Implementing Finite State Machines Using the CD++ toolkit

Tao Zheng **Gabriel A. Wainer**

Department of Systems and Computer Engineering.Carleton University
4456 Mackenzie Building. 1125 Colonel By Drive. Ottawa, ON. K1S 5B6. Canada
zhengtao@sce.carleton.ca **gwainer@sce.carleton.ca**

Keywords: DEVS, CD++, finite state machines, Moore machines

Abstract

DEVS (Discrete EVents systems Specification) is a formal modeling and simulation framework based on generic dynamic systems concepts. DEVS well-defined concepts of coupling of components, hierarchical, modular model construction permitted to provide a common representation for different existing formalisms (including Petri Nets, PDE, and different state machines). Here, we show how to apply the DEVS to build a library of finite state machine (Moore machines) using a toolkit called CD++. State machines can be easily created and simulated by composing the instances of the state and connecting those states according to some rules.

1. INTRODUCTION

The DEVS formalism [1, 2] was originally defined as a discrete-event modeling specification mechanism. It is a systems theoretical approach that allows the definitions of hierarchical modular models that can be easily reused. A real system modeled with DEVS is described as a composite of submodels, each of them being behavioral (atomic) or structural (coupled).

A DEVS atomic model is formally described by:

$$M = < X, S, Y, \delta_{int}, \delta_{ext}, \lambda, D >$$

where X is the input events set; S is the state set; Y is the output events set; δ_{int} is the internal transition function; δ_{ext} is the external transition function; is the output function; and D is the duration function.

A DEVS coupled model is composed of several atomic or coupled submodels. They are formally defined as:

$$CM = < X, Y, D, \{Mi\}, \{Ii\}, \{Zij\} >$$

where X is the set of input events; Y is the set of output events; D is an index for the components of the coupled model, and $\forall i \in D$, Mi is a basic DEVS (that is, an atomic or coupled model), Ii is the set of influencees of model i (that is, the models that can be influenced by outputs of model i), and $\forall j \in Ii$, Zij is the i to j translation function.

We can see that coupled models are defined as a set of basic components (atomic or coupled) interconnected through the model's interfaces. The translation function is in charge of converting the outputs of a model into inputs for the others. To do so, an index of influencees is created for each model (Ii). This index defines that the outputs of the model Mi are connected to inputs in the model Mj, where j is an element of Ii.

DEVS hierarchical constructions enable multi-formalism modeling (that is, the coupling of and transformation between models described in different formalisms). Using different formalisms to represent such systems enables a modeler to choose the formalism that lends itself best to represent each sub-systems.

CD++ [3, 4] is a tool that allows a user to implement DEVS models. The tool is built as a hierarchy of models, each of them related with a simulation entity. Atomic models can be programmed and incorporated onto a basic class hierarchy programmed in C++. A specification language allows defining the model's coupling, including the initial values and external events. The tool also enables a user to build models using graph-based notations, which allows visualization of the problem in a more abstract way. Therefore, we have used an extended graphical notation to allow the user define atomic models behavior. Each graph defines the state changes according to internal and external transition functions, and each is translated into an analytical definition. Our long term goal is to provide the user with a set of libraries to develop complex models based on multiformalisms. We have already included Finite State Automata, Petri Nets, DEVS graphs and DEVS atomic models written in C++, enabling the users to use different formalisms to describe different properties of a system. In this work we will focus on the implementation of finite state machines as an example of multi-formalism model definition.

Finite State Machines (FSM) are popular for modelling systems in a variety of areas such as software design and digital logic design etc. It is formally defined as follows:

$$FSM = < S, X, Y, f, g >$$

where
- X: finite input set
- Y: finite output set
- S: finite state set
- f: next state function
  - f: X * S -> S
- g: output function
  - g: S -> Y : Moore machine
  - g: X * S -> Y: Mealy machine

Every Finite State Machine (FSM) is supposed to have an initial state, a next state function which defines how to obtain the next state in the system, and an output function that uses current state and inputs to generate outputs. According to the relationship between the input sets and output functions, two kinds of FSMs can be defined: Moore machines, in which outputs are independent from the inputs, and Mealy machines, in which, besides the current state variables, the current inputs are analyzed to decide the current output value.

We describe an implementation where the CD++ toolkit was used for the purpose of simulating FSM Moore machines (thus, FSM means a Moore machine from here on) using DEVS specifications. We focused on developing a library for FSM on CD++, obtaining an unmodified DEVS simulator. The construction of a library of atomic models to represent FSM was straightforward since it was shown in [2] that FSMs can be embedded in DEVS because any discrete event behaviour can be expressed as a DEVS model. The remaining sections are devoted to describe how this can be achieved.

2. FSMs IN CD++

Every Finite State Machine consists of a number of states with transition functions. If we can define the behavior of a generic state as an atomic model, a Finite State Machine could be created easily and formally by generating a coupled DEVS model which consists of a number of those atomic models. This section defines an atomic model called *MooreState* and FSM models based on this atomic model.

2.1 Atomic model *MooreState*

Figure 1 shows the sketch of the atomic model *MooreState*. This atomic model represents the behavior of a generic state in Moore machines:

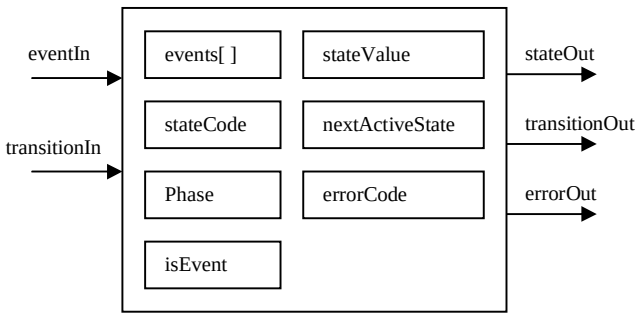

Figure 1. Sketch of the Atomic Model Definition of *MooreState*

A unique global *stateCode* is assigned to each state in a Finite State Machine. The *phase* indicates whether this state is active or passive. Only one state is active at a time in one Finite State Machine. The events of a FSM are encoded as integers as well. The *eventIn* receives encoded numbers representing external events. If a state receives a legal event listed in *events[ ]* when it is active, the *stateOut* port sends out the current *stateValue*, and the *transitionOut* port sends out the *nextActiveState* signal to all the *transitionIn* ports in the FSM announcing which state is active in the next step. A state becomes active if the encoded number received from *transitionIn* port is same as its *stateCode*. An *errorCode* would be sent out from the *errorOut* port if an error occurs in the state.

The state is defined an atomic model in CD++. Using this atomic model definition, a coupled model can be built which defines a state in an FSM. The following figure describes such component:

```
[top]
components : moorestate@MooreState
out : stateOut
out : transitionOut
out : errorOut
in : transitionIn
in : eventIn
Link : eventIn eventIn@moorestate
Link : transitionIn  transitionIn@moorestate
Link : stateOut@moorestate stateOut
Link : transitionOut@moorestate  transitionOut
Link : errorOut@moorestate errorOut
Link : transitionOut@moorestate
       transitionIn@moorestate

[moorestate]
StateCode : 0
// current state has stateCode 0 (initial state)
StateValue : 1
// current State has the stateValue 1
NumberOfTransitions : 2
//two legal transitions defined in current state
Transition01 : 0->1
//if event 0 is received, the state with stateCode
1 is active next
Transition02 : 1->0
//if event 1 is received, the state with stateCode
0 is active next
```

Figure 2. Sketch of the Coupled Model Definition of *MooreState*

All the items listed in the *[moorestate]* part are specific and required for the states in a FSM. The number of *TransitionXX* items depends on the item *NumberOfTransition*s. If the number of defined transitions is more than *NumberOfTransition*s, some of the defined transitions listed last will be ignored.

We executed this coupled model with different test cases. Here we show an execution scenario in which we consider we trigger the events shown in Figure 3. According to the model definition, the current state is the initial state, and it is set as active initially. It responses to the first 3 events and generates outputs. The next two events with bold fonts don't cause any outputs in this state because the current state is not active anymore. At time 00:00:25:000, the current state is active again by receiving 0 from

transitionIn. The event with bold and italic fonts is an illegal event for the current state. An error messaged is generated. However, the current state retains its *phase* after this illegal event. The output is the same as what we expected.

00:00:02:00 eventIn 1 00:00:05:00 eventIn 1 00:00:10:00 eventIn 0 **00:00:15:00 eventIn 1** **00:00:20:00 eventIn 2** 00:00:25:00 transitionIn 0 00:00:30:00 eventIn 1 ***00:00:33:00 eventIn 2*** 00:00:35:00 eventIn 1 00:00:40:00 eventIn 0	00:00:02:000 transitionout 0 00:00:02:000 stateout 1 00:00:05:000 transitionout 0 00:00:05:000 stateout 1 00:00:10:000 transitionout 1 00:00:25:000 stateout 1 00:00:30:000 transitionout 0 00:00:30:000 stateout 1 ***00:00:33:000 errorout -999*** 00:00:35:000 transitionout 0 00:00:35:000 stateout 1 00:00:40:000 transitionout 1

Figure 3. Executing the *MooreState* Atomic Model

Finite State Machine models could be easily created as coupled DEVS models by connecting a number of instances of *MooreState*.

There are several basic rules to create a Finite State Machine by connecting the states

- All the *transitionOut* and *transitionIn* ports should be connected together inside the FSM. I.e. in the view of a FSM, these ports don't communicate with the external environment and they are not visible to the outside world.
- All the *eventIn* ports of states should be connected to an input port of a FSM.
- All the *errorOut* ports of states are connected together as an output port of a FSM
- Each *stateOut* port of state could either be connected to an individual output port of a FSM or connected together as one output port of a FSM
- All the states in a FSM are encoded as integer numbers (*stateCode*) during simulation. The state with the *stateCode* 0 is the initial state in the FSM
- All the legal events for this FSM are also encoded as integer numbers during simulation.

2.2 A Simple FSM

A simple moore machine shown in Figure 4 which was introduced in [5]. This finite state machine sends out value 1 whenever its input string has at least two 1's in sequence. Otherwise, value 0 is sent out. Three states are defined in this FSM. The state machine should send out 0 at state *0* or*1*, and 1 at state *2*.

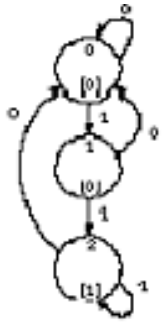

Figure 4. A Simple FSM

The following figure shows a sketch of the definition of this FSM defined as a DEVS coupled model. Each of the states is represented using the *MooreState* atomic model. The state *S0*, *S1* and *S2* represent the 3 states. The *in* port receives the external events from the environment. The ports *s1*, *s2* and *s3* generate the output from respective states. The port *error* is used to check whether there are illegal events received.

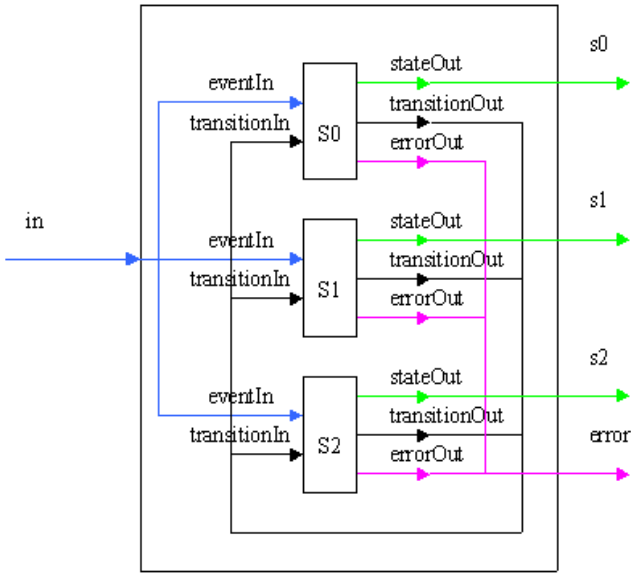

Figure 5. Coupled Model Definition of the FSM in Figure 4

The event codes and state codes are defined as:

Event Code Table:

Event	Event Code
Input 0	0
Input 1	1

State Code Table:

State	State Code
S0	0
S1	1
S2	2

The FSM of this simple moore machine is defined as follows in CD++:

```
[top]
components : s0@moorestate s1@moorestate
components :s2@moorestate
out : s0 s1 s2 error
in : in
Link : in eventIn@s0
Link : in eventIn@s1
Link : in eventIn@s2
Link : transitionOut@s0 transitionIn@s0
Link : transitionOut@s0 transitionIn@s1
Link : transitionOut@s1 transitionIn@s0
Link : transitionOut@s1 transitionIn@s2
Link : transitionOut@s2 transitionIn@s2
Link : transitionOut@s2 transitionIn@s0
Link : stateOut@s0 s0
Link : errorOut@s0 error
Link : stateOut@s1 s1
Link : errorOut@s1 error
Link : stateOut@s2 s2
Link : errorOut@s2 error

[s0]
StateCode : 0
StateValue : 0
NumberOfTransitions : 2
Transition01 : 0->0
Transition02 : 1->1

[s1]
StateCode : 1
StateValue : 0
NumberOfTransitions : 2
Transition01 : 0->0
Transition02 : 1->2

[s2]
StateCode : 2
StateValue : 1
NumberOfTransitions : 2
Transition01 : 0->0
Transition02 : 1->2
```

Figure 6. Coupled Model Definition of the FSM in Figure 4 in CD++

We executed this coupled model with different test cases. Here we show an execution scenario in which we consider we trigger the following events:

00:00:00:10 in 0	00:00:00:010 s0 0
00:00:00:20 in 1	00:00:00:020 s1 0
00:00:00:30 in 1	**00:00:00:030 s2 1**
00:00:00:40 in 1	**00:00:00:040 s2 1**
00:00:00:50 in 0	00:00:00:050 s0 0
00:00:00:60 in 0	00:00:00:060 s0 0
00:00:00:70 in 1	00:00:00:070 s1 0
00:00:00:80 in 0	00:00:00:080 s0 0
00:00:00:90 in 0	00:00:00:090 s0 0
00:00:00:100 in 0	00:00:00:100 s0 0
00:00:00:110 in 1	00:00:00:110 s1 0
00:00:00:120 in 0	00:00:00:120 s0 0
00:00:00:130 in 1	00:00:00:130 s1 0
00:00:00:140 in 1	**00:00:00:140 s2 1**
00:00:00:150 in 0	00:00:00:150 s0 0

Figure 7. Executing the FSM in Figure 4 in CD++

The input events with bold fonts should cause the FSM to generate 1 as the output, and the output with bold fonts do have the expected results.

2.3 Vender Machine Model

A vender machine model shown in Figure 8 was introduced in [6]. This vender machine accepts 5, 10 cents coins and sells candy bars worth 15 or 20 cents. The simplified coupled model definition (the connections among the *transitionIn*s and *transitionsOut*s are omitted) is shown in Figure 9.

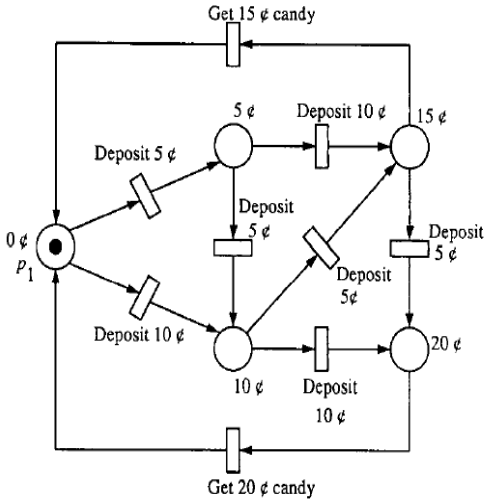

Figure 8. Vender Machine Model

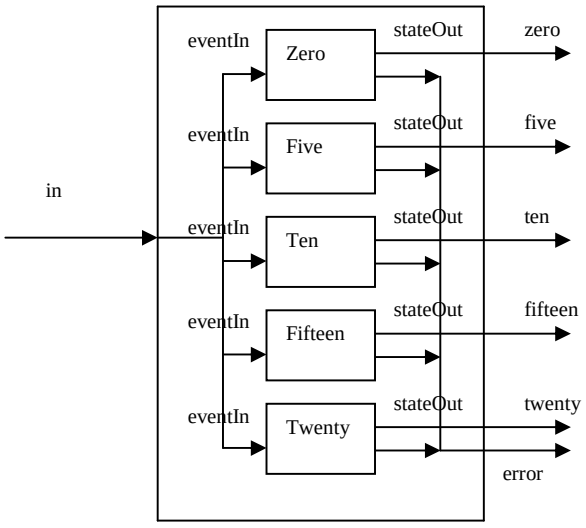

Figure 9. Simplified Coupled Model Definition of the Vender Machine Model in Figure 8

In Figure 9, the states *Zero, Five, Ten, Fifteen, Twenty* indicates the mount of coins that has been inserted before

selling a candy. The *in* port receives the external events from the environment. The ports *zero*, *five*, *ten*, *fifteen* and *twenty* generate the output from respective states. The port *error* is used to check whether there are illegal events received.

The vender machine has the following event code table and state code table:

Event Code Table:

Event	Event Code
Deposit 5¢	0
Deposit 10¢	1
Get 15¢ Candy	2
Get 20¢ Candy	3

State Code Table:

State	State Code
Zero	0
Five	1
Ten	2
Fifteen	3
Twenty	4

The FSM of this vender machine model is defined as follows in CD++:

```
[top]
components : zero@moorestate five@moorestate
components : ten@moorestate fifteen@moorestate
components : twenty@moorestate
out : zero five ten fifteen twenty error
in : in

Link : in eventIn@zero
Link : in eventIn@five
Link : in eventIn@ten
Link : in eventIn@fifteen
Link : in eventIn@twenty

Link : transitionOut@zero transitionIn@five
Link : transitionOut@zero transitionIn@ten
Link : transitionOut@five transitionIn@ten
Link : transitionOut@five transitionIn@fifteen
Link : transitionOut@ten transitionIn@fifteen
Link : transitionOut@ten transitionIn@twenty
Link : transitionOut@fifteen transitionIn@twenty
Link : transitionOut@fifteen transitionIn@zero
Link : transitionOut@twenty transitionIn@zero

Link : stateOut@zero zero
Link : errorOut@zero error
Link : stateOut@five five
Link : errorOut@five error
Link : stateOut@ten ten
Link : errorOut@ten error
Link : stateOut@fifteen fifteen
Link : errorOut@fifteen error
Link : stateOut@twenty twenty
Link : errorOut@twenty error

[zero]
StateCode : 0
StateValue : 0
NumberOfTransitions : 2
Transition01 : 0->1
Transition02 : 1->2

[five]
StateCode : 1
StateValue : 5
NumberOfTransitions : 2
Transition01 : 0->2
Transition02 : 1->3

[ten]
StateCode : 2
StateValue : 10
NumberOfTransitions : 2
Transition01 : 0->3
Transition02 : 1->4

[fifteen]
StateCode : 3
StateValue : 15
NumberOfTransitions : 3
Transition01 : 0->4
Transition02 : 1->4
Transition03 : 2->0

[twenty]
StateCode : 4
StateValue : 20
NumberOfTransitions : 1
Transition01 : 3->0
```

Figure 10. Coupled Model Definition of the Vender Machine Model in Figure 8 in CD++

The scenarios of buying one 15¢ candy and two 20¢ candies are simulated and the expected results are shown in Figure 11.

00:00:00:10 in 0	00:00:00:010 five 5
00:00:00:20 in 0	00:00:00:020 ten 10
00:00:00:30 in 0	00:00:00:030 fifteen 15
00:00:00:40 in 2	00:00:00:040 zero 0
00:00:00:50 in 0	00:00:00:050 five 5
00:00:00:60 in 0	00:00:00:060 ten 10
00:00:00:70 in 1	00:00:00:070 twenty 20
00:00:00:80 in 3	00:00:00:080 zero 0
00:00:00:90 in 1	00:00:00:090 ten 10
00:00:00:110 in 1	00:00:00:110 twenty 20
00:00:00:130 in 3	00:00:00:130 zero 0

Figure 11. Executing the Vender Machine Model in Figure 8 in CD++

Two more FSM models, an ATM model introduced in [7] and a "Plain Ordinary Telephony Service" (POTS) model described in [8], were created, simulated and tested. Due to the page limitation, these two models can't be shown in this paper. The atomic models, FSMs implemented and the CD++ tools can be found in:
`http://www.sce.carleton.ca/faculty/wainer/wbgraf/`

3.CONCLUSION

We showed how we implemented an atomic DEVS model called *MooreState* representing the generic state behavior of the Finite State Machines (Moore machines) using CD++ toolkit. By connecting a number of these generic states according to some rules, Finite State

Machines could be created easily and formally. CD++ provides an effective way to create and generate Finite State Machines.

ACKNOWLEDGMENTS

This work was partially funded by the Natural Sciences and Engineering Research Councel of Canada (NSERC) and the Institute of Robotics and Intelligent Systems (IRIS, Canada).

REFERENCES

[1] B. Zeigler, “Theory of modeling and simulation”. Wiley, 1976

[2] B. Zeigler, T. Kim, H. Praehofer, “Theory of Modeling and Simulation: Integrating Discrete Event and Continuous Complex Dynamic Systems”. Academic Press, 2000

[3] G. Wainer, “CD++: A toolkit to Develop DEVS Models”, Software - Practice and Experience, Vol. 32, No. 13, pp1261-1306, 2002 .

[4] Rodriguez, D.; Wainer, G. "New Extensions to the CD++ tool". In *Proceedings of SCS Summer Computer Simulation Conference*, Chicago, IL. 1999.

[5] A. Rosetti, “Finite State Machine Design”, http://www2.ele.ufes.br/~ailson/digital2/cld/chapter8/chapter08.doc4.html (Accessed on Nov. 30, 2002)

[6] G. Quan, “Computational Models” http://www.cse.sc.edu/~gquan/Course790/Lecture6-7.ppt (Accessed on Nov. 30, 2002)

[7] “Finite State Machines”, http://www.csc.ncsu.edu/eos/users/e/efg/210/s99/course_locker/www/Notes/fsm/ (Accessed on Nov. 30)

[8] Erlang, ”Finite State Machines” http://www.erlang.org/doc/r8b/doc/design_principles/fsm.html (Accessed on Nov. 30, 2002)