**Assignment 1**

**Cadmium Simulation by Adapting a Rock Paper Scissors CD++ Model**

**Part 1: Model Selection**

Diagram

Description automatically generatedThis assignment will cover the transformation of a CD++ model to be adapted in Cadmium. The model to be adapted is a CD++ simulation of a Rock-Paper-Scissors game with two players. Refer to the block diagram below for a representation of the models contained within.

There are three distinct atomic models, and two coupled models present within this DEVS model, constituting a complexity level of 2.

|  |  |  |
| --- | --- | --- |
| Model Name | Type | Purpose (Brief) |
| Comparer | Atomic | Referee role: Receives input to begin game and signals players to produce an output (rock, paper, or scissors). Upon receiving an input from each player, decides the victor, outputting a winReportOut. |
| Request Receiver | Atomic | Simulates the thought process within a human, and when the thought process turns from passive to active, sends an action request to Action Maker. |
| Action Maker | Atomic | Produces an action output to send back to Comparer, in the form of a selection from rock, paper, or scissors. |
| Player | Coupled | Contains request receiver and action maker, two models that simulate a typical human player. |
| Rock Paper Scissors Game/Top Model | Coupled | Contains the two player coupled models, as well as the comparer atomic model. |

*Data Structures:*

There are a few data structures that will be used for testing and passing data between our models. Please refer to the table below for more details.

|  |  |  |
| --- | --- | --- |
| Structure Name | Variable (Type) in Structure | Purpose |
| PlayGame\_t | isTriggerGame (bool) | Signify if a game request has been sent or received; true for sent, false for not sent. |
| GameAction\_t | choice (int) | Rock (1), paper (2), or scissor (3) choice made by player |
| WinReport\_t | winner(int) | Winner player id stored here (1 or 2). |

**Part 2: Model Description and DEVS Formalism with Sample Test Cases**

*Comparer (Atomic):*

The Comparer atomic model begins the game upon receiving a trigger signal input of type PlayGame\_t 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 players have made their choice, it is entered into the Comparer sub-model through the gameAction1/gameAction2 input ports, of type GameAction\_t. In this case, the ta or time advance to an internal transition is 15 seconds, since we are simulating how long a referee would take to make a decision. After 15 seconds, the Comparer will make a decision on the victor using logic coded inside the model. The output port, winReportOut, will deliver the final result of who the winner is once the output function is triggered (0 signifies a tie). This is stored as type WinReport\_t. Every winner from each round will be stored in a vector by the name of winnerTracker, and the player who is leading the score with total wins will be displayed in the state variable leading.

Comparer = < X, Y, S, 𝛿𝑖𝑛𝑡, 𝛿𝑒𝑥𝑡, 𝜆, ta >

X = {(playGameStartIn ∈ {true, false}), (gameActionIn1 ∈ {1, 2, 3}), (gameActionIn2 ∈ {1 , 2, 3})};

Y = {(playGameOut1 ∈ {true, false}), (playGameOut2 ∈ {true, false}), (winReportOut ∈ {0 ,1, 2})};

S {(active ∈ {true, false}), (playerResult1 ∈ {-1,1,2,3}), (playerResult2 ∈ {-1,1,2,3}), (received1 ∈ {true, false}), (received2 ∈ {true, false}, (playerIDWin ∈ {-1,0,1,2}), (winnerTracker ∈ {[{1,2},,,]})}, (leading ∈ {0,1,2})}}

\*default state values in code\*

// default constructor

Comparer() {

state.active = false; //set to true when input received

state.playerResult1 = -1; //default setting, meaning no response yet from player

state.playerResult2 = -1; //default setting, meaning no response yet from player

state.received1 = false; //set to true when plyer response received

state.received2 = false; //set to true when plyer response received

state.playerIDWin = -1; //no winner selected yet

state.winnerTracker; //tracks the scores of the past rounds with playerIDs

state.leading=0; //tracks which player has one more games in rounds played

}

𝛿𝑖𝑛𝑡(s){

//return to default settings

state.playerIDDisplay = state.playerIDWin;

state.leadingDisplay = state.leading;

state.active = false;

state.received1 = false;

state.received2 = false;

state.playerIDWin = -1;

state.playerResult1 = -1;

state.playerResult2 = -1;

}

𝛿𝑒𝑥𝑡(s, e, x){

if (get\_messages<typename Comparer\_defs::playGameStartIn>(mbs).size() > 1) {

assert(false && "One game at a time!"); //Make sure more than one game is not started at once

}

//if current game is ongoing, refuse

if (get\_messages<typename Comparer\_defs::playGameStartIn>(mbs).size() == 1) {

//trigger game to start

if (state.active == false) {

state.active = true;

}

}

//receiving player input

if (get\_messages<typename Comparer\_defs::gameActionIn1>(mbs).size() == 1) {

state.active = true;

if (state.active == true && state.received1 == false){

vector<GameAction\_t> player1;

player1 = get\_messages<typename Comparer\_defs::gameActionIn1>(mbs);

state.playerResult1 = player1[0].choice;

state.received1 = true;

}

}

𝜆(s){

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

if (state.active == true && state.received1 == true && state.received2 == true && state.playerIDWin != -1) {

vector<WinReport\_t> report;

report.push\_back(state.playerIDWin);

get\_messages<typename Comparer\_defs::winReportOut>(bags) = report;

}

else if (state.active == true && state.received1 == false && state.received2 == false) {

vector<PlayGame\_t> playerTrigger1;

vector<PlayGame\_t> playerTrigger2;

//trigger players to provide response, tell them game has begun

playerTrigger1.push\_back(true);

playerTrigger2.push\_back(true);

get\_messages<typename Comparer\_defs::playGameOut1>(bags) = playerTrigger1;

get\_messages<typename Comparer\_defs::playGameOut2>(bags) = playerTrigger2;

return bags;

}

return bags;

}

ta(s){

TIME next\_internal;

if (state.active == true && state.received1 == false && state.received2 == false) {

next\_internal = TIME("00:00:05:000"); //referee time to alert players

}

else if (state.active == true && state.received1 == true && state.received2 == true) {

next\_internal = TIME("00:00:15:000"); //referee time to make winning decision

}

else {

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

}

return next\_internal;

}

*Improvements to CD++ Model:*

In the CD++ model, if another input to the model is received while the RequestReceiver is still in the time advance, then the entire model will restart. This causes an issue, as if the model were to keep receiving continuous input, then it would never finish the time advance and execute the state functions or output function. This model stops that from happening, because once the output function has triggered in the Comparer, the internal function will set the active variable to false, with a condition in the output function to only output from the ports when the state is active. As such, the Comparer will only be active until one output has been delivered, or until another game trigger has been received.

Additionally, this comparer model also tracks which player is in the lead out of the current rounds that have been played. This will be mentioned in the state log files.

*Testing Strategy:*

There are three inputs to the Comparer model. The first input, playGameStartIn, triggers the comparer to begin the sequence. At 28 seconds past the game was first initialized, both players should return their choice for the game, from which the Comparer will take 15 seconds to make a decision on which player won the game. Three inputs will be given to each input port, representing three different rounds of rock, paper, scissors. One game takes 43 seconds to complete. The input data for each port can be seen below. Note: The requestReceiver input and Comparer playGameStartIn input ports use the same test file, since the input types are both of type PlayGame\_t.

Graphical user interface, text

Description automatically generated

The expected output is that the first game will be a tie, the second game will be won by player 2, and the last game will be won by player 1. Hence there overall leading player variable will be set to 0, since both players won an equal amount of times. The playerID of the winning player will be output every round.

*RequestReceiver (Atomic):*

The RequestReceiver model is simply meant to imitate the human instinct of receiving a request, pondering about the decision, and then sending a request a notification to the next module to make an action. No action is made in this model. A trigger is received from the input port playGameIn of the type PlayGame\_t. The state of the model is then set to active. After a time advance of twenty seconds, the output function is executed, sending an output message from the port playGameOut to the ActionMaker model of type PlayGame\_t. The internal transition function then executes, changing the state variable sent to true, and resetting the active variable, signifying a request to ActionMaker has been sent.

RequestReceiver = < X, Y, S, 𝛿𝑖𝑛𝑡, 𝛿𝑒𝑥𝑡, 𝜆, ta >

X = {(playGameIn ∈ {true, false})};

Y = {(playGameOut ∈ {true, false})};

S {(active ∈ {true, false}), (sent ∈ {true, false})};

// default constructor

RequestReceiver() {

state.active = false; //true if game request from Comparer has been received

state.sent = false; // true if game request has been sent from RequestReceiver to ActionMaker

}

𝛿𝑖𝑛𝑡(s){

state.active = false;

state.sent = true;

}

𝛿𝑒𝑥𝑡(s, e, x){

if (get\_messages<typename RequestReceiver\_defs::playGameIn>(mbs).size() > 1) {

assert(false && "One request at a time!"); //Make sure more than one request is not made at the same time

}

else {

if (state.active == false) {

state.active = true;

state.sent = false;

}

}

}

𝜆(s){

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

if (state.active == true) {

vector<PlayGame\_t> playerTrigger;

//tell next model to stay ready to make decision

playerTrigger.push\_back(true);

get\_messages<typename RequestReceiver\_defs::playGameOut>(bags) = playerTrigger;

}

return bags;

}

ta(s){

TIME next\_internal;

if (state.active == true) {

next\_internal = TIME("00:00:20:000"); //decision making time

}else {

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

}

return next\_internal;

}

*Improvements to CD++ Model:*

The original simulation only requires the RequestReceiver to have a time advance, or thinking time, of 1 second. However, that is not a legitimate representation of a realistic time to ponder. As such, the thinking time for this model was set to 20 seconds. Also, there is another state variable, sent, that signifies if the request has been sent to the ActionMaker, once the 20 seconds are up.

*Testing Strategy:*

There is only one input port, of type PlayGame\_t to the RequestReceiver model. The only expected out is that after 20 seconds of receiving the input request, the model should output the trigger to ActionMaker to make a request through the playGameOut port.

Graphical user interface, text

Description automatically generated

*ActionMaker (Atomic):*

The ActionMaker model simulates receiving a request to make an action, spending three seconds to simulate human hand motion, (“Rock, Paper, Scissors!”), and present their final decision. The playGameIn input port of type PlayGame\_t is inputted to the model, and once a decision is made, the gameActionOut carries out a decision of type GameAction\_t, which essentially holds the choice as an integer value: Rock -1, Paper-2, Scissors-3. The internal transition will reset the states to their default values.

ActionMaker = < X, Y, S, 𝛿𝑖𝑛𝑡, 𝛿𝑒𝑥𝑡, 𝜆, ta >

X = {(playGameIn ∈ {true, false})};

Y = {(gameActionOut ∈ {1, 2, 3})};

S {(active ∈ {true, false}), (choice ∈ {-1, 1, 2, 3})};

// default constructor

ActionMaker() {

state.active = false; // set to true once game request received

state.choice = -1; //default is -1, means no choice, otherwise 1-3

}

𝛿𝑖𝑛𝑡(s){

state.active = false;

state.choice = -1;

}

𝛿𝑒𝑥𝑡(s, e, x){

if (get\_messages<typename ActionMaker\_defs::playGameIn>(mbs).size() > 1) {

assert(false && "One request at a time!"); //Make sure more than one request is not made at the same time

}

else {

if (state.active == false) {

state.active = true;

std::random\_device rand;

std::mt19937 generate(rand());

std::uniform\_int\_distribution<> distribute(1, 3);

state.choice = distribute(generate); //generate value between 1-3

}

}

}

𝜆(s){

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

if (state.active == true) {

vector<GameAction\_t> gameChoice;

//output choice

gameChoice.push\_back(state.choice);

get\_messages<typename ActionMaker\_defs::gameActionOut>(bags) = gameChoice;

}

return bags;

}

ta(s){

TIME next\_internal;

if (state.active) {

next\_internal = TIME("00:00:03:000"); //time to provide signal

}else {

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

}

return next\_internal;

}

*Improvements to CD++ Model:*

The original simulation only requires the ActionMaker model to have a time advance, or time to make an action, 0.5 seconds. This does not represent a real rock, paper, scissors game accurately, so the time advance was set to 3 seconds for this model.

*Testing Strategy:*

There is only one input port, of type PlayGame\_t to the ActionMaker model. After the time advance of three seconds, the model will output an integer choice back to the Comparer model through the gameActionOut port. Once again, the same input file as before is used.

Graphical user interface, text

Description automatically generated

*Player (Coupled):*

There are two player coupled models present in this simulation, Player1 and Player2. The player model contains the RequestReceiver and ActionMaker atomic models, with a total of one input from Comparer to signify the game has started of type PlayGame\_t, and an output returning to Comparer with the player’s choice of type GameAction\_t.

Player = < X, Y, D, {Mi}, IC, EIC, EOC, select >

X = {(playGameIn ∈ {true, false})};

Y = {(gameActionOut ∈ {1, 2, 3})};

D = {RequestReceiver, ActionMaker};

Mi = {MRR, MAM};

IC = {playGameOut@RequestReceiver 🡪 playGameIn@ActionMaker};

EIC = {playGameIn 🡪 playGameIn@RequestReceiver};

EOC = {gameActionOut@ActionMaker 🡪 gameActionOut};

*Testing Strategy:*

There is only one input port, of type PlayGame\_t to the Player coupled model (which directly feeds to the requestReceiver atomic model inside). Once the transition is performed to ActionMaker, the output should be of type GameAction\_t. The Player model should take 23 seconds to output a choice. The input file is shown below:

Graphical user interface, text

Description automatically generated

*Top Model (Coupled):*

The top model ties all models together as shown in the diagram at the beginning of this report. This includes the two player coupled models and the comparer atomic model.

Player = < X, Y, D, {Mi}, IC, EIC, EOC, select >

X = {(playGameStartIn ∈ {true, false})};

Y = {(winReportOut ∈ {0, 1, 2})};

D = {Player1, Player2, Comparer};

Mi = {MP1, MP2, MC };

IC = {playGameOut1@Comparer 🡪 playGameIn@Player1, playGameOut2@Comparer 🡪 playGameIn@Player2, gameActionOut@Player1 🡪 gameActionIn1@Comparer, gameActionOut@Player2 🡪 gameActionIn2@Comparer};

EIC = {playGameStartIn 🡪 playGameStartIn@Comparer};

EOC = {winReportOut@Comparer 🡪 winReportOut};

*Testing Strategy:*

Testing this coupled model is synonymous to testing the entire sequence and simulation. The total time taken by the top model should be 43 seconds and required an input of type PlayGame\_t (feeding into Comparer), and an output of type WinReportOut\_t (feeding out of Comparer). The input file is shown below, with three rounds starting 2 minutes apart from each other.

Graphical user interface, text

Description automatically generated

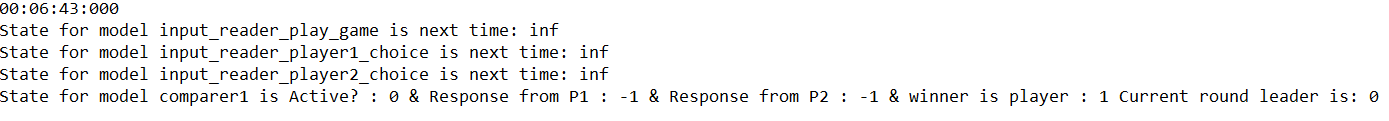
**Part 3: Model Simulations and Results**

*Comparer (Atomic):*

Calendar

Description automatically generated with low confidence

Looking over the simulation results, the results were as expected. The time for each round is 43 seconds, the first round was a tie, and the later rounds were won by player 2 and then player 1 respectively. Hence the overall leader is 0, since each player won the rounds an equal amount of times. Refer to *Current round leader* in the output below.



*RequestReceiver (Atomic):*

A picture containing text

Description automatically generated

Simulation results were correct, since after initialization, the playGameOut port was triggered after twenty seconds. Likewise, the state variables were reset appropriately.

A screenshot of a computer

Description automatically generated with medium confidence

*ActionMaker (Atomic):*

Text

Description automatically generated with medium confidence

Simulation results were correct, since after initialization, the gameActionOut port was triggered after three seconds with a random value between 1-3. Likewise, the state variables were reset appropriately after triggering the internal transition function once the time advance was complete.

Text, letter

Description automatically generated

*Player (Coupled):*

The total simulation time was 23 seconds as expected, with the RequestReceiver and ActionMaker taking their expected time to transition. The output port, gameActionOut was also verified, with a random number generated between 1-3 each round. This ensured all external and internal couplings were done correctly.

Text

Description automatically generated with medium confidence

*Top Model (Coupled):*

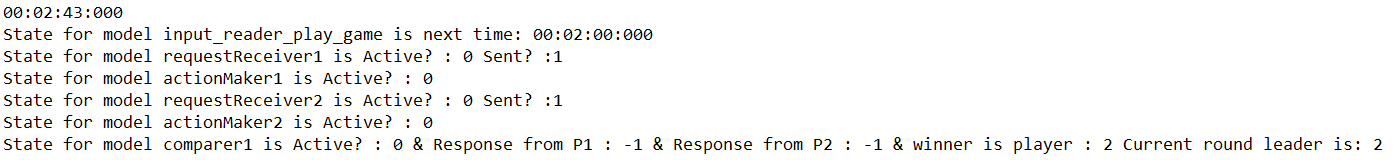
This the main simulation that covers the entire three rounds of the Rock Paper Scissors Game. The total time for each round was 43 seconds, and each started at 00:02:00, 00:04:00, and 00:06:00, as mentioned before. The state variables were reset where required to not affect future simulations as expected. The current round leader was calculated after every time the Comparer chose a round winner, to display which player was in the lead.

*Round 1:*

Text

Description automatically generated with medium confidence

* Player 2 won the first round, hence, player 2 is in the lead

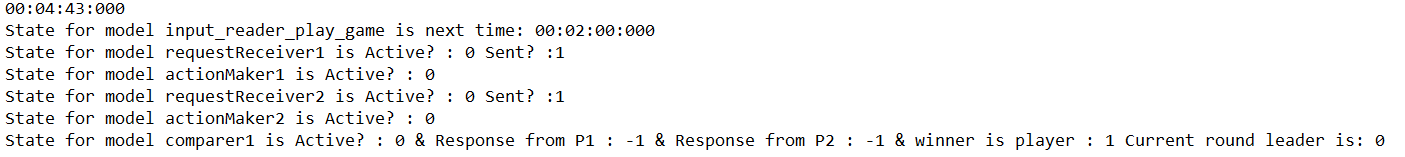


*Round 2:*

Text

Description automatically generated

* Player 1 won this round, so no player is in the lead (0)



*Round 3:*

Text

Description automatically generated with medium confidence

* Both players, picked the same choice so round was tie, meaning no one is in the lead still
* Last round, end of simulation

