation: A Practitioner's Approach*. CRC Press, Boca Raton, FL, USA

Zeigler, B.P., H. Praehofer, and T. G. Kim 2000 *Theory of Modeling and Simulation: Integrating Discrete Event and Continuous Complex Dynamic Systems*, San Diego, CA: Academic Press

AUTHOR BIOGRAPHIES

CRISTINA RUIZ MARTIN, PhD, is an Instructor at the Department of Systems and Computer Engineering at Carleton University. Her research interests are in the area of Modeling and Simualtion. Her email address is cristinaruizmartin@sce.carleton.ca

GABRIEL WAINER, PhD, is a Full Professor at the Department of Systems and Computer Engineering at Carleton University. He is a Fellow of SCS. His research interests are in the area of Modeling and Simualtion His email address is gwainer@sce.carleton.ca