**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 winReport. |
| 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) | 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: Atomic Model Description and DEVS Formalism**

*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.

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})}

\*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

}

𝛿𝑖𝑛𝑡(s){

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

if (state.playerResult1 == state.playerResult2) {

state.playerIDWin = 0;

}

else if (state.playerResult1 == 1 && state.playerResult2 == 2) {

state.playerIDWin = 2;

}

else if (state.playerResult1 == 1 && state.playerResult2 == 3) {

state.playerIDWin = 1;

}

else if (state.playerResult1 == 2 && state.playerResult2 == 1) {

state.playerIDWin = 1;

}

else if (state.playerResult1 == 2 && state.playerResult2 == 3) {

state.playerIDWin = 2;

}

else if (state.playerResult1 == 3 && state.playerResult2 == 1) {

state.playerIDWin = 2;

}

else if (state.playerResult1 == 3 && state.playerResult2 == 2) {

state.playerIDWin = 1;

}

}

}

𝛿𝑒𝑥𝑡(s, e, x){

//trigger game to start

if (state.active == false && state.playerIDWin == -1) {

state.active = true;

}

//receiving player input

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

vector<GameAction\_t> player1;

vector<GameAction\_t> player2;

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

player2 = get\_messages<typename Comparer\_defs::gameActionIn2>(mbs);

state.playerResult1 = player1[0].choice;

state.received1 = true;

state.playerResult2 = player2[0].choice;

state.received2 = true;

}}

𝜆(s){

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

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

vector<WinReport\_t> report;

report.push\_back(state.playerIDWin);

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

return bags;

}

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;

}

}

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;

}

*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, 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){

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

state.sent = true; //variable to signify request has been sent

}

}

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;

}

}

}

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;

}

*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.

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){

if (state.active == true && state.choice == -1) {

state.choice = 1 + (rand() % 3); //generate value between 1-3

}

}

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;

}

}

}

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;

}

**Part 3: Coupled Model Description and DEVS Formalism**

*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};

*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};