Modeling and Controlling a Robotic Arm with E-CD++

Faezeh Rafsanjani Sadeghi, Gabriel Wainer, Mohammad Moallemi
Department of Systems and Computer Engineering, Carleton University
Ottawa, ON, Canada
frsadegh@connect.carleton.ca , {gwainer, moallemi}@sce.carleton.ca

ABSTRACT

The E-CD++ tool uses the RT-DEVS (real-time DEVS) formalism for modeling, simulation and execution of real-time and embedded applications. This formal modelling and simulation approach can be used as a robust foundation for developing real-time and embedded applications. It eases verification of the product, as sometimes verifying an embedded application in the real environment can be very risky or impractical. We show the use of this methodology to model a real-time robotic application. We show the ease of integrating a simulated DEVS model using E-CD++, embedding it in a robotic device (consisting of a robotic arm with a claw to grab and load objects). The complete model design is done through DEVS graphs based on DEVS formal model specifications. E-CD++ allows one implement and execute the model on a RTLinux kernel.

1. INTRODUCTION

Real-Time System (RTS) design and implementation is a challenging process, the construction and verification of these systems need a tremendous effort because of the high risk factors existing in the environments these systems control. A RTS needs to be thoroughly tested before they can be integrated to the real environment they control, and Modeling and Simulation (M&S) methods offer a robust and cost-effective method to do it. Testing a virtual simulation of the actual RTS is easier, cheaper and less risky than testing on the target platform. M&S combined with formal methods and graphical notations can be used to test the system on a virtual mirror of the actual world.

The use of a formal methodology like the Discrete Event System Specification (DEVS) formalism [1] can make this task even simpler. DEVS can be used to model any discrete system (for instance, those including RT constraints), while formally proving properties about the models developed and the execution engines. We show how this methodology has been used to build a DEVS-based robotic arm system. The robot application is first modeled using DEVS, and it is then simulated it in the CD++ toolkit [2] (a software tool for simulation of DEVS models). Once the application is completely developed on the simulated environment, one can start deploying the actual software application in the target platform by running it on the E-CD++ tool [3]. After a second simulation round on the RT environment, is deployed on to the actual robotic arm for RT execution. We first show the design of this system using DEVS state diagrams and the importance of this stage in the development of the system. Then, the model implementation in CD++ is discussed in detail, and the testing and simulation in the virtual environment is analyzed. After, we show the actual execution results on the target platform.

2. BACKGROUND

The DEVS formalism is a systems-theoretical notation based on the concept of hierarchical model development. A DEVS model can be seen as composed of two parts: structural (coupled) and behavioural (atomic) models. Coupled models interconnect other models (coupled and atomic) while atomic models perform the actual processing task. Atomic models are represented by DEVS state diagrams. Figure 1 shows the formal behaviour of an atomic model. Each state has a duration which is controlled by the time advance function *ta(s)*. When this time expires the model can produce an output using the ? function, and the next state is calculated using the internal transition function d_{int}. When the system receives an input, the external transition function d_{ext}, is invoked to determine how the internal state will change. Parallel DEVS (P-DEVS) [4] allows dealing with simultaneous events using a confluent function d_{con} which is activated when an external and an internal event occur at the same time. Inputs and outputs at the same time are stored into I/O bags, which save all the simultaneous inputs and outputs for the model.

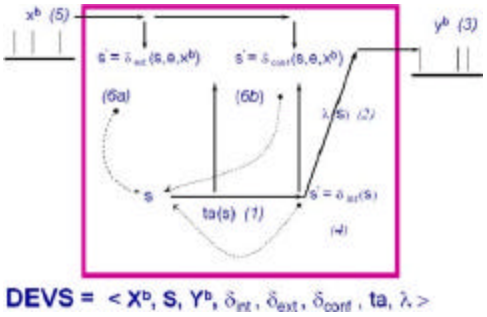


Figure 1. P-DEVS Atomic Model [1]

The E-CD++ toolkit uses an extension to parallel-DEVS, called Real-time DEVS or RT-DEVS [5], which uses a real-time advance function and the concept of driver for Top coupled model ports for interconnection with the environment. The model is therefore defined as:

$$RTAM = < X, S, Y, d_{ext}, d_{int}, d_{con}, ?, ta >,$$

Where *X, S, Y,* d_{ext}, d_{int}, d_{con} and *?* are the same as P-DEVS, and

ta: S ? $R^{+}_{0,8}$ uses the actual wall clock time.

Coupled models are depicted in Figure XXX. Atomic models A1 and A1 construct coupled model A2, while A2, is coupled with A3 and A4 to make the top coupled model A5.

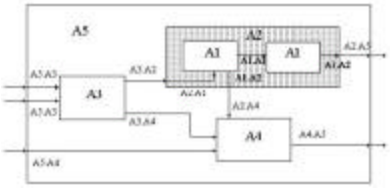


Figure 2.Figure Example of Structural model [2]

3. Related Works

Model based approaches are considered reliable and promising methods for developing real-time and embedded applications. The methodology introduced in [6] demonstrates the potential of Model-Integrated Computing in providing a unified environment for multi-granular simulation of embedded systems. The authors illustrate many issues in computer automated multi-language modeling, using the Model-based Integrated Simulation Framework (MILAN) project as a vehicle. UML class diagram-based meta-models along with OCL constraints are used to define the syntax and static semantics of a highly domain-specific modeling language. Meta-model composition techniques were used to combine different modeling formalism, such as synchronous and asynchronous dataflow, data type systems, hardware architecture and behaviour modeling. We also demonstrated separation of concerns with multiple aspects, and how it could be utilized effectively in managing design complexity.

There have been few attempts to use DEVS formalism in real-time and embedded control applications.

In [7] a software development methodology for dynamic distributed real-time systems was presented. The methodology is based on DEVSJAVA modeling and simulation environment. It supports model continuity so that a dynamic distributed real-time system can be designed, analyzed and tested by simulation methods, and then migrated to execute in a distributed network while preserving its control models. To handle the dynamic properties of a distributed real-time system, the variable structure modeling capability is integrated into the proposed methodology. Stepwise simulation methods such as central simulation, distributed simulation, and hardware-in-the-loop (HIL) simulation are developed to incrementally test the control models in a virtual environment. A distributed robotic "team formation" example was developed and presented in the paper to demonstrate how this dynamic system can be developed by applying the proposed methodology in different stages.

In [8], RT-DEVS/CORBA, is presented as a modeling and simulation framework, to support the development of distributed real-time systems. The framework supports model continuity for real-rime software development from model design to performance evaluation and even to final real-time control. This approach is based on RT-DEVS formalism and maps activities to each state. The authors do not mention details about real-time control part and the focus is on real-time simulation and a case study is presented.

In [9] a hybrid methodology has been developed for integrating different types of DEVS models using a Knowledge Interchange Broker (KIB). A supply-chain semiconductor application is describe where the KIB has been used as an integral part of developing and deploying a commercial Model Predictive Control model for use in operating a semiconductor manufacturing supply chain. The simulation based experiments facilitated developing and validating the controller design and data automation for a real-world semiconductor manufacturing system.

In this paper the design and implementation of a real-time and embedded application is presented, which is backed by DEVS formalism as a robust mathematical background. The E-CD++ software developed in our lab let us overcome the hardware integration limitation by providing open-source interface development environment. The interface development environment enables the designer to integrate any hardware or external environment driver with the DEVS run-time engine, in order to embed the final product in the real-environment.

The other advantage of this method is the ease of verification (a critical challenge in hard real-time systems) which can be done in virtual-time or real-time using simulated inputs injected to the system and verifying the outputs and system behaviour.

4. ROBOT ARM FORMAL SPECIFICATION AND MODELS

DEVS modeling allows for modeling and simulating systems before using them in a RT environment. One of the primary steps in developing any system is to analyze the requirements for designing the system. GGAD diagrams are used to show states and transitions between the different states. If we refer to figure one which is the atomic model in a DEVS, each state has a specified duration and after that duration the state changes to the next state and produces an output, and if the state is interrupted by an external event, an external transition function determines the behaviour of the system and the next state transition. In a GGAD diagram these transition are shown and formalized. Each state is shown by a circle and the initial state is shown by a double circle, and the duration of each state is shown inside the circle as well. The dotted lines are used to show

the internal transition function, and the required output is shown on each transition. The external transition function is shown by solid lines and the required input from an outside port is shown on top of each line.

The robot arm is composed of 5 different atomic models. The structural model is shown in figure XXX.

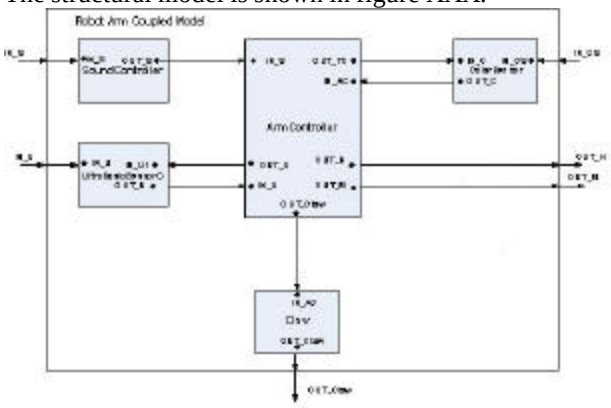


Figure xxx. RobotArm Structural Diagram

The model is composed of five atomic models which are ArmController, SoundController, UltrasonicSensorC, ColorSensor and Claw. The sound controller waits until it gets an input from the sound sensor and it sends it to arm controller in order for the system to start. The arm controller is responsible for taking in values from each of the sensor and acting accordingly with the values it has received. The ultrasonic sensor is responsible for sending values to the ArmController to determine if the robot has found a ball. The color sensor is then activated to tell the ArmController that it has found a blue or red ball and the ArmController using this value tells the claw to either grab the ball or not. The inputs and outputs to the model are shown by arrows.

4.1 Formal Specification:

The formal specification for each of the models is included in this section.

3.1.1 SoundController

M = <X, S, Y, d_{ext}, d_{int},?, ta>, where:

X: IN_S

S: initial, sendToArm, stop

Y: OUT_S

d_{ext}: Receives inputs from the input port and initiates appropriate state transitions.

d_{int}: defines state changes based on current state.

?: sends value of 1 to ArmController when a value greater than 25 is received from the sound sensor

ta: RT advance function for each state.

3.1.2 ColorSensor

M = <X, S, Y, d_{ext}, d_{int},?, ta>, where:

X: IN_C, IN_CS

S: idle, sensing, send

Y: OUT_C

d_{ext}: Receives inputs from the input port and initiates appropriate state transitions.

d_{int}: defines state changes based on current state.

?: send the value of 1 if color is blue and 2 if color is red

ta: RT advance function for each state.

3.1.3 UltraSonicSensorC

M = <X, S, Y, d_{ext}, d_{int},?, ta>, where:

X: IN_U1, IN_U

S: idle,checksensor,godown,foundball

Y: OUT_U

d_{ext}: Receives inputs from the input port and initiates appropriate state transitions.

d_{int}: defines state changes based on current state.

?: if distance is less than 10 and greater than 2 it sends a 2 and if distance is less than 2 it sends a 3 to ArmController.

ta: RT advance function for each state.

3.1.4 Claw

M = <X, S, Y, d_{ext}, d_{int},?, ta>, where:

X: IN_AC

S: release, grab, idle, done

Y: OUT_Claw

d_{ext}: Receives inputs from the input port and initiates appropriate state transitions.

d_{int}: defines state changes based on current state.

?: sends a 2 to claw if release required, sends 1 if grab is required and sends 3 is stop is required.

ta: RT advance function for each state.

3.1.5 ArmController

M = <X, S, Y, d_{ext}, d_{int},?, ta>, where:

X: IN_S, IN_U, IN_AC.

S: idle, informsensor,goingDown,goDown checkball, informcolor, checkingColor, closeclaw, closingclaw, goUp, stop

Y: OUT_U, OUT_TC, OUT_M,OUT_H,OUT_Claw

d_{ext}: Receives inputs from the input port and initiates appropriate state transitions.

d_{int}: defines state changes based on current state.

?: Sends values to ColorSensor and UltraSonicSensor when it requires a value (value is 1 for both), sends outputs

to horizontal and vertical motors when it requires the motor to move. (sends 1 to h when moving left, 0 when stop is required, 1 to arm to move up, 2 to move down and 0 to stop.)

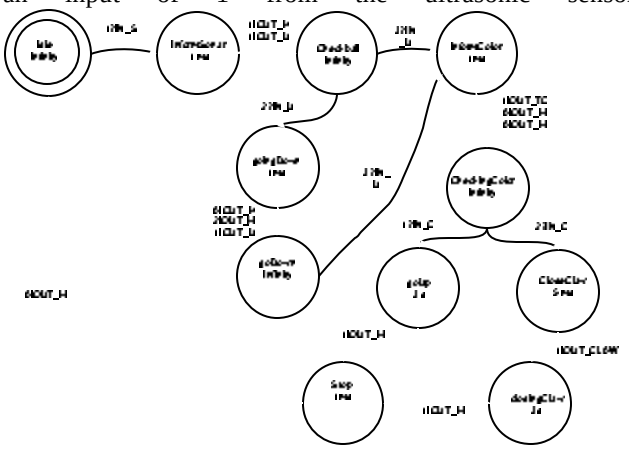
ta: RT advance function for each state.

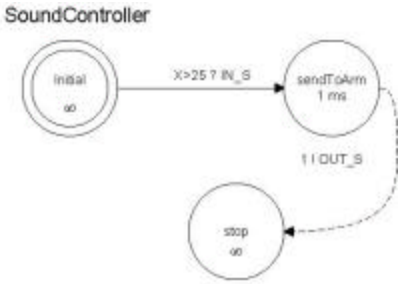
4.2 GGAD Diagrams

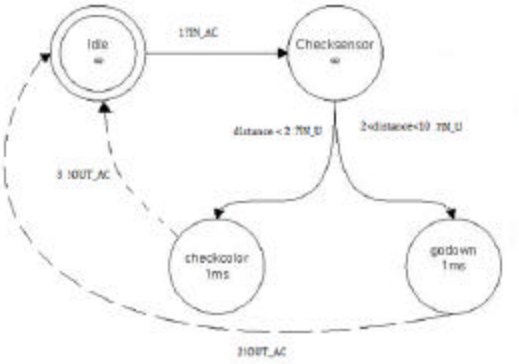
A GGAD diagram is used to show the state transitions for each of the atomic models. This is the most important step in the modeling of the robot arm. Since DEVS is formalism for modeling systems, if the requirements and state transitions are known using this formalism is straight forward. The GGAD diagram uses circles to represent states in the atomic model. To show an internal transition dotted lines are used and the output of the state is also shown in the diagram. The duration of the particular state is also shown inside the circle. Solid lines between states are used for external transition function which happens when an input is received and the state is changed.

The state diagram for each of the atomic models has been produced in figures xxx.

Figure xxx shows the diagram for the ArmController model. The arm controller is in idle state for an infinite amount of time until it receives an input from the ultrasonic sensor. When a value of 1 is received from the ultrasonic sensor the arm goes into the informsonar state. The duration of this state is 1 ms, which means that after 1 ms, an output of 1 is sent to ultrasonic sensor and a value of 1 is sent to start horizontal movement. The ArmController then goes to the next state which is checkball and waits for an input of 1 from the ultrasonic sensor.







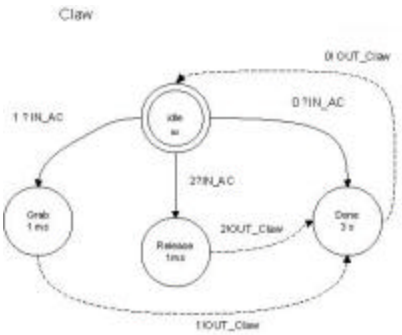


Figure www. Claw GGAD diagram

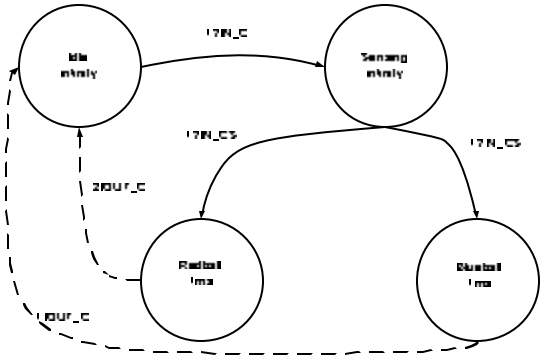


Figure zzz. ColorSensor GGAD diagram

4.3 CD++ Coding

After the structural model and the GGAD diagrams are drawn, the CD++ code can be written to simulate the model in a virtual environment. As explained the internal transition functions are shown by dotted lines and the external transition functions are shown with solid lines.

This step in the modeling should be straight forward if the GGAD diagram is drawn correctly. CD++ is a formalized language, allowing users to map their design to the CD++ structure.

CD++ is implemented using C++ language in an eclipse based environment. The time for each of the states can be defined as a Time variable, in virtual time. Each atomic model has an **initFunction**, used for initialization of the atomic model, an **internalFunction** used for the internal transition function of the model, an **externalFunction** used for the external transition function and an **outputFunction**, used for the output function. The holdIn() function is used to define how long the system stays in a particular state and it the system should stay in that state for a long period of time, the passivate() function is used. When the system receives an input from an external source, i.e. External event, the state is changed. Using the **registerNewAtomic** function, each atomic model is registered and these models each contain input and output ports which can be connected using an .ma file. The .ma file connects all the models through their ports using the Link function.

After all the atomic models, their internal, external and output functions and links are defined, an event file can be defined by the user. The external event and the time at which they are sent is included in the event file.

In order to simulate the behaviour of the model different event files were created. The models were tested individually to ensure correct results.

4.4 E-CD++

As mentioned in section 2, after the model has been formalized using GGAD diagram, and the implementation of the model in CD++ the model can be transferred to E-CD++, a language used for RT modeling of systems.

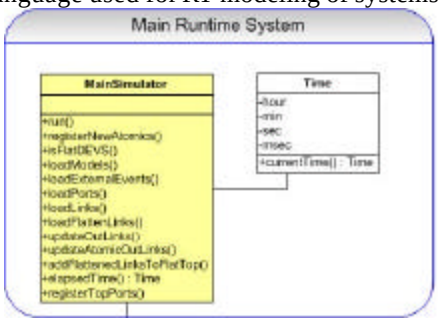


Figure3.2.1 Main Runtime System Class Diagram [2]

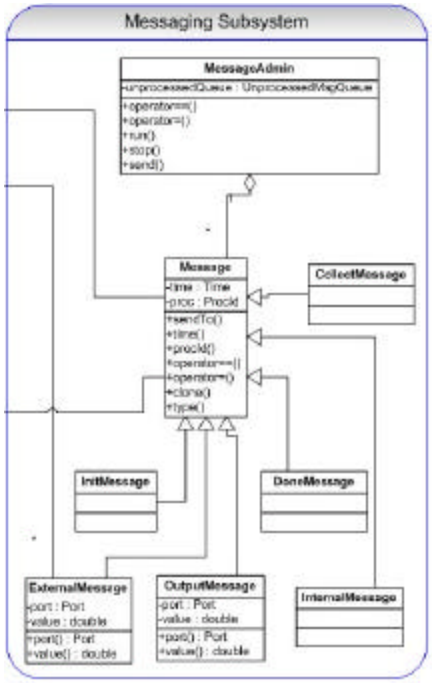


Figure3.2.2. Messaging Subsystem Class Diagram [2]

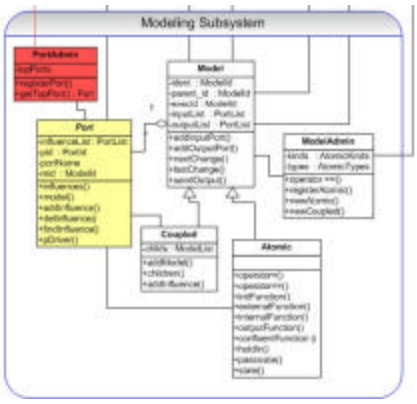


Figure3.2.3. Modeling Subsystem Class Diagram [2]

Figure 3.2.4, shows the connection between all the class diagrams for E-CD++.

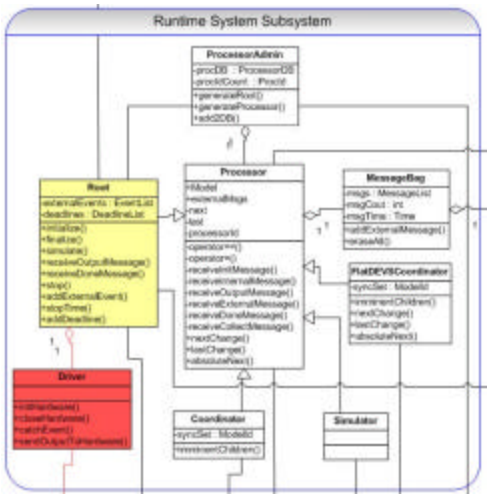


Figure3.2.4. Runtime System Subsystem Class Diagram [2]

E-CD++ is composed of four main components: Main Run Simulator (Figure 3.2.1), Modeling Subsystem (Figure 3.2.2), Runtime System Subsystem (Figure 3.2.3) and Messaging Subsystem (Figure 3.2.4).

The Main Run simulator, as the name suggests is the main program that runs. In this component, the Atomic model objects are registered, the external events are read and an external events table is built, the model file is read and a model hierarchy is built and a Root Coordinator is created and triggers it to run.

The Runtime System Subsystems consists of coordinators, and processors manager. The processor class objects are managed by the Processor Manager, which is implemented by the ProcessorAdmin class.

The processor class implements the DEVS Realtime System framework.

The Messaging Subsystem consists of Message manager class and other message objects. Processors and coordinators send messages via the Messages Manager, which is implemented by the MessageAdmin class.

The Modeling subsystem is a logical representation of the DEVS models defined by the modeller. It is composed of the Models manager and the DEVS Models Hierarchy tree. The Models manager manages the models hierarchy [2].

This part can be modified by the used to connect the model to the hardware. The ports in the model developed earlier can be connected to the ports defined in E-CD++.

When transferring the model from CD++ to ECD++, ports have to be defined using the Port class. In the case of the robot, these ports include the ports to the sensors and motors. The ports should also be registered in the register file, using registerPort() function in PortAdmin. To be able to define ports, each of port can be defined in a header file and the port extends the port class. The specific port number on the microcontroller of the robot can be defined using inithardware() and extending the pDriver class, specific instructions can be given fir each port.

The functions that can be used to get the values from the hardware are, GetValue(port number) and GetSonarValue(port number), for ports that are ultrasonic.

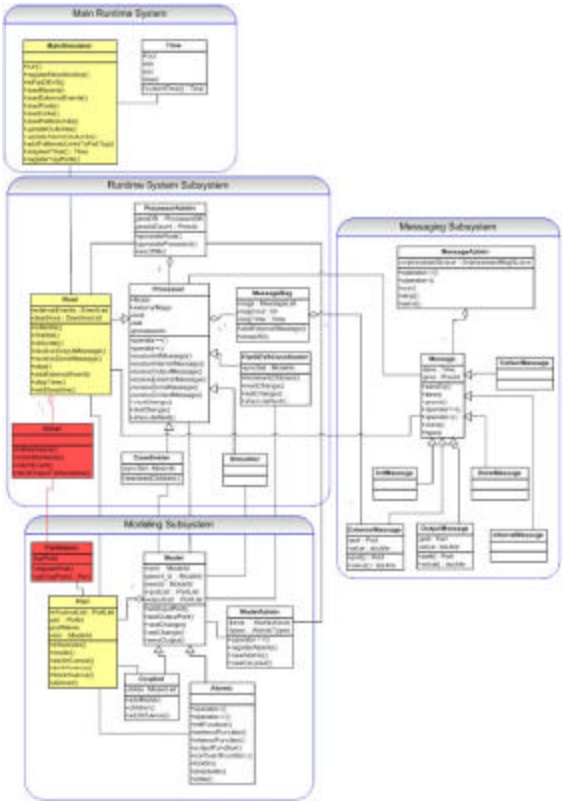


Figure3.2.5. Class Diagram of E-CD++[2]

The color sensor and the ultrasonic sensor, the GetSonarValue() is used.

To be able to have horizontal and vertical movement in the arm and movement in the claw, the Stop(),SetReverse(), and SetForward() command can be used. These commands can be sent to the port that is desired. For example if port B is used for horizontal motor and the motor has to go left, the command SetReverse(B,90), can be used. In the robot arm example, there were six header files, three for the sensors, color, ultrasonic and sound; two for the horizontal and vertical movement for motors and one for the claw. These files are simple header files which are ports and use the pDriver class.

The next step is to define the specific functionality for each of the ports. The nxtdriver.cpp file is used to define the specific functionality of each port. As discussed earlier the GetSonarValue(port number) is used to get inputs from the color and ultrasonic sensor, and the GetValue(port number) is used to get inputs from the sound sensor. The programmer can send only the required values to the model. The Stop, SetReverse and SetForward functions are used to the motors according to the specific value sent by the programmer. For example, if the programmer sends a 1 to the horizontal the motor has to move left, so the command SetReverse(port, 90) is used.

5. SIMULATION AND RESULTS

To simulate the models designed for the robot arm, the models had to be tested individually first, using CD++ eclipse simulator. The models were also tested as a whole using this simulator as well. The next step was to transfer these results to E-CD++ and test them using this simulator.

E-CD++ allows two types of simulation, one is in virtual time and the other is in RT. After the models have been tested in both environments, the models can then be tested on the actual environment which is the robot.

The makefile in the E-CD++ has to be modified to include the robot arm model. The make command is used to prepare the simulator for running the model. the following commands can be used to simulate results for virtual, RT and hardware, in order.

../../simu –m –e -o –t –l

The m is for the ma file, the e is for the event file, the o is for the output file, the t is for simulation time and l is for log file

../../simu –m –e –o –r –l –W

This command is similar to the virtual time, the difference is that a –w is added for RT and the event file has to be modified. The event file contains a RT and a deadline for the task to be completed.

../../simu –m –e –o –t –l –w –g

This command is used for executing the model on the hardware. There is no event file and the –g is added for the time at which signals are received from hardware, i.e. sensors.

The testing strategy is to test each of the atomic models separately, then test the models as a whole in CD++, move the models and test them in virtual and RT in E-CD++. The last step would be to connect the robot and execute the code on the robot to see the final results.

This section includes testing of each unit and the results in CD++ and ECD++ in virtual and RT.

5.1 Testing of different units:

5.1.1 Testing SoundController

5.1.1.1 CD++

The model to be tested is the soundController to make sure that when a sound is heard (greater than 25 decibels in this case), the soundController sends an output to the armController to start execution. The input to the soundController atomic model is IN_S as seen in figure 2, and the output is OUT_S to the arm controller. The event file and the .ma file are shown in figures 4.1.1.1.1 to 4.1.1.13.

```
[top]
components : C@SoundController

in : IN_S
out : OUT_S

Link : IN_S IN_S@C
Link : OUT_S@C OUT_S


[C]
soundprocessingtime : 00:00:00:01
```

Figur4.1.1.1.1. SoundController.ma file

```
00:00:05:00  IN_S  2
00:00:07:00  IN_S  3
00:00:10:00  IN_S  25
```

Figure4.1.1.1.2 Events to test SoundController

```
00:00:10:001 out_s 1
```

Figure4.1.1.1.3. Output file for sound controller

To test Sound Controller we have provided an event file with external events defined. At time 5 and 7 there are inputs of 2 and 3 into the system, but since these are less than 25 we see that there is no output produced in figure 10. But at time 10 there is an input of 25 to the soundController and therefore the value of 1 is sent through the port OUT_S after 1 ms. This shows that the correct result is produced.

5.1.1.2 E-CD++

The files used are the same for testing the unit. When testing in virtual time we get the exact same results as

CD++, therefore they are not shown. The results in RT however are shown here since they are different and have to be tested. Figure 15 shows the RT event file used to test SoundController. The deadline can be any time and it doesn't have to be exact because we are not dealing with RTSs, where deadlines have to be met. A "dummy" output port has to be defined as well. OUT_S has been used as the output port in this case.

```
00:00:05:00  00:00:06:00 IN_S  OUT_S 2
00:00:07:00  00:00:08:00 IN_S  OUT_S 3
00:00:10:00  00:00:11:00 IN_S  OUT_S 25
```

Figure4.1.1.2.1. Realtime Event file for SoundController '

```
Cur Time: 00:00:10:001 Deadline: 00:00:06:000
(NOT succeeded) OutPort: out_s PortValue: 1
```

Figure4.1.1.2.2. Realtime output file for SoundController

The results can be seen in figure 4.1.1.2.2 As we can see the results obtained are the same as in virtual time, so this unit is functional and we can move on to the next unit for testing.

5.1.2 Testing UltraSonicSensorC

5.1.2.1 CD++

The UltraSonicSensorC model is responsible for sending values received from the ultrasonic sensor to the ArmController according to the distance it has received. As seen in figure 2, the UltraSonicSensorC has two inputs and one output port. The input ports are from the ArmController and the ultrasonic sensor. When the model receives a 1 from the ArmController it has to send a value received from the sensor to the ArmController. If the UltraSonicSensorC receives a value that is greater than 10, it sends a value of 1 to the ArmController, meaning that it is still very far from the ball and the arm has to move horizontal. If the distance is less than 10 and greater than 2 it means the arm is not close enough to the ball therefore it has to move down and a value of 2 is sent to the armController. When the arm is close enough to the ball the value is less than 2cm, therefore the arm has to stop and a value of 3 is sent to the arm controller. Figures 4.1.2.1.1 to 4.1.2.1.3 show the .ma, event and output files for the tested results.

```
[top]
components : U@UltraSonicSensorC

in : IN_U
in : IN_U1
out : OUT_U
Link : OUT_U@U OUT_U
Link : IN_U IN_U@U
Link : IN_U1 IN_U1@U

[U]
UProcessingtime : 00:00:00:01
```

Figure4.1.2.1.1. UltraSonicSensorC .ma file

```
00:00:05:00 IN_U1 1
00:00:06:00 IN_U 19
00:00:07:00 IN_U 2
00:00:08:00 IN_U1 1
00:00:09:00 IN_U 9
00:00:10:00 IN_U1 1
00:00:11:00 IN_U 2
```

Figure4.1.2.1.2 UltraSonicSensorC event file

```
00:00:06:001 out_u 1
00:00:09:001 out_u 2
00:00:11:001 out_u 3
```

Figure4.1.2.1.3. UltraSonicSensorC output file

The UltraSonicSensorC waits for an input from ArmController, and sends the value through port OUT_U. As seen in the event file another input is received at time 0:07 through port IN_U, but this input is ignored because no value is received from the ArmController. The output is 1 since the value of 19 is greater than 10. To show that the rest of the model is functional, other inputs are provided and every time a value of 1 is received from ArmController, the input is from ultrasonic sensor is analyzed to send output to the ArmController. Values of 9 and 2 are received and values of 2 and 3 are sent through output port OUT_U, to ArmController.

5.1.2.2 E-CD++

The unit was tested in E-CD++ RT simulation. The event file used is shown in figure 4.1.2.2.1, and the output is shown in figure 4.1.2.2.2. As can be seen the results are the same as virtual time, therefore the unit has been tested and it functions correctly.

```
00:00:05:00 00:00:06:00 IN_U1 OUT_U 1
00:00:06:00 00:00:07:00 IN_U OUT_U 19
00:00:07:00 00:00:08:00 IN_U OUT_U 2
00:00:08:00 00:00:09:00  IN_U1 OUT_U 1
00:00:09:00 00:00:10:00  IN_U OUT_U 9
00:00:10:00 00:00:11:00 IN_U1 OUT_U 1
00:00:11:00 00:00:12:00 IN_U OUT_U 2
```

Figure4.1.2.2.1. Realtime event file to test UltraSonicSensorC

```
Cur Time: 00:00:06:001 Deadline: 00:00:06:000
(NOT succeeded) OutPort: out_u PortValue: 1
Cur Time: 00:00:09:001 Deadline: 00:00:07:000
(NOT succeeded) OutPort: out_u PortValue: 2
Cur Time: 00:00:11:001 Deadline: 00:00:08:000
(NOT succeeded) OutPort: out_u PortValue: 3
```

Figure4.1.2.2.2. Realtime output file for UltraSonicSensorC

5.1.3 Testing ColorSensor

5.1.3.1 CD++

The ColorSensor gets the value from the color sensor and returns it to the ArmController whenever it receives a 1

from ArmController. If the value that it receives is 2, the ball is red and it returns a value of 1 to the ArmController. If the value received from the sensor is 9, the ball is blue and the value of 2 is sent to ArmController. Figures 4.1.3.1.1 to 4.1.3.1.3 show the .ma file, the event file, and the output file for the particular event file.

```
[top]
components : C@ColorSensor

in : IN_C
in : IN_CS
out : OUT_C
Link : OUT_C@C OUT_C
Link : IN_C IN_C@C
Link : IN_CS IN_CS@C

[C]
colorSensedTime : 00:00:00:01
```

Figure4.1.3.1.1. ColorSensor .ma file

```
00:00:05:00 IN_C 1
00:00:05:03 IN_CS 2
00:00:07:00 IN_C 1
00:00:07:02 IN_CS 9
```

Figure4.1.3.1.2. ColorSensor event file

```
00:00:05:004 out_c 1
00:00:07:003 out_c 2
```

Figure4.1.3.1.3. ColorSensor output file

The model returns a value every time ArmController requests a value. In the first case the color sensor sends a value of 2 therefore the ball is red and a value of 1 is sent to ArmController. In the second case a value of 9 is sent, meaning the ball is blue, therefore a value of 2 is sent to ArmController. The results are therefore correct.

5.1.3.2 E-CD++

The ColorSensor was tested in E-CD++ using the event file shown in figure 4.1.3.2.1, and the results are shown in figure 4.1.3.2.2. The results are the same as in CD++ results and it behaves correctly when given an input.

```
00:00:05:00 00:00:05:30 IN_C OUT_C 1
00:00:05:03 00:00:06:00 IN_CS OUT_C 2
00:00:07:00 00:00:07:06 IN_C OUT_C 1
00:00:07:03 00:00:08:00 IN_CS OUT_C 9
```

Figure4.1.3.2.1. Realtime event file to test ColorSensor

```
Cur Time: 00:00:05:008 Deadline: 00:00:05:030
(Succeeded) OutPort: out_c PortValue: 1
Cur Time: 00:00:07:008 Deadline: 00:00:06:000
(NOT succeeded) OutPort: out_c PortValue: 2
```

Figure4.1.3.2.2. Realtime output file for ColorSensor

5.1.4 Testing Claw

5.1.4.1 CD++

When the claw receives a value of 1 from ArmController it has to grab the ball, when it receives a 2 it has to release the ball and when it receives a 0 it has to stop the claw from moving. Figure 4.1.4.1.*1* to 4.1.4.1.*3* show the .ma file, the event file and the output for the given event file.

```
[top]
components : C@Claw

in : IN_AC
out : OUT_Claw
Link : OUT_Claw@C OUT_Claw
Link : IN_AC IN_AC@C

[C]
clawpreparationTime : 00:00:30:00
grabReleaseTime : 00:00:00:10
```

Figure4.1.4.1.*1*. Claw.ma file

```
00:00:05:00  IN_AC 1
00:00:06:00 IN_AC 2
00:00:11:00 IN_AC 0
```

Figure4.1.4.1.*2*. Claw event file

```
00:00:05:010 out_claw 1
00:00:06:010 out_claw 2
00:00:11:010 out_claw 0
```

Figure4.1.4.1.*3*. Claw output file

As can be seen by the result, the right values are sent to the claw's motor to get the corresponding action.

4.1.4.2 E-CD++

The Claw unit was tested in ECD++ and the same results as CD++ were maintained. 4.1.4.2.*1* shows the event file for testing claw in RT and figure 4.1.4.2.*2* shows the results which are the same as the results in CD++.

```
00:00:05:00 00:00:06:00 IN_AC OUT_Claw 1
00:00:06:00 00:00:07:00 IN_AC OUT_Claw 2
00:00:07:00 00:00:08:00 IN_AC OUT_Claw 0
```

Figure4.1.4.2.*1*. Realtime event file for Testing Claw

```
Cur Time: 00:00:05:010 Deadline: 00:00:06:000
(Succeeded) OutPort: out_claw PortValue: 1
Cur Time: 00:00:06:010 Deadline: 00:00:07:000
(Succeeded) OutPort: out_claw PortValue: 2
Cur Time: 00:00:07:010 Deadline: 00:00:08:000
(Succeeded) OutPort: out_claw PortValue: 0
```

Figure4.1.4.2.*2*. Realtime output file of Claw

5.2 Testing the ArmController

5.2.1 CD++

The next step in testing is to test the model as a whole. The ArmController is the main model that receives inputs from sensors and sends outputs to the motors to act accordingly to the situation.

The arm controller's functionality was discussed in the GGAD diagram, or state diagram. The reader can refer to

section 3.2 for detail.

Figures 4.2.1.*1* to 4.2.1.*3* show the .ma file, the event file and the corresponding output values received.

```
[top]
components : AC@ArmController
components : CL@Claw
components : CS@ColorSensor
components : SC@SoundController
components : USSC@UltraSonicSensorC

in : IN_S
in : IN_U
in : IN_CS
out : OUT_M
out : OUT_H
out : OUT_Claw

Link : IN_S IN_S@SC
Link : IN_U IN_U@USSC
Link : IN_CS IN_CS@CS
Link : OUT_M@AC OUT_M
Link : OUT_Claw@CL OUT_Claw
Link : OUT_H@AC OUT_H
Link : OUT_S@SC IN_S@AC
Link : OUT U@USSC IN U@AC
Link : OUT_TC@AC IN_C@CS
Link : OUT_U@AC IN_U1@USSC
Link : OUT_Claw@AC IN_AC@CL
Link : OUT_C@CS IN_AC@AC

[SC]
soundprocessingtime : 00:00:00:01

[USSC]
UProcessingtime : 00:00:00:001

[CL]
clawpreparationTime : 00:00:00:05
grabReleaseTime : 00:00:00:10

[CS]
colorSensedTime : 00:00:00:01

[AC]
armpreparationTime : 00:00:00:01
colorTime : 00:00:00:10
armTime : 00:00:02:00
clawTime : 00:00:01:10
clawTime2 : 00:00:01:10
hTime : 00:00:00:01
armTime : 00:00:00:01
goTime : 00:00:00:01
```

Figur4.2.1.*1*. ArmController .ma file

```
00:00:05:00  IN_S  2
00:00:06:00 IN_S 26
00:00:06:00 IN_U 10
00:00:07:00 IN_U 2
00:00:08:00 IN_U 1
00:00:09:00 IN_CS 9
```

Figure4.2.1.*2*. ArmController event file

```
00:00:06:002 out_h 1
00:00:07:002 out_m 0
00:00:07:002 out_h 0
00:00:10:021 out_claw 1
00:00:11:021 out_m 1
00:00:13:021 out_m 0
```

Figure 4.2.1.*3*. ArmController output file

It is very easy to see the flow of events using COUT statements in CD++. In this scenario a red ball was found therefore the ball was grabbed by the arm and brought up for 2 seconds. The outputs are seen in figure 22, where a value of 1 is sent to claw to grab the ball, a value of 1 is sent to arm to move up, and after 2 seconds a value of 0 is sent to arm to stop the arm from moving up.

In a second scenario where a blue ball is found, the event file would be different in that the input to ColorSensor should be 2 instead of 9. The output is shown in Figure 4.2.1.*5*.

```
00:00:06:002 out_h 1
00:00:07:002 out_m 0
00:00:07:002 out_h 0
00:00:09:002 out_m 1
00:00:11:002 out_m 0
```

Figure4.2.1.*5* ArmController output, Blue ball found

5.2.2 E-CD++

The Arm controller was tested in ECD++ to see the results. Figure 4.2.2.*1* shows the event file and the case were the ball is red and 4.2.2.*2* shows the output result of the event file. Figure 4.2.2.*3* shows the event file and the case were the ball is blue and 4.2.2.*4* shows the output result of the event file.

```
00:00:05:00 00:00:06:00 IN_S OUT_M 30
00:00:07:00 00:00:08:00 IN_U OUT_M 4
00:00:09:00 00:00:10:00 IN_U OUT_M 2
00:00:10:00 00:00:11:00 IN_CS OUT_M 9
```

Figure 4.2.2.*1* Realtime event file for ArmController: Red Ball

```
Cur Time: 00:00:05:002 (no dealine specified)
OutPort: out_h PortValue: 1
Cur Time: 00:00:07:002 Deadline: 00:00:06:000 (NOT
succeeded) OutPort: out_m PortValue: 0
Cur Time: 00:00:07:002 (no dealine specified)
OutPort: out_h PortValue: 0
Cur Time: 00:00:11:025 (no dealine specified)
OutPort: out_claw PortValue: 1
Cur Time: 00:00:12:025 Deadline: 00:00:08:000 (NOT
succeeded) OutPort: out_m PortValue: 1
Cur Time: 00:00:17:025 Deadline: 00:00:10:000 (NOT
succeeded) OutPort: out_m PortValue: 0
```

Figure 4.2.2.*2* Realtime Output for ArmController: Red ball

```
00:00:05:00 00:00:06:00 IN_S OUT_M 30
00:00:07:00 00:00:08:00 IN_U OUT_M 4
00:00:09:00 00:00:10:00 IN_U OUT_M 2
00:00:10:00 00:00:11:00 IN_CS OUT_M 2
```

Figure 4.2.2.*3* Realtime event file for ArmController: Blue Ball

```
Cur Time: 00:00:05:002 (no dealine specified)
OutPort: out_h PortValue: 1
Cur Time: 00:00:07:002 Deadline: 00:00:06:000
(NOT succeeded) OutPort: out_m PortValue: 0
Cur Time: 00:00:07:002 (no dealine specified)
OutPort: out_h PortValue: 0
Cur Time: 00:00:10:006 Deadline: 00:00:08:000
(NOT succeeded) OutPort: out_m PortValue: 1
Cur Time: 00:00:15:006 Deadline: 00:00:10:000
(NOT succeeded) OutPort: out_m PortValue: 0
```

Figure 4.2.2.*4* Realtime Output for ArmController: Blue Ball

5.3 Execution of Code in Realtime –Robot connected

The code was executed on the robot and as can be seen in figure 4.3.1 when the ball was red, the robot grabbed the ball. This figure also shows a picture of blue ball. It was not grabbed since it is blue [7][8].

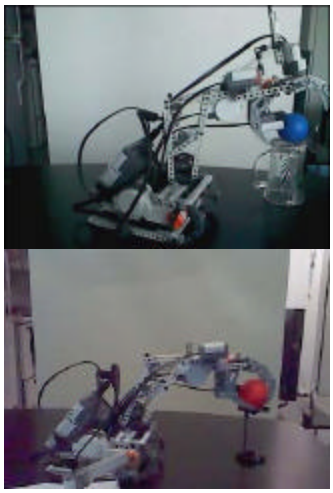


Figure 4.3.1 Robot arm grabbing red ball and not grabbing blue ball

6. CONCLUSION

DEVS modeling is a useful modeling technique that can be used to model different systems. In this paper we have presented a way to model a RTS using DEVS and simulating and running the model using E-CD++ which is an extension to CD++ for RT environments.

The first step in developing the system was to define each model and their functionality and formalization. These steps included the formalization of the model, structural diagram and state diagrams for each of the models. This step is the most important step in developing a model since the actual implementation is straight forward if the state diagrams are drawn correctly.

The next step was to write the code for the model in CD++ and ECD++. This step was done incrementally and each step was tested during the development of the code.

After the implementation of each unit, they were tested individually to ensure correct behaviour of the unit. The models were then connected and tested as a whole, and the results were shown.

We have shown that with DEVS formalization, it is very straight forward to develop models, in this case in RTSs.

6. REFERENCES

[1] B. Zeigler, T. Kim, H. Praehofer. “Theory of Modeling and Simulation”. Academic Press 2000, ISBN-10: 0127784551.

[2] G. Wainer, “Discrete-Event Modeling and Simulation: a Practitioner’s approach”. CRC Press. Taylor and Francis. 2009.

[3] Y. H. Yu, and G. Wainer, "eCD++: an engine for executing DEVS models in embedded platforms" Proceedings of the 2007 SCS Summer Computer Simulation Conference, San Diego, pp. 323-330. CA - 2007

[4] Chow A, Kim D, Zeigler B. “Parallel DEVS: A parallel, hierarchical, modular modeling formalism” In Proceedings of Winter Simulation Conference, 1994, Orlando, Florida.

[5] "Designing an Interface for RT and Embedded DEVS", Mohammad Moallemi, Gabriel A. Wainer, Proceedings of 2010 Spring Simulation Conference (SpringSim10), DEVS Symposium - April 2010

[6] Ledeczi, A.; Davis, J.; Neema, S.; Agrawal, A. “Modeling methodology for integrated simulation of embedded systems”. ACM TOMACS 13(1), 82-103. 2003.

[7] Hu, X.; Zeigler, B.P. “Model Continuity in the Design of Dynamic Distributed Real-Time Systems”, IEEE Transactions On Systems, Man And Cybernetics— Part A: Systems And Humans, 35: 6, pp. 867- 878, November, 2005.

[8] Cho, Y. K.; Hu, X.; Zeigler, B.P. “The RTDEVS/CORBA Environment for Simulation-Based Design of Distributed Real-Time Systems”, SIMULATION: Transactions of The Society for Modeling and Simulation International, Volume 79, Number 4, 2003.

[9] Godding, G., H.S. Sarjoughian, K. Kempf, (2007) . “Application of Combined Discrete-event Simulation and Optimization Models in Semiconductor Enterprise Manufacturing Systems", Winter Simulation Conference.

[10] Discrete Event Modelling and Simulation A practitioner’s Approach, Gabriel W.Wainer, 2009, Taylor & Francis Group,

[11] Video of Blue ball http://www.youtube.com/watch?v=j5QhX4QFER8

[12] http://www.youtube.com/watch?v=R1MT8OLu8Co Video of Red Ball