Dynamic Structure DEVS: Improving the Real-Time Embedded Systems Simulation and Design

Hui Shang Gabriel A. Wainer
Department of System and Computer Engineering, Carleton University
1125 Colonel By Drive. Ottawa ON. K1S 5B6, Canada
{shanghui, gwainer} @ sce.carleton.ca

Keywords: DEVS, Dynamic Structure DEVS, Real-Time Systems, Real-Time Simulation, Experimental Environment.

Abstract

We present an improved simulation engine that combines dynamic structure DEVS with the real time simulation engine used in eCD++, a DEVS-based real-time experimental environment. The improved simulation engine is implemented in an advanced dynamic structure real-time experimental environment, namely DS-eCD++ (Dynamic structure Embedded CD++) to adjust the model structure automatically according to the changes in the internal/external environments. The existing abstract simulators, which are simulation, coordinator and root coordinator, were redefined to adapt to dynamic structure. In addition, a new abstract simulator, revsimulator, was developed to implement the structural change behaviours of Structure Agent that executes the structural changes. We discuss how the simulation engine works, and introduce the implementation of DS-eCD++, including a case study. With this dynamic structure function, DS-eCD++ exposes to more rigorous challenges in terms of reliability and flexibility of Real-Time embedded system and facilitates the design and development of real-time embedded systems.

1. INTRODUCTION

DEVS (Discrete EVent System Specification) [1] is a sound mechanism for Modeling and Simulation (M&S) of discrete-event dynamic systems. This methodology has recently gained popularity in real-time applications due to the fact that it enables not only an interactive simulation but also a smooth transformation from models to executing code in real-time environments [2-5]. Applying hardware-in-the-loop [6] to develop an intelligent device is a helpful attempt. MDA (Model-driven Architecture) is a high level scheme to support the transformations from simulation modeling to design of real-time systems. Based on the MDA technology, eCD++ (embedded CD++), integrating the real-time simulation engine, provides a real-time DEVS-based experimental environment, permitting developing hybrid hardware and software systems [6-7]. However, eCD++ cannot provide effective support to the real-time embedded systems residing in changing environments due to the absence of dynamic structure. Dynamic structure is a feasible solution to fitting the varied environments or recovering from errors automatically. Flexibility and reliability, therefore, could be reached by adjusting the structures of models dynamically.

Flexible Dynamic Structure DEVS algorithm (FDSDE) [8] defines a set of new message-passing algorithms to support dynamic structure. Here we present an improved simulation engine that combines dynamic structure DEVS with the real time simulation engine to adapt to not only the dynamic structure real-time simulations but also the real-time embedded system development. The software with the improved simulation engine is called DS-eCD++. The conception of Structure Component is introduced to represent the coupled models which are subject to structural changes. Each Structure Component is furnished with a Structure Agent to implement the structural changes. An abstract simulator, revsimulator, is defined to process the messages for Structure Agents. DS-eCD++ is able to run dynamic structure real time simulations. Moreover, DS-eCD++ takes advantages of the major functionalities in eCD++, such as GGAD notation, the flattened coordinator technique and the transition mechanism. DS-eCD++ can act as a powerful platform to undergo real-time embedded systems development with dynamic structure.

The structure of the paper is organized as follows: the second section depicts the background; the third section describes the FDSDE algorithm; the fourth section explains the algorithm implementations; the fifth section discusses the case studies.

2. BACKGROUND

DEVS is a formal specification M&S framework [1]. In DEVS, a basic component, which is called *atomic*, is specified as a black box with a state and a duration for that state. DEVS has well-defined concepts for coupling of components and hierarchical, modular model composition, which permits defining composite models, called *coupled*.

DEVS allows defining abstract simulation algorithms that are independent from the model definitions and model behaviours. Simulators progress through the simulation by exchanging messages according to the described message-passing mechanisms.

P-DEVS (Parallel DEVS) [9] is an extension to the original DEVS formalism that permits dealing with simultaneous events in an effective fashion. The semantics of the P-DEVS definition introduces inputs in bags (multi sets that can receive multiple events before starting processing them), and a confluent transition function, which handles the collision behavior when an external event arrives at the same time of its internal transition. Therefore, all imminent components can be activated in parallel.

RT-DEVS [4] allows DEVS models to interact with their surrounding environments (i.e., software components, hardware components or human operators), in real time. Thus, RT-DEVS presents a better adaptation to real-time environment. It also permits developing real-time embedded systems using hardware-in-the-loop technology. The RT-DEVS formalism replaces virtual time advance in the DEVS formalism with real- time advance. The time advance function is no longer a fixed value. Instead, a time interval is defined. The RT-DEVS simulator checks a specified time advance of a RT-DEVS model against a real time clock.

The DEVS simulation provides a good framework for Real-Time systems development, because its mathematical foundation, the well-defined concepts of coupling of components, hierarchical, modular model construction, support for discrete event approximation of continuous systems and an object-oriented substrate supporting repository reuse is well tailored for real-time systems development. Nevertheless, we need to address the dynamic adaptation to dynamic changes in the environment.

Dynamic structure systems focus on the possibility to dynamically change the model structure according to the real requirements, which is useful for real time systems (in which sometimes it is impossible to interfere with the running of the system manually, and self-adaptation is needed) In this way, the system is more feasible to fit the internal/external environments. Dynamic Structure DEVS allows addressing some of these issues. Based on DEVS theory, Dynamic Structure DEVS is a simulation paradigm supporting structural changes to full extent, ranging from simple model/connection addition/deletion to the exchange of models between networks of models [10]. DSDE [11-12] (Dynamic Structure Discrete Event System Specification) is an extension of DEVS formalism in dynamic structure. Two model groups, basic models and network models, are defined in terms of models' structure. The basic models are atomic structure units which cannot be split. Network models are the components consisting of multiple basic structure models and interconnections. A network executive is a modified basic model to execute structural changes in the network models. In a network model, network executive is the only component to conduct structural changes, which enables executing the structural transitions sequentially without any conflicts. DSDE is the theoretical basis of the FDSDE algorithms.

We have defined FDSDE and implemented it based on the CD++ toolkit [13], an object-oriented software implementation of the DEVS simulation mechanism. Atomic models are defined using a state-based approach (encoded in C++ or an interpreted graphical notation); while coupled models contain model's composition and interconnection information of those models. A *Simulator* is in charge of executing the behaviour of atomic models while a *Coordinator* copes with the messages in coupled models. The simulation evolves through message-passing, using six kinds of messages: I (Initialization), * (Internal), X (Inputs), Y (Output), @ (Collect) and D (Done).

eCD++ [14] is a version of the CD++ software family that has been adapted for real time and embedded system applications. The software is modularized as a group of components that have well-defined behaviors and have independent functionalities. Four major components are included: Main Simulator, DEVS Modeling Subsystem, Simulation Subsystem and Messaging Subsystem. It is based on the P-DEVS formalism, which provides the modeling principles to characterize the structural and behavior aspects of real-time embedded systems. Moreover, RT-DEVS enables eCD++ to simulate the hybrid software and hardware systems. Finally, eCD++ supports smooth transformations from simulation models to real components of the systems. The flattened coordinator in eCD++ provides an alternative simulation fashion by eliminating the coordinators in the processor hierarchy and exchanging messages directly between the flattened coordinator and the simulators. The GGAD interpreter (Generic Graphical advanced environment for DEVS modeling and simulation) in eCD++ enables to specify atomic models graphically. It is an easier way for the non-expert users to build atomic models intuitively.

3. FLEXIBLE DYNAMIC STRUCTURE DEVS ALGORITHM

FDSDE [8] defines four kinds of abstract simulators in which a group of receiving functions specify the message-passing paradigms for the DEVS models. The *root coordinator*, *coordinator* and *simulator* were redefined to accommodate to structural change processes. In FDSDE, a *Structure Agent* is defined to specify the structural change behaviours in a Structure Component. Therefore, our new abstract simulator (named *RevSimulator*) generates the structural change behaviours of S*tructure Agents*. The *flattened coordinator* is also redefined to adapt to the dynamic structure.

To do so, the FDSDE algorithm defines three new message types to carry the messages related to structural changes, i.e. * (sc) (structural change message), D (sc) (structural change request) and St (start message).

The detailed definition of FDSDE, presented in [8], defines how the simulation evolves. A message-passing scenario presents the message flow among the simulation processors in Fig. 3. The model structure change of C1 and the corresponding processor hierarchy change are shown in Figure 1 and Figure 2. In Figure 3, RC denotes the Root Coordinator; S1, S2 and S3 are three simulators in charge of message processing of the three atomic models A1, A2 and A3. C indicates the parent coordinator of the three simulators handling the messages of the coupled model C1. Initially, the coupled model C1 contains two atomic models A1 and A2, which are associated with S1 and S2 respectively. Another atomic model A3 which is associated with S3 is added to the coupled model. RS is a revsimulator of a Structure Agent, which implements the structural changes for the coupled model C1. The message flow is presented using an event precedence graph, where a vertex denotes a message, and a directed edge represents an action of sending a message. The message type with a time stamp is marked beside the edge. The simulation stages of the structural change process are indicated at the bottom of Fig.3.

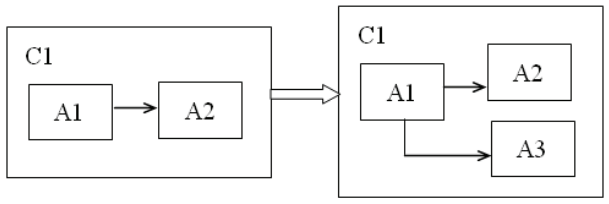

Figure 1. The model structure change of C1

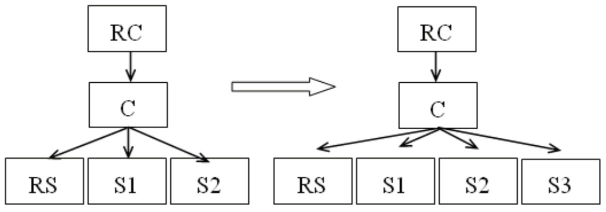

Figure 2. The processor hierarchy change

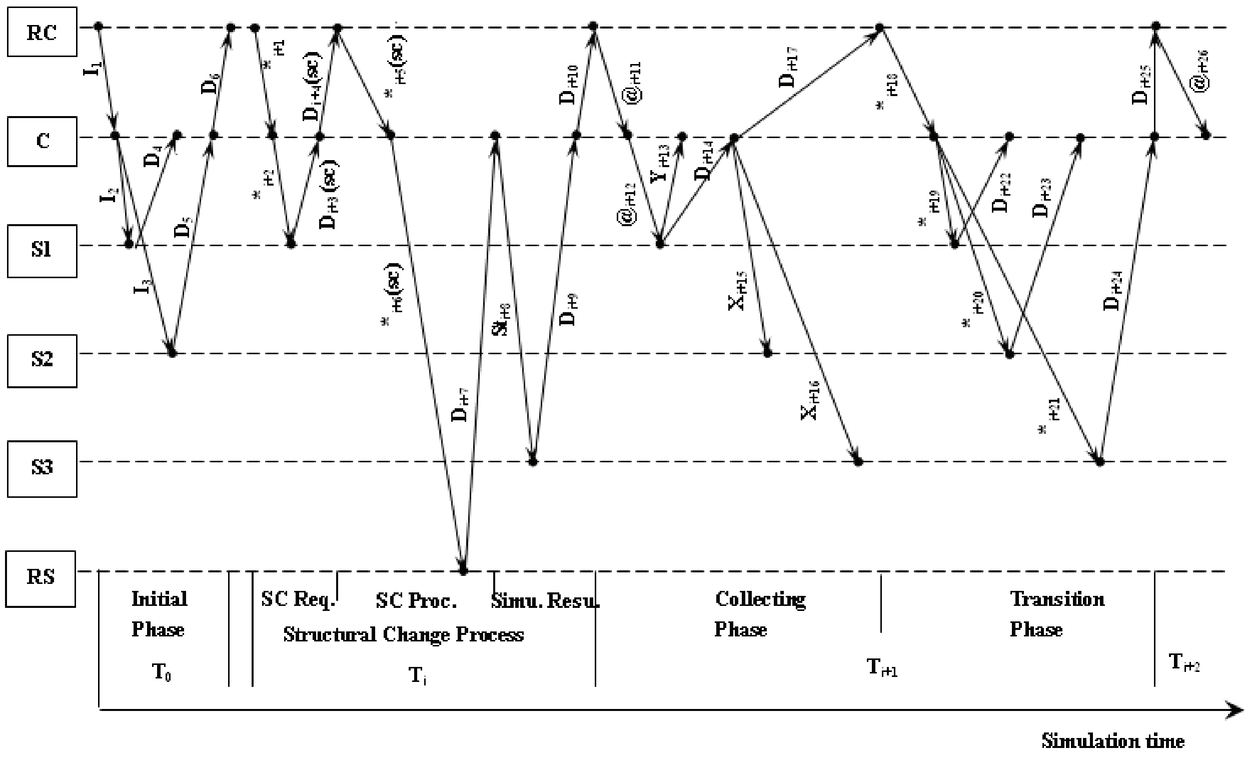

Figure 3. A Message-Passing Scenario

Initially, I_1, I_2 and I_3 initialize C1, S1 and S2. Correspondingly, D_4, D_5 and D_6 are answered. For simplicity, we skip the regular simulation cycles in this scenario. It the simulation cycle T_i, RC sends $*_{i+1}$ to C1 and then C1 sends $*_{i+2}$ to S1 because S1 is the only model that needs to be synchronized in this cycle. Assume that S1 raises a structural change request, the D_{i+3} (sc) is sent back to C1 and C1 passes the structural change request to RC using D_{i+4} (sc). Instead of issuing an @ message to enter the collect phase, RC issues a structural change message $*_{i+5}$ (sc) to C1 and C1 delivers it to RS1 (the processor of the associated structure agent) through $*_{i+6}$ (sc). The message D_{i+7} is answered by RS1 after the structural change finishes. Suppose a new atomic model A3, with its corresponding simulator S3, needs to be added to the simulation. First, C1 initializes S3 using St_{i+8}.

Then, S3 answers C1 with D_{i+9}. Next, C1 determines the minimum tN and sends it to RC by D_{i+10}, notifying the completion of the structural change process. After this structural change process, S3 joins the simulation. The simulation advances and RC sends $@_{i+11}$ to collect outputs in the simulation cycle T_{i+1}. Suppose S1 is the receiver of the collect message $@_{i+12}$. It returns an output message (Y_{i+13}) and the done message (D_{i+14}) to C1. Since S2 and S3 are the influencers of S1, C1 converts the output message (Y_{i+13}) into the proper input messages and routed them to S2 (X_{i+15}) and S3 (X_{i+16}) respectively. S1, S2 and S3 are all cached into the synchronization set of C1. C1 sends D_{i+17} to RC, marking the end of the collect phase in the cycle T_{i+1}. In the transition phase of T_{i+1}, the internal messages are dispatched to each model (A1, A2 and A3) and trigger the transition functions ($*_{i+18}$, $*_{i+19}$, $*_{i+20}$, $*_{i+21}$). Accordingly, tNs are returned to RC with the done messages (D_{i+22}, D_{i+23}, D_{i+24}, D_{i+25}). Then, RC sends a collect message ($@_{i+26}$) to start the next simulation cycle T_{i+2}.

4. ALGORITHM IMPLEMENTATION

DS-eCD++ takes advantages of the four software components: the *Main Simulator*, the *DEVS modeling Subsystem*, the *Simulation Subsystem* and the *Messaging Subsystem*. However, substantial modifications have been made to fit the new features. These modifications are characterized as follows:

a) Main Simulator takes charge of separating the model definition into two groups: the active components that participate in simulation and the structure components. It loads the active components, including coupled models, atomic models and structure agents, into the simulation system. In a simulation using a flattened coordinator, Main Simulator takes charge of storing the initial model compositions and the couplings used in simulation.

b) Model Hierarchy Tree is composed of atomic models, coupled models and structure agents. It is updated by Structure Agents. In Simulation Subsystem, the receive function in each abstract simulator class are redefined to implement the message-passing algorithms in FDSDE. *RevSimulator* is an abstract simulator class processing messages for structure agents. Simulators/coordinators Hierarchy Tree in DS-eCD++ includes *Root, Coordinator, Simulator* and *RevSimulator. FlatDEVSCoordinator* is redefined to implement the flattened coordinator in the simulation with a flattened processor structure. The five abstract simulators constitute the improved simulation engine in DS-eCD++, supporting dynamic structural changes in real-time simulation.

c) The messages related to structural changes lead to the expansions of some messaging classes in Messaging Subsystem. *InternalMessage* class and *DoneMessage* class are reused to carry the structural change messages and the structural change requests by appending a non-zero value in each of these messages. A new message class *StartMessage* is created for the St message.

4.1. Structure Agent and the Simulation Processor

Two classes, *RevAtomic* and *RevSimulator*, are created to specify the model behaviours of Structure Agents and to realize the message-passing algorithm defined in the abstract simulator *RevSimulator* respectively.

RevAtomic specifies the structural change behaviours of Structure Agents. A group of structural change operations are encapsulated in the *RevAtomic* class, including the structural change operations and the supplement operations:

```
◆ Structural change operations:
  ~DiffModel()
  ~DiffLink()
  ~Getmodelset()
  ~Getlinkset()
  ~Getportset()
  ~AddModel()/~FullAddModel()
  ~DelModel()/~FullDelModel()
  ~AddLink()
  ~DelLink()
  ~AddInputPort() / ~AddOutputPort()
  ~DelInoutPort() / ~DelOutputPort()
◆ Supplementary operations:
  ~FindModel()
  ~FindInputPorts() / ~FindOutputPort()
```

Structural change operations provide necessary manipulations to the component properties (models, links and ports). The *Get* actions retrieve the specified component properties. The *Diff* actions aim to calculate (identify) the differences between the component properties to be changed and those to be added. The *Add/Del* and *FullAdd/FullDel* actions realize adding / removing of the component properties. The operations can be performed in two ways: full operations and simple operations. Full operations add or remove the atomic model objects, the associated simulator objects and their model references. Simple operations only add /remove the model references of the atomic models to/from the corresponding Structure Component, while keep the model objects and the associated simulator objects. The full operations are suitable for new atomic models that will be added to or removed from simulation system permanently. For the models that are temporarily removed in simulation and may be reused later, the simple operations can be applied. The two types of structural change operations provide operational flexibilities for modellers to keep balance between minimum memory usage and fast loading time. Supplementary operations are used to locate the component elements. These structural change operations can be called by concrete structure agents to define real structural change behaviours.

The behaviours of Structure Agents are represented by *InitFunction,* which is used to initialize the Structure Agents and *InternalFunction* which processes the structural changes. A model of a structure agent can be created by the modeller by including a new class derived from *RevAtomic* class. The model behaviours are defined in the *InitFunction* and *InternalFunction*.

RevSimulator, a class of simulation processor, executes the message-passing algorithm to generate the behaviours of structure agents. An instance of a processor of a Structure Agent is created from the class *RevSimulator*. The processor is in charge of the message processing of the corresponding Structure Agent by invoking the *InitFunction* to initialize the model when an initial message is received, and triggering the *InternalFunction* to process the structural changes according to the expected structural change variables when a structural change message is received.

4.2. New Relationships

Dynamic structure changes produce new relationships among the models and the simulation processors. Three kinds of interconnections are highlighted:

1. The structural change conditions are rooted in the atomic model behaviors (the model transition functions built by modelers). When any of the external/internal/confluent transition function of an atomic model is invoked, the structural change conditions are examined. Once the structural change conditions are satisfied, the atomic model will raise a structural change request to its associated simulator. A structural change request is sent out. The atomic model assigns the expected structural change command through *strucChange(int &)* which calls the homonymous method in the simulator associated with the atomic model and updates the data member `struc` in the simulator. `struc_rec,`an other data member in the simulator, holds a copy of `struc.` When comparing `struc_rec` with *strucChange()* which retrieves the value of `struc` in the simulator, we can identify the difference. This difference can be regarded as a structural change request from the atomic model. Also the simulator can raise a structural change request with this value from *strucChange()* and sends it to the upper level coordinator.
2. The consistency of the model structure between a Structure Component and the associated Structure Agent. The structural changes implemented by the Structure Agent cause the changes of the model composition and the couplings. The model composition and the couplings should be updated accordingly in the Structure Component. A pair of methods *AddModel()* and *DelModel()* are invoked by the Structure Agent to update the model composition. The couplings in the Structure Component are updated through modifying the influences list of the corresponding ports.
3. The processor of a Structure Component keeps track of the structural changes and adjusts its simulating behaviours. The two lists in the Structure Component store the model compositions before and after the structural changes respectively. The processor invokes a pair of methods *getnewmodels()* and *getremmodels()*. *getnewmodels()* retrieves the new models which are initialized before the new simulation cycle starts, and *getremmodels()* gets the removed models which are deleted from the synchronization set.

5. CASE STUDY

In this section we will introduce the simulation of Dynamic Structure Automated Manufacturing System (DSAMS). This real-time application is composed of the dedicated stations that perform assembling and painting tasks on different products in a manufacturing plant. a conveyor belt is included to transport the products to/from those workstations. The DSAMS uses four components: a *Conveyor,* the *Engine Assembly (ES)* workstation, the *Painter* workstation and the *Controller Unit*, depicted in Figure 4. The *Controller Unit* is an atomic model used to control the actions of the *Conveyor* according to external inputs (which schedule the manufacturing of a given product). The *Conveyor* is a coupled model consisting of an *Engine* (to move the belt) and a *Sensor* (to detect the current position in order to decide when we need to stop the belt). It transports the products being manufactured to the other units, as indicated by *Controller Unit.* The *Engine Assembly* workstation *(ES)* is an atomic model, modeling a dedicated workstation standing beside the *Conveyor* to take assembling tasks. The second dedicated workstation - *Painting workstation (PS)* - is a coupled model containing a *Painter* (which paints the products) and two models of painting arms: *Chrome* and *Color*.

Initially, a product is supposed to be placed on the conveyor belt besides *ES*, waiting for the instructions from *Controller Unit*. All the external events influencing the system are received by *Controller Unit* through two input ports: *btn1A*, indicating that the product will be processed in *ES,* or *btn2A*, telling that the product will be processed in *PS*. *Controller Unit* receives these events and determines where to dispatch each piece by activating the engine of the conveyor belt (*active_A* and *direction_eng_A*). Besides, *Controller Unit* receives inputs about the position status of the moving products from *Sensor*. (*s1A* and *s2A*), and sends them out through the output ports *sta_disp_A* and *dirn_disp_A*. The value of *sta_disp_A* represents the workstation at which a product has arrived (i.e., ES = 11 and PS = 21), while the value of *dirn_disp_A* implies the moving directions of these pieces (0: stopped, 1: moving forward and 2: moving backward). In addition, *Controller Unit* receives the signals indicating the completion of the tasks in *ES* (through *st1_A*) or *PS* (through *st2_A*). Two LED output ports, *led1* and *led2,* are associated with the two workstations *ES* and *PS* respectively to indicate the position status of the moving products. For example, if a piece needs to be dispatched to *ES*, its corresponding LED, *led1*, turns on (value = 1), and when the product reaches, it turns off (value = 0). *Engine* receives instructions from *Controller Unit* via *active_A* that determines the workstation and *direction_eng_A* that designates the moving direction. The output port *s1a_eng* tells *Sensor* the current station an engine reached. *Engine* activates *ES* via *es_in* and receives the ending signal via *es_out*. The ports *ps_in* and *ps_out* are used to transfer start and end messages between *Engine* and *PS*. *Painter* initiates the chrome arm and color arm via *chrome_in* and *color_in*. The preparation done messages are returned from *Chrome* and *Color* through *chrome_out* and *color_out.*

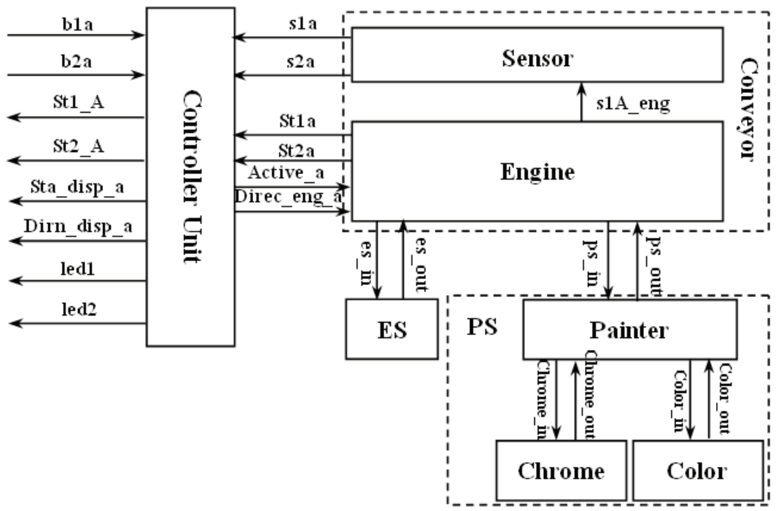

Figure 4. Scheme of Dynamic Structure Automated Manufacturing System

In this DSAMS system, the following two possible changes are considered:

1) Variation of the duties between *ES* and *ES1*. *ES* and *ES1* do the same duties but have different performances. *ES* has shorter working time than *ES1*. As shown in Table 1, *ES* takes 1 second to finish an assembling task while *ES1* needs 1 second and 50ms to finish the same kind of task. For simulation purpose, we suppose that ES and ES1 work for 10 seconds then the duty shift occurs.

Switch of painting modes. Some products need painting both color and chrome (painting mode = 1) while other require painting either color (painting mode = 2) or chrome (painting mode = 3). The painting mode is determined by the external events, in which the event values represent the painting mode.

Table 1 shows the timing parameters used in the AMS for each of them.

Table 1. Timing parameters in AMS

Models	Variables	Durations	Descriptions
Engine	Time2Start	00:00:00:005	Start a product
	Time2Stop	00:00:00:005	Stop a product
	movingTime	00:00:00:005	Move a product between stations
ES	workingTime	00:00:01:000	Working time
ES1	workingTime	00:00:01:050)	Working time
Painter	workingTime1	00:00:02:020	Paint both color and chrome
	workingTime2	00:00:01:000	Paint color
	workingTime3	00:00:02:000	Paint chrome
Color	preparationTime	00:00:00:010	Prepare the color arm
Chrome	preparationTime	00:00:00:020	Prepare the chrome arm

5.2. Experimental Results

This experiment aims to verify the dynamic structure of the simulation environment. The atomic models of the AMS were defined in C++, and the compositions and the couplings are specified in the coupled models. Two Structure Components are identified, *PS* and *TOP*. *PSEXEC* is a Structure Agent executing the structural changes on behalf of *PS* according to the indicated painting modes. *TOPEXEC* is another Structure Agent taking charge of the duty shifts between *ES1* and *ES* on behalf of *TOP*. The model definitions of DSAMS with the structure agents are displayed in Figure 5.

```
[top]
components : conveyorA dsecu@DSECU es@ES ps
components : topexec#TOPEXEC
SComm : structop1
in : …
out : …
Link : …
[topupdate1]
components : conveyorA dsecu@DSECU
components : es1@ES1 ps
SComm : structop2
…
[conveyorA]
…
[ps]
components : painter@Painter color@Color
components : chrome@Chrome psexec#PSEXEC
SComm : struc1
…
[psUpdate1]
components : color@Color painter@Painter
SComm : struc2
…
[psUpdate2]
components : chrome@Chrome painter@Painter
SComm : struc3
…
```

Figure 5. Model definitions of DSAMS

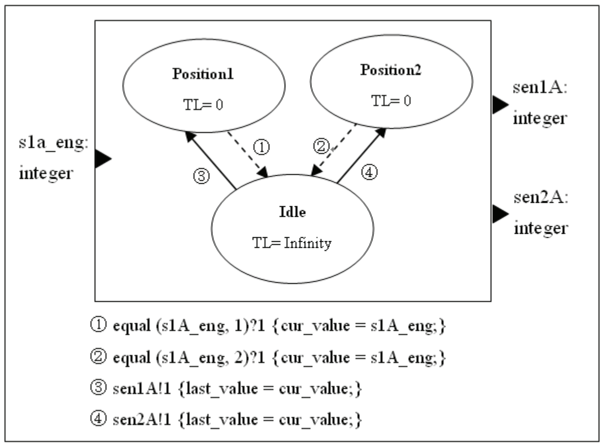
Position1
TL= 0
Position2
TL= 0
s1a_eng:
integer
sen1A:
integer
sen2A:
integer
Idle
TL= Infinity
①
②
③
④
① equal (s1A_eng, 1)?1 {cur_value = s1A_eng;}
② equal (s1A_eng, 2)?1 {cur_value = s1A_eng;}
③ sen1A!1 {last_value = cur_value;}
④ sen2A!1 {last_value = cur_value;}

Figure 6. GGAD graphical equivalent definition of Sensor

```
[Sensor]
in: s1A_eng
out: sen1A sen2A
var : cur_value last_value
state: idle position1 position2
initial: idle
ext: idle position1 equal(s1A_eng, 1)?1
{cur_value = s1A_eng;}
ext: idle position2 equal(s1A_eng, 2)?1
{cur_value = s1A_eng;}
int: position1 idle sen1A!1 {last_value =
cur_value;}
int: position2 idle sen2A!1 {last_value =
cur_value;}
idle: infinite
position1: 0:0:0:0
position2: 0:0:0:0
```

Figure 7. GGAD Definition of *Sensor*

Figure 6 and Figure 7 display GGAD definitions *of Sensor*, which is used to replace the *Sensor* defined with C++. It was found that the *Sensor* defined in the two ways behaved exactly the same and had the same simulation results. Since the GGAD notation can build equivalent atomic models with less effort than C++ definition, it is useful for non-expert modellers.

The simulation runs in a real-time mode. The external events in Table 2 were scheduled and sent to the *Controller Unit*. The first event in the table arrived at time 00:00:01:500 from the input port *btn1A*, which means the product would be transported to *ES* (or *ES1*). The associated output port of this event is *st1_A* and the output time should be no later than 00:00:03:500. The other events scheduled for *ES* (or *ES1*) are the fourth at time 00:00:12: 500, the sixth at time 00:00:19:985 and the seventh at time 00:00:25:000. The events scheduled for *PS* are the second at 00:00:10:500, the third at 00:00:10:500 and the fifth at 00:00:15:000. Among the events scheduled for *PS*, the values of the events shown in the last column designate the painting modes.

Table 2. The Table of the External Events

Event time	Deadline	Input port	Output port	Value
00:00:01:500	00:00:03:500	btn1A	st1_A	1
00:00:04:500	00:00:08:500	btn2A	st2_A	1
00:00:10:500	00:00:13:500	btn2A	st2_A	2
00:00:12:500	00:00:14:500	btn1A	st1_A	1
00:00:15:000	00:00:17:500	btn2A	st2_A	3
00:00:19:985	00:00:23:000	btn1A	st1_A	1
00:00:25:000	00:00:27:500	btn1A	st1_A	1

The simulation ran in real time mode. Four dynamic structure changes were identified during the simulation:

1. At 00:00:10:000, the scheduled work duration of *ES* was expired and a duty shift between *ES* and *ES1* occurred.
2. At 00:00:10:505 (supposing 5ms was used to activate the engine), *PS* switched its painting mode from 1 to 2. The *Chrome* model was removed, while the models of *Painter* and *Color* were maintained in *PS*.
3. At 00:00:15:015 (supposing the engine took 15ms to be activated and moved to *PS*), *PS* switched its

painting mode from 2 to 3. The *Color* model was replaced with the *Chrome* model.

4. At 00:00:20:000, *ES* was shifted by *ES1*. It was noticed that the input event reached *ES1* at time 00:00:20:000. Simultaneously, the scheduled internal state of *ES1* expired. Therefore, the confluent function of *ES1* was invoked at 00:00:20:000. In the confluent function of *ES1*, the external transition function has a higher priority over the internal transition function. Accordingly, at time 00:00:20:000, *ES1* executed the assembling task first, and then the duty shift happened.

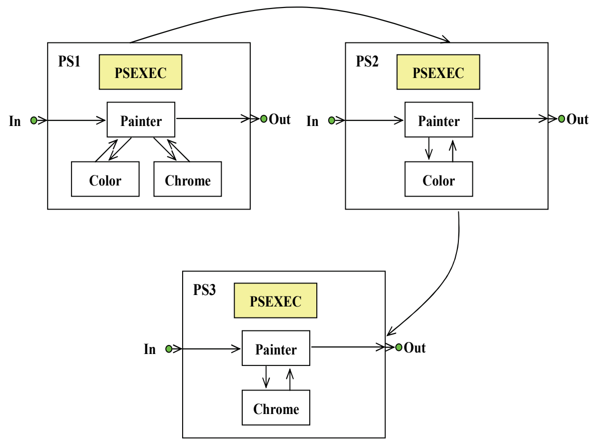

Figure 8. Structural Changes in *PS*

Figure 8 exhibits the structural changes in *PS*. Initially, *PS* was in painting mode 1 (structural state is *PS1*), which included both the color arm and the chrome arm. At 00:00:10:505, the painting mode switched to 2 (structural state is *PS2*), which included the color arm. *PSEXEC* executed the structural changes by deleting the links (*out@Chrome inchrome@Painter* & *outchrome@Painter in@Chrome*) and the *Chrome* model. The painting mode shifted to 3 (structural state is *PS3*) at 00:00:15:015. The links (*out@Color, incolor@Painter* and *outcolor@Color, in@Color)* were deleted, and the *Color* model was removed from the *PS* as well. The links (*out@Chrome, inchrome@Painter outchrome@Painter,* and *in@Chrome*) and the *Chrome* model were reused.

The structural changes in PS were triggered by the external transition function in the *Painter* model when the *Painter* model received a painting mode different from the previous one. Figure 9 shows how the structural change requests are raised from the external transition function in the *Painter* model. When the *Painter* model receives an external event, the value of the event specifies the painting mode. If this value is different from the previous value, the *Painter* model uses *strucChange(paintingmode)* to raise a structural change request.

```
Model &Painter::externalFunction (const
ExternalMessage &msg) {
     if(msg.port() == inA){
       if (msg.port() == inA) side = 1;

       paintingmode =(int)msg.value();
       if(paintingmode != scomm)
       {
         nextChange(Time::Zero);
         strucChange(paintingmode);
         scomm = paintingmode;
       }
       else{
…
  }
return *this;
}
```

Figure 9. The Structural Change Requests Raised from the External Transition Function in the *Painter* model

To some extent, the switch of painting modes in *PS* can also be realized in traditional static-structured simulation. However, the *Painter* model has to handle a more complex painting mode selecting mechanism. Also, extra memory is consumed because all possible models have to be loaded into simulation system. The complicated processing mechanism and uneconomical memory usage are not feasible in real-time embedded environment. Dynamic structure provides an elegant substitution by adjusting model structure automatically. The unused models can be deleted; therefore, the extra memory can be released immediately to maintain an economical memory usage. In addition, simpler model structure also leads to the simpler painting mode selecting mechanism which causes less running time. Moreover, dynamic structure is the only solution in some conditions. For instance, the structural changes in *TOP* are unavoidable to carry out duty shifts between *ES* and *ES1*.

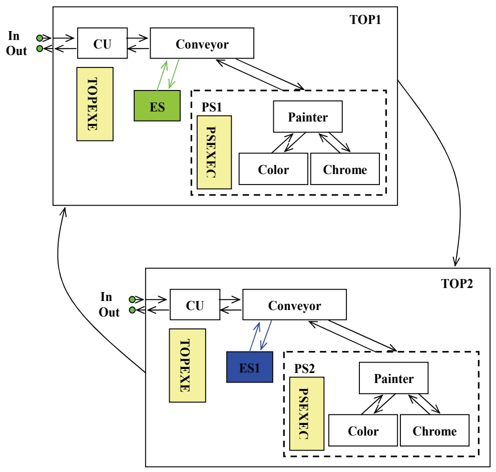

Figure 10. Structural Changes of the *TOP* model.

Figure 10 presents the structural changes in *TOP*. *TOPEXEC* executed the structural changes on behalf of *TOP*. *ES* and *ES1* switched every 10 minutes. At 00:00:10:000, *ES* (structural state is *TOP1*) was replaced by *ES1*(structural state is *TOP2*). The links (*out@ES es_in@Conveyor* & *es_out@Conveyor, in@ES*) and the model *ES* were replaced by the links (*out@ES1 es_in@Conveyor* & *es_out@Conveyor, in@ES1*) and the model *ES1*. As scheduled, the duty shift from *ES1* to *ES* would occur at 00:00:20:000. The real duty shift occurred at 00:00:21:500 for the confluent function gave higher priority to the external function of *ES1*. Consequently, the structural change, which happened in the internal function of *ES1*, has been delayed.

The structural changes in TOP are triggered in the internal transition function of *ES* or *ES1*. As scheduled, *ES* (*ES1*) will experience duty shifts every 10 seconds. Figure 11 presents how the structural change requests are raised from the internal transition function in *ES1*. Every time when *ES1* finishes assembling task (readyA = 1), the scheduled working time is checked. The structural change request is raised if the message time is equal or greater than the scheduled working time (duetime). Otherwise, the remaining working time is calculated (duetime – msg.time()). If the internal transition function is fired due to the expiration of the scheduled working time, the structural change request is raised immediately.

```
Model &Es1::internalFunction (const
InternalMessage &msg) {
  if (readyA){
    readyA = 0;
  if(msg.time() >= duetime)
       {
         backup = 4;
         strucChange(backup);
       }
   else if(msg.time() < duetime)
       {
         Time elapse = duetime - msg.time();
         holdIn(active, elapse);
       }
  }
  else if(struc == 0){
       backup = 4;
       strucChange(backup);
      struc = 1;
     }
  else if (struc == 1)
    {
    passivate();
    }
    return *this ;
}
```

Figure 11. The Structural Change Requests Raised from the Internal Transition Function in ES (ES1)

The simulation results are listed in Table 3. The first column shows the wall-clock value (the time elapsed since the beginning of the simulation execution) at which the outputs have been sent out. The second column is the expected deadlines. The results and the output ports are displayed in the third and the fourth column. The fifth column presents the values output from the output ports. According to the external event time and the timing parameters shown in the table 1 and the table 2, we have verified that the results reflect the external events correctly and meet the expected deadlines.

Table 3. Real Time Simulation Results

Output time	Deadline	Result	Out	V
00:00:02:510	00:00:03:500	Succeed	St1_A	1
00:00:06:560	00:00:08:500	Succeed	St2_A	1
00:00:11:520	00:00:13:500	Succeed	St2_A	1
00:00:14:030	00:00:14:500	Succeed	St1_A	1
00:00:17:040	00:00:17:500	Succeed	St2_A	1
00:00:21:510	00:00:23:000	Succeed	St1_A	1
00:00:26:020	00:00:27:000	Succeed	St1_A	1

A different test applied the flattened coordinator technique to the dynamic structure simulation of DSAMS. The *flattened coordinator* helps to improve the simulation performance by flattening the model hierarchy and reducing the number of messages delivered among the models dramatically. The *flattened coordinator* in DSAMS enables the atomic models to exchange messages with the *FLATTOP* directly. *FTOPEXEC* is the solo Structure Agent taking charge of the structural changes on behalf of *FLATTOP*.

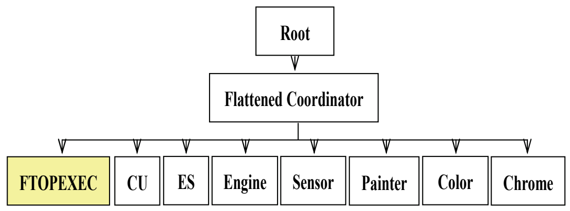

Figure 12. Simulation Hierarchy with a Flat Coordinator

Figure 12 exhibits the processor hierarchy using the flat coordinator. The coordinators of the structure components *PS* and *TOP* are replaced with the flat coordinator. The *FLATTOP* exchanges the messages directly with the atomic models, while *FTOPEXEC* executes the structural changes on behalf of *FLATTOP*. The simulation with the flat coordinator produces the same simulation results as those of Experiment 1, but played a higher simulating performance. The total messages exchanged among the processors in Experiment 1 are 1,104; while the total messages delivered in Experiment 2 are 703. The comparison of the numbers of messages between the two experiments is shown in Figure 13.

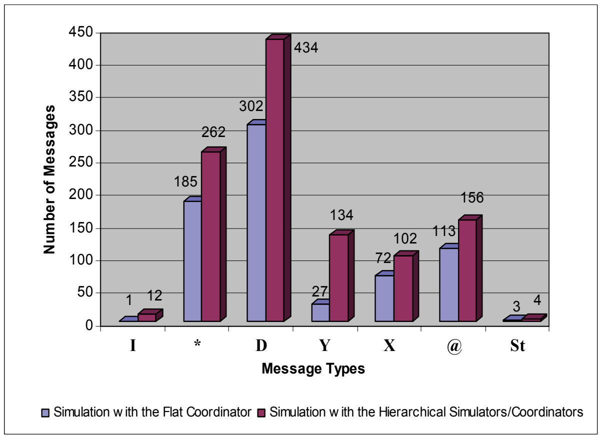

Figure 13. Comparison of the Number of Messages between Experiment 1 and 2

6. CONCLUSIONS

DS-eCD++ offers an advanced simulation engine by integrating dynamic structure DEVS and the real-time simulation engine seamlessly. The simulation engine supports varied dynamic structure change forms [15] and the real-time simulation. DS-eCD++ expanded the major software components of eCD++ and advanced adaptability to the real-time environment. The case studies demonstrated that DS-eCD++ supports real time simulations with various forms of dynamic structural changes. Also, it fits the GGAD notation and the flat coordinator technique.

In DS-eCD++, the real time running mode permits the execution of hybrid hardware and software simulation. This advanced real-time experimental environment provides a platform for real-time embedded systems development based on M&S for testing and development. Besides the seamless transformation from the simulation stage to the design stage of real-time systems, the dynamic structure allows defining the dynamic nature of real time applications more accurately.

REFERENCES

[1] Zeigler, B.P.; T.G. Kim; and H. Praehofer. 2000. Theory of Modeling and Simulation: Integrating Discrete Event and Continuous Complex Dynamic Systems. Academic Press.

[2] Cho, S., and T.G. Kim. 2001. "Real Time Simulation Framework for RT-DEVS Models". Transactions of the Society for Computer Simulation International.Vol. 18, No. 4, 203 – 215.

[3] Cho, Y. K.; B.P. Zeigler; H. J. Cho; H. S. Sarjoughian, and S. Sen. 2000. "Design Considerations for Distributed Real-Time DEVS". AIS 2000. Tucson, USA.

[4] Hong, J.S.; H. Song; T.G. Kim, and K.H. Park. 1997. "A Real-time Discrete Event System Specification Formalism for Seamless Real-time Software Development". Discrete Event Dynamic systems: Theory and Applications. Vol. 7, No. 4, 355-375.

[5] Kim, T.G.; S.M. Cho, and W.B. Lee. 2001. "DEVS Framework for Systems Development". In Discrete Event Modeling & Simulation: Enabling Future Technologies. Springer-Verlag. New York.

[6] Glinsky, E., and G. Wainer. 2004. "Modeling and Simulation of Systems with Hardware-in-the-loop". In the Proceedings of the 2004 Winter Simulation Conference. Washington DC, USA.

[7] Glinsky, E., and G. Wainer. 2004. "Model-Based Development of Embedded Systems with RT-CD++". In the Proceedings of the WIP session, IEEE Real-Time and Embedded Technology and Applications Symposium. Toronto, Canada.

[8] Shang, H., and G. Wainer. 2007. "Flexible Dynamic Structure DEVS Algorithm towards Real-Time Systems". Proceedings of Summer Computer Simulation Conference. San Diego. CA. USA.

[9] Chow, A.C., and B.P. Zeigler. 1994. "Revised DEVS: A Parallel, Hierarchical, Modular Modeling Formalism". In the Proceedings of the SCS Winter Simulation Conference. IEEE Computer Society Press, Los Alamitos, CA. USA.

[10] Barros, F.J. 1995. "Dynamic Structure Discrete Event System Specifications: A New Formalism for Dynamic Structure Modeling and Simulation". In the Proceedings of the 1995 Winter Simulation Conference, pp.781-785. Arlington, USA.

[11] Barros, F.J. 1997. "Modelling Formalisms for Dynamic Structure Systems". ACM Transactions on Modeling and Computer Simulation, Vol. 7, No. 4, pp. 501-515.

[12] Barros, F.J. 1998. "Abstract Simulators for the DSDE Formalism". In the Proceedings of the 1998 Winter Simulation Conference. Washington DC, USA.

[13] Wainer, G. 2002. "CD++: a toolkit to define discrete-event models". In Software, Practice and Experience. Wiley. Vol. 32, No.3, 1261-130.

[14] Yu, Y.H., and G. Wainer. 2007. "eCD++: an engine for executing DEVS models in embedded platforms". In the Proceedings of Summer Computer Simulation Conference. San Diego. CA. USA.

[15] Hu, X.L., B.P. Zeigler, and S. Mittal. 2005. "Variable Structure in DEVS Component-Based Modeling and Simulation". Simulation, Vol. 81, Issue 2, 91-102.