Definition of Dynamic DEVS Models- Dynamic Structure CD++

Monageng Kgwadi mkgwadi@connect.carleton.ca
Hui Shang shanghuibox@gmail.com
Gabriel Wainer gwainer@sce.carleton.ca

Department of Systems and Computer Engineering
Carleton University
4456 Mackenzie Building
1125 Colonel By Drive
Ottawa, ON, K1S5B6, Canada

Keywords: Dynamic Structure DEVS, CD++.

Abstract **Dynamic Structure DEVS allows one to model systems that under go structural and behavioural changes. Dynamic Structure CD++ is a modelling and simulation tool that allows models to be changed at run time to facilitate accurate modelling of dynamic systems. We model and simulate two different examples of dynamic systems focusing on their dynamic behaviour using DS-CD++.**

1 INTRODUCTION

Discrete event system specification (DEVS) is a technique derived from system theory that describe systems whose state variables change at discrete set of points in time [4] [2]. DEVS framework provides well-defined definitions for modular model description, coupling of components and heirarchy[3].Dynamic DEVS is a modelling technique that addresses behavioural and structural changes that models incur as a result of changes in the environment. A dynamic model is a model that change its behaviour and/or structure based on a set of input variables or its environment. Dynamic Structure CD++ (DS-CD++) is a modelling and simulation tool that allows changes to be made on models at run-time. DS-CD++ is an extension to Embedded CD++(e-CD++) and Real-Time CD++(RT-CD++) which were designed to model systems for embedded applications in real-time. DS-CD++ uses the Dynamic DEVS formalism to define models.
We introduce two different dynamic models in DS-CD++ specifications showing the main features of this environment. The models presented herein are an active fuel management system of a vehicle and a threaded server. Both models exhibit dynamic behaviour and change their structure with time. The active fuel management system of a vehicle is a system that minimises fuel consumption by turning off some engine cylinders when the vehicle does not require a lot of power. A threaded server running on a computer may have multiple threads at any time based on the number of clients which it is in communication with at any instant in time. This paper present a detailed and formal descriptions of the models in section 2 and the tests performed on the models. A conclusion of the project and references are presented in sections 3 and 4.

2 MODEL DESCRIPTIONS

CD++ toolkit allows object oriented model specification based on DEVS formalism [4]. CD++ defines two types of models, atomic and coupled models. Coupled models are made up of interconnected atomic and/or coupled model components. DS-CD++ defines models as basic and network models. Basic models are atomic structures while network models are coupled components made of basic and/or network models. DS-DEVS formalism describes a special model called a network executive which performs the structural changes of the dynamic network models [1],[3] . Hence DS-CD++ defines a network executive model which is a specialized atomic model that performs the network structural changes at run-time. This section presents sample models that were developed in DS-CD++. The models were developed and defined using the Dynamic DEVS formalism as described in [1] and [3].

2.1 Active Fuel Management Model

An Active Fuel Management system (AFM) is a system that manages fuel consumption of a vehicle during driving. In a high-power vehicle with an eight cylinder (V8) engine, the AFM system allows the full use of the engine cylinders when demanded by the driver and conditions [5]. For example when the driver is accelerating, maximum capacity of the engine is used. When the driver does not demand a lot of power, the AFM system turns off some cylinders to minimise power consumption.

2.1.1 Conceptual Model Description

The Active Fuel Management system is modelled herein as a system that manages a V8 engine that changes to a V6 engine to minimise fuel consumption. The system has a control module that takes inputs from the driver and the environment and then changes the engine structure based on the inputs. The control model takes inputs from the driver and environment. The driver inputs are defined as:

0: driver stopping the engine
1: driver starting the engine
2: driver accelerating
3: driver idle (taking his foot off the gas pedal)

The states of the engine depend on the inputs from the driver and are defined as:

OFF: the engine is off
IDLE: the engine is on but not accelerating
ACCELERATE:the engine is accelerating
DECELERATE: the engine is decelerating

The inputs from the environment are modelled as discrete events that tells the system that maximum power is required or power-save mode can be used. These are environment conditions that may result from the road conditions or from terrain sensors on the car. They are presented here as:

1: power-save
2: maximum power required

The states of the control unit are

POWERSAVE: minimal fuel consumption
MAXPOWER: maximum power.

2.1.2 Dynamic DEVS Formalism Description

The system is defined using the description in [1] and [3] as follows: $DSDEN_{AFM} = (X_{AFM}, Y_{AFM}, X, M_x)$ where:

AFM: network name
$X_{AFM} = \{(driver, N), (enviro, N)\}$
input set of AFM network
$Y_{AFM} = \{(revsperminute, N)\}$
output set of AFM network
$X = activeexec$: network executive
$M_x = (X_x, S_x, S_{o,x}, Y_x, \gamma, \Sigma^*,$
$\delta_x, \lambda_x, \tau_x)$:defines the network structure
X_x : inputs to the network
Y_x : output of the network
S_x : current network structure
$S_{o,x}$: the initial netwrok structure
$\gamma : S_x ... \Sigma_\alpha$: the structure function that converts current network structure to the next structure based on inputs where:
$\Sigma_\alpha \epsilon \Sigma^*$ and
$\Sigma^* = \{(EngControl, EngineV6),$
$(EngControl, EngineV8)\}$ the set of all possible network structures.

Figure 1 shows the coupling descriptions of the AFM model. The network executive(*activeexec*) is defined as a component of the *engine* coupled model, and is responsible for changing the structure of the network when the *EngContrl* module executes the code shown in figure 2. The method *strucChange(int i)* invokes the network executive to change the network to the structure with the identity i as specified by the *Scomm* variable in the structure couplinng definition file, refer to figure1.

```
[top]
components : engcntrl@EngControl engine
in : in  in_env
out : out
Link : in driver_out@engcntrl
Link : eng_out@engcntrl in1@engine

[engine]
components : enginev6@enginev6  activeexec#activeexec
in : in in1
Scomm : struc1
Link : in1 inv@enginev6
Link : outv@enginev6 out1

[engineupdate1]
components : enginev8@enginev8
in : in in1
Scomm : struc2
Link :  in1 inw@enginev8
```

Figure 1: Definition of the structure couplings

2.1.3 Testing of Models and Integration

The models were designed using a top-down approach. To start, the EngControl model was created which

```
if (phase == POWERSAVE){
        if(msg.port() == conditn_out && msg.value() == 2) {
        phase = MAXPOWER;
        int backup = 2;
        strucChange(backup);
        }
}
else {

        if(msg.port() == conditn_out && msg.value() == 1) {
        phase = POWERSAVE;
        int backup = 1;
        strucChange(backup);
        }
}
```

Figure 2: Code snippet showing how the structure is changed in run-time

takes control inputs from the driver and environment then chooses the suitable engine for the combination of state and inputs. The network executive was then created to allow structural changes. Initially the EngControl model changed between two void engine models that did not do anything. After the Engcontrol model was verified to work properly, the EngineV6 model and the EngineV8 models were created using DEVS formalism. The two models are similar the difference being that the EngineV8 model produces more power than that of the EngineV6 hence reach higher engine revolutions per minute faster than the EngineV6. Figure 3 is an extraction of the logfile which shows structural changes at execution time. The first line in bold shows the current engine model is EngineV6, the following bold line reflects an input from the driver corresponding to acceleration and hence the structural change to an EngineV8 model shown by the third line in bold.

```
MSG: D / 00:00:03:550 / enginev6(04) / ... /     0.00000 TO engine(03)
MSG: D / 00:00:03:550 / engine(03) / ... /     0.00000 TO top(01)
MSG: D / 00:00:03:550 / top(01) / ... /     0.00000 TO Root(00)
MSG: X / 00:00:05:000 / Root(00) / in /     2.00000 TO top(01)
                .
                .
                .
MSG: St / 00:00:05:000 / engine(03) TO enginev8(07)
MSG: D / 00:00:05:000 / enginev8(07) / ... /     0.00000 TO engine(03)
                .
                .
                .
MSG: X / 00:00:05:000 / engine(03) / inw /     2.00000 TO enginev8(07)
```

Figure 3: part of logfile showing structural change

2.2 Threaded Server Model

The threaded server model is designed herein to model a multitask server that runs on a computer. A multitask server offers parallel service to multiple clients by creating threads to serve each client. The threads are created on demand by the server. A multitask server runs a main thread that just listens to clients requests on a port on the computer. Each time a client requests service, a new thread to serve the client is created. The thread is then closed at the request of the client after the service is finished.

2.2.1 Conceptual Model Description

The threaded server is described by a main server thread that listens for client requests on a port. Every time a client requests a connection to the server, a new thread is created provided the number of threads does not exceed a predefined maximum number of threads. The client can then stay connected for a time and then close the connection. The inputs that the main server thread takes are:

1: Service Request
2: Close Request

The main server thread keeps the number of threads created and updates the number each time a new thread is created or closed. A predefined maximum number of threads is used to keep control of the number of existing threads and consequently the number of open connections with clients at a given point in time. The states of the main server thread are:

LISTEN : server is listening to client requests
ACCEPT : server creates a new thread
DELETE : server deletes an existing thread

The threads are represented by models that take input from a client and the main server thread.

2.2.2 Dynamic DEVS Formalism Description

The Multitask server system is defined using the description in [1] as follows:
$DSDEN_{ThrdServer} = (X_{ThrdServer}, Y_{ThrdServer}, X, M_x)$
where:

ThrdServer: network name
$X_{ThrdServer} = \{(driver, N), (enviro, N)\}$
input set of ThrdServer network
$Y_{ThrdServer} = \{(revsperminute, N)\}$
output set of ThrdServer network
$X = activeexec$: Network executive

$M_x = (X_x, S_x, S_{o,x}, Y_x, \gamma, \Sigma^*, \delta_x, \lambda_x, \tau_x)$
The structure function represented as: $\gamma : S_x ... \Sigma^*$
The structure definition function that maps a network structure$\Sigma_\alpha \epsilon \Sigma^*$ where
$\Sigma^* = \{(ServerThread), (ServerThread, Thread1),$
$(ServerThread, Thread1, Thread2),$
$(ServerThread, Thread1, Thread2, Thread3)\}$

Figure 4 shows the structure of the network. For this example only a maximum of 3 threads is presented but that could be extended by extending the Σ^* set and performing the necessary editing in the model description files .

```
[top]
components : servrThrd@ServerThread    servercood#servercood
             :
Scomm : struc1


[threadupdate1]
components :  servrThrd@ServerThread thread@Thread
             :
Scomm : struc2

[threadupdate2]
components :servrThrd@ServerThread    thread@Thread  thred2@Thread
             :
Scomm : struc3
Link : out@servrThrd out

[threadupdate3]
components : servrThrd@ServerThread    thread@Thread  thred2@Thread
thred3@Thread
             :
Scomm : struc4
```

Figure 4: The Structure definition of the Server

2.2.3 Testing of models and Integration

The threaded server model was created using an incremental method. Initially the serverThread model was designed and tested for operational correctness by having an empty executive model that shown when a change was to be performed. After the operation of the serverThread model was verified, the network executive was then filled in to perform the structural changes that were to be performed by adding and removing threads. The Thread model was then created and the overall system was tested and verified to work as expected under different inputs scenarios. Figure 5 shows the output of the threaded server which reflects the number of active threads at a time

```
Cur Time: 00:00:01:500 Deadline: 00:00:01:540 (Succeeded) OutPort: out PortValue: 0
00:00:01:500 out 1 (no dealine specified)
Cur Time: 00:00:02:500 Deadline: 00:00:02:540 (Succeeded) OutPort: out PortValue: 1
00:00:02:500 out 2 (no dealine specified)
Cur Time: 00:00:03:500 Deadline: 00:00:03:540 (Succeeded) OutPort: out PortValue: 2
00:00:03:500 out 3 (no dealine specified)
Cur Time: 00:00:04:000 Deadline: 00:00:04:040 (Succeeded) OutPort: out PortValue: 3
00:00:04:000 out 3 (no dealine specified)
```

Figure 5: Output of the number of threads

3 CONCLUSIONS

The DS-CD++ toolkit provides an effective and effecient way of capturing the dynamic nature of models. Network modules allow structural changes to be made to network models at run-time facilitating easy representation of model bahaviour. The toolkit allows the DEVS framework of modular design, and hierarchy to be applied to dynamic models. Thus complex dynamic behaviour can be easily represented by using simple basic and network models in a hierarchical design.
In the examples presented in this paper, we showed that the dynamic nature of the models can be easily produced using simple models. DS-CD++ allows dynamic systems to be modeled by simple models which would otherwise require complex models.

References

[1] Fernando J. Barros, Bernard P. Zeigler and Paul A. Fishwick *Multimodels and Dynamic Structure Models: An Integartion of DSDE/DEVS and OOPM*, Proceedings of the 1998 Winter Simulation Conference

[2] Jerry Banks, John S. Carson II, Barry L. Nelson, David M. Nicol *Discrete-Event System Simulation* , Prentice Hall, 2005

[3] Hui Shang, Gabriel Wainer *A simulation Algorithm for Dyanamic Structure DEVS Modelling* Proceedings of the 2006 Winter Simulation Conference

[4] Gabriel Wainer *CD++: A Toolkit to Develop DEVS Models* In Software, Practice and Experience. Wiley. Vol 32, No 3. November 2002. pp 1261-130. CA, TO

[5] Engine af *Engine Advances*; cited 13-02-2008, Available at:
www.gm.com/explore/fuel_economy/engine_advances/