DEVS-C++© Reference Guide

31 October 1997

Hyup J. Cho and Young K. Cho

Artificial Intelligence and Simulation Research Group

Department of Electrical and Computer Engineering

The University of Arizona

Copyright © The University of Arizona

Table of Contents

1 INTRODUCTION **1**

2 DISCRETE EVENT SIMULATION **3**

2.1 MODELING AND SIMULATION 3
2.2 THE TYPES OF MATHEMATICAL MODELS 3
2.3 DISCRETE EVENT SYSTEM 4

3 FORMALISM: FORMAL SYSTEM DESCRIPTION **5**

3.1 PARALLEL DEVS FORMALISM FOR BASIC MODELS 5
3.2 DEVS FORMALISM FOR COUPLED (MULTI-COMPONENT) MODELS 8

4 DEVS ENVIRONMENT **10**

4.1 CONTAINER LIBRARY 10
4.2 DEVS LIBRARY 12
4.3 SIMULATION ENGINE 15

5 BASIC MODELS **17**

5.1 DEVS MODEL 17
5.2 ATOMICBASE AND ATOMIC MODELS 17
5.3 CELLBASE AND CELL MODELS 18
5.4 COUPLED MODELS 18
5.4.1 Instance Variables 19
5.4.2 Convert_input() 21

6 DERIVED CLASSES OF COUPLED CLASS **22**

6.1 DIGRAPH MODELS 22
6.1.1 Instance Variables 22
6.1.2 wrap_deltfunc Function 23
6.1.3 Output (compute_input_output) Function 23
6.2 BLOCK MODELS 23
6.2.1 Instance Variables 23
6.2.2 Coupling method of the Block 25
6.3 DIGCELL MODELS 25
6.4 COMPARISON OF *DIGRAPH, BLOCK*, AND *DIGRAPHCELL* MODELS 26

7 HIERARCHICAL DYNAMIC STRUCTURE **27**

8 TEST OPERATIONS **30**

8.1 CONTROL FLOW OF DEVS 30
8.2 INITIALIZE OPERATION 30
8.3 INJECT OPERATION 30

9 INSTALLATION OF DEVS-C++ IN UNIX **31**

9.1 STRUCTURE OF DIRECTORIES 31
9.2 HOW TO RUN TEST MODELS 32

APPENDIX **35**

A1. SOURCE CODE FOR BASIC (*ATOMIC*) MODEL EXAMPLE 36
A2. SOURCE CODE FOR GPT TEST PROGRAMS 38
A3. SOURCE CODE FOR HIERARCHICAL BLOCK MODEL TEST PROGRAMS 45
A4. SOURCE CODE FOR DIGRAPH CELL (DIGCELL) MODEL 55

List of Figures

FIGURE 1 MODELING AND SIMULATION ENTERPRISE 3
FIGURE 2 AN EXAMPLE OF DISCRETE EVENT SYSTEM : A QUEUEING SYSTEM 5
FIGURE 3 SYSTEMATIC REPRESENTATION OF A BASIC(*ATOMIC*) MODEL. 6
FIGURE 4 MODELING AND TRAJECTORY FOR SIMPLE PROCESSOR 7
FIGURE 5 VARIOUS REPRESENTATION OF COUPLED MODELS 9
FIGURE 6 AN EXAMPLE OF COUPLED MODELS: GPT MODEL 10
FIGURE 7 CLASS HIERARCHY FOR CONTAINERS 11
FIGURE 8 A MESSAGE (CONTAINER) STRUCTURE WITH 3 CONTENTS 12
FIGURE 9 CLASS HIERARCHY FOR DEVS MODELS IN DEVS-C++ 13
FIGURE 10 THE STRUCTURE OF HIERARCHICAL MODEL 14
FIGURE 11 COMPOSITION TREE 14
FIGURE 12 SIMULATION CYCLE OF COUPLED MODEL 15
FIGURE 13 TIME ADVANCE FUNCTION OF COUPLED MODEL 16
FIGURE 14 COMPONENTS DATA STRUCTURE 19
FIGURE 15 COUPLING DATA STRUCTURE 20
FIGURE 16 COMP_MAP STRUCTURE 21
FIGURE 17 CONVERT INPUT FUNCTION OF COUPLED MODEL 21
FIGURE 18 EXTERNAL TRANSITION FUNCTION OF DIGRAPH MODEL 22
FIGURE 19 COMPUTE INPUT OUTPUT FUNCTION 24
FIGURE 20 *DIGRAPHCELL* STRUCTURE 25
FIGURE 21 HIERARCHICAL DYNAMIC MOVEMENT 27
FIGURE 22 DYNAMIC STRUCTURE : MOVE SIDEWARD CASE 28
FIGURE 23 CONTROL FLOW OF DEVS 30
FIGURE 24 INJECT OPERATION OF DIGRAPH MODEL 31
FIGURE 25 DIRECTORY STRUCTURE OF DEVS-C++ 32

List of Tables

TABLE 1 THE ORGANIZATION OF THE DOCUMENT 2
TABLE 2 MATHEMATICAL MODELS IN TERMS OF TIME AND STATES 4
TABLE 3 DIFFERENCES AMONG DIGRAPH, BLOCK, AND DIGRAPH-CELL MODELS 26
TABLE 4 COMPONENT RELATIONSHIP AMONG DIGRAPH, BLOCK, AND DIGCELL MODELS 27

1 Introduction

Simulation is one of the most powerful tools for the planning, design, and control of complex processes or systems. Commercial simulators allow users to find solutions by simulating their models representing the real system or entity being modeled. In such commercial simulators, numerous variables and functions are pre-defined to support the users' requirements, providing the advantage of easy access and fast design for pre-defined types of models. However, most commercial simulators have severe limitations in modeling and running complex real world systems. From the dynamic structural point of view, the methods of adding or deleting a variable or a component during simulation run-time, are not provided because no command in the commercial simulators is supported for such a specific behavior. Were they to provide such a detailed behavior, the number of commands including different parameter formats would become too large and complex for users to easily seek appropriate commands. For example, the current number of the commands in the SIMAN simulator, which does not support a dynamic structural scheme, is too large for users to look up --64 pages to describe commands and variables. Therefore, limitation on the practical number of commands and parameter formats of the commercial simulators constrains the expression of complex behaviors. Most commercial simulators are incapable of extending the problem space or describing models deeper than some fixed level.

To overcome these limitations requires a methodology that provides the flexibility to design, and expandability to explore, the problem space. As a possible solution, DEVS (Discrete Event System Specification) environment is a system-theory based simulation tool that provides expandability with modular and hierarchical features. It offers flexibility with object-oriented messages as user-defined data structures. Based on a system theory called the DEVS formalism [1,2], the DEVS environment offers a library with which users can easily build models in a hierarchical modular way. In fact, the coupled model in the DEVS library is the major class to construct models hierarchically, while atomic models are the most basic classes. Due to the hierarchical property, users can quickly expand their models. Since a message in the DEVS environment carries an object--the data structure defined by the user, information as a message type can be delivered from one model to another. In other words, the size or scope of the object can be determined by the user, and can be different from one application to another, while the DEVS environment delivers the message containing the object from a source model to a destination. (However, in DEVS-C++, the user is responsible for creating and deleting the data structure in the

message). The DEVS environment helps overcome the complexity of the user's problem. Because models in the problem space can be developed to the depth and scope needed to meet the modeling objectives.

The DEVS formalism is theoretically well-defined means of expressing hierarchical, modular models in discrete event simulation. A DEVS model works as a timed state machine so that the state of the system is changed by external or internal events with elapsed time. DEVS-C++, based on the parallel DEVS formalism [4], is a modular hierarchical discrete event simulation environment implemented in the object-oriented C++ language.

The purpose of this document is to help users to understand and developers to extend the DEVS environment. The organization of this document is as shown in Table 1.

Table 1 The organization of the document

Chapter	Title	Contents
2	Discrete Event Simulation	- Types of mathematical models - Conceptual framework of modeling and simulation - Discrete Event Simulation
3	Formalism	- Parallel DEVS formalism - Basic and Coupled Models Formalism - Modeling examples
4	DEVS Environment	- *Container* Library - *Devs* library - Simulation Engine
5	Basic Models	- Devs, atomicbase, atomic models - Cellbase, cell models - Coupled models
6	Derived Models from a Coupled Model	- Digraph models - Block models - Digcell models - Comparison of digraph, block, and digcell
7	Hierarchical Dynamic Structure	- Hierarchical structure - Dynamic Structure
8	Test Operations	- Control flow - Initialize operation - Inject operation
9	Installation of DEVS-C++ in UNIX	- Structure of directories - How to run test models
Appendix	Test Examples	- Basic(atomic) model example - GPT test example - Hierarchical block model example - Digraph cell(digcell) model example

2 Discrete Event Simulation

2.1 Modeling and Simulation

Figure 1 shows a conceptual framework for the modeling and simulation enterprise. The real system, regarded as a source of behavioral data, is some part of the real world of interest. The model is a set of instructions for generating behavioral data of the form of plots of X(variable of interest) against T(time). The modeling relation, linking real system and model, concerns how well the model represents the system. The simulator exercises the model's instructions to generate its behavior. The simulation relation, which links model and simulator, concerns how faithfully the simulator can carry out the instructions of the model.

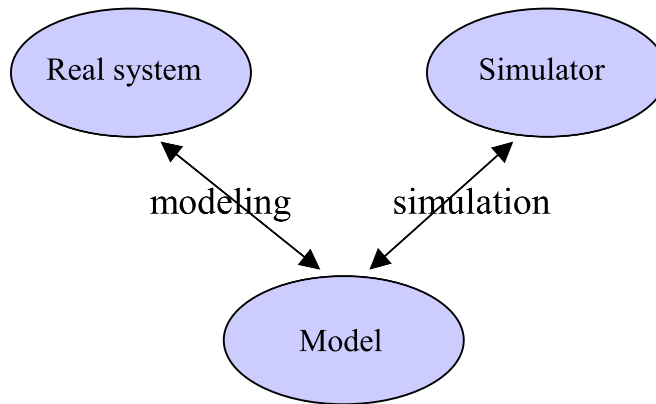


Figure 1 Modeling and simulation enterprise

2.2 The Types of Mathematical Models

There are several types of mathematical models in terms of time and states shown in Table 2. A continuous state variable changes over continuous time in continuous models, while discrete state variables range over discrete time in digital models. Continuous models are represented through sets of differential equations, and discrete time models through sets of difference equations. Qualitative models are continuous time models in which dependent variables are discretized. Sampled data models use continuous state variables over discrete time. Digital models can be represented through finite state machines.

Discrete event models which use continuous state and continuous time axis differ from continuous models by the fact that only a finite number of state changes may occur within finite time interval depending on instantaneous "events".

Table 2 Mathematical models in terms of time and states

Characteristics / Types of models	Mathematical Formalism And Application Area	Time	States
Continuous Models	Differential Equation Analog Circuits	Continuous	Continuous
Discrete Event Models	DEVS Formalism Distributed Systems	Continuous	Continuous
Sampled Data Models	Difference Equations Digital Signal Processing	Discrete	Continuous
Qualitative Models	Artificial Intelligence	Continuous	Symbolic
Digital Models	State Machine Digital Circuits	Discrete	Discrete

2.3 Discrete Event System

An event is usually a specific action such as customer arrival, a system going down, a stone hitting a window. Events occur instantaneously, and cause transitions from one state to another. The discrete event system is driven by events, and a typical example of discrete system is a queueing system shown as Figure 2.

Customers come in the queue randomly through an input port, and go out to an output port after the amount of time delayed in the queue. The state which is the number of customers in the queue will be changed by incoming customers(input events) or outgoing customers(output event). The behavior of a state transition function is to just add one when a customer comes in (the external transition function in DEVS-C++), or to delete one when a customer goes out from the system (the output and internal transition functions in DEVS-C++).

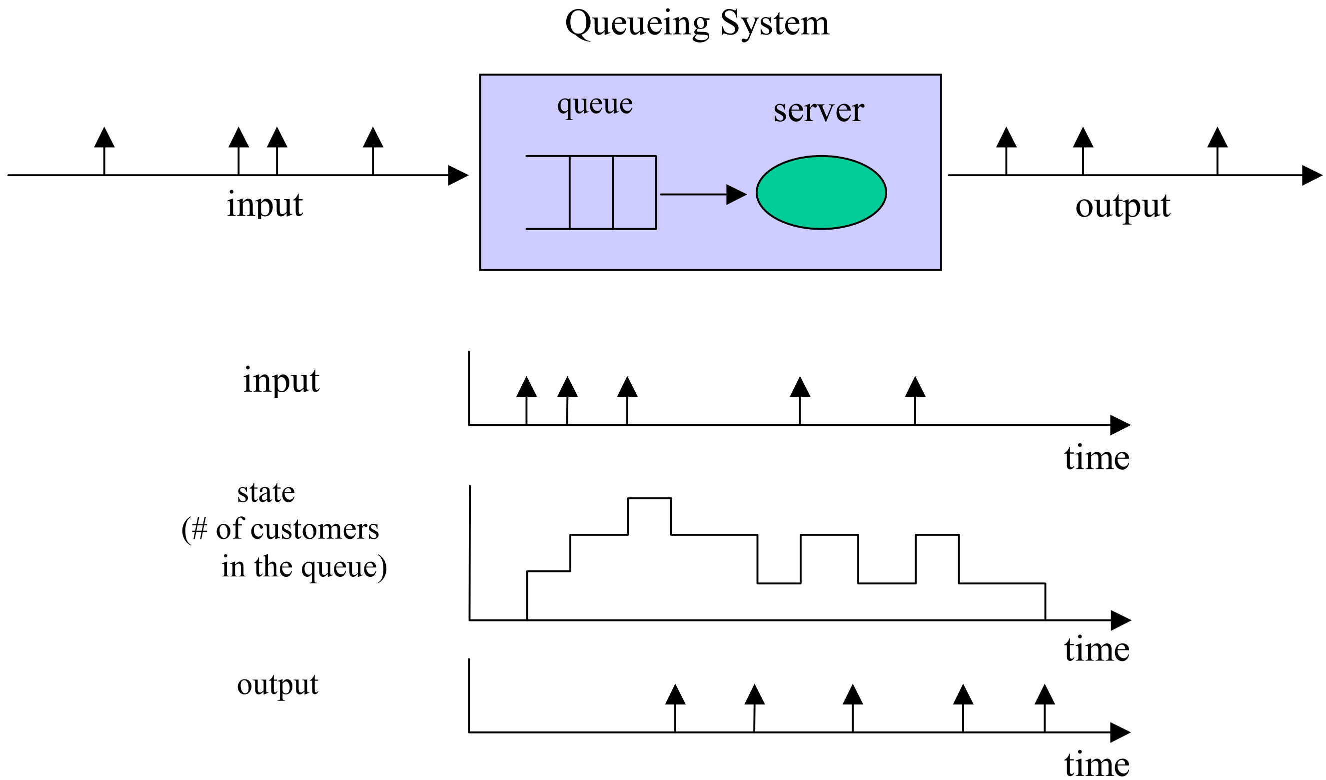


Figure 2 An example of discrete event system : a queueing system

3 Formalism: Formal System Description

The structure of a model(system) may be expressed in a mathematical language called a formalism. The discrete event formalism focuses on the changes of variable values and generates time segments that are piecewise constant. In essence the formalism defines how to generate new values for variables and the times the new values should take effect. An important aspect of the formalism is that the time intervals between event occurrences are variable.

DEVS-C++ has been developed based on DEVS formalism. Models, the basic constructs needed for modeling and simulation, can be specialized into two major classes, an *atomic* model and a *coupled* model. The *atomic* model realizes the atomic level of the model formalism, while the *coupled* model embodies the hierarchical model composition constructs [1].

3.1 Parallel DEVS Formalism for Basic Models

$$M = \langle X, S, Y, \delta_{int}, \delta_{ext}, \delta_{con}, \lambda, ta \rangle$$

Where

X : set of external input events;

S : a set of sequential states;

Y : a set of outputs;

δ_{int}: $S \rightarrow S$: internal transition function

δ_{ext} : $Q \times X^b \rightarrow S$: external transition function

X^b is a set of bags over elements in X,

(where $\delta_{ext}(s,e,\phi) = (s,e)$);

δ_{con} : $S \times X^b \rightarrow S$: confluent transition function;

λ : $S \rightarrow Y^b$: output function generating external events at the output;

ta : $S \rightarrow$ Real : time advance function;

Where $Q = \{ (s,e) \mid s \in S, 0 \leq e \leq ta(s) \}$

e is the elapsed time since last state transition.

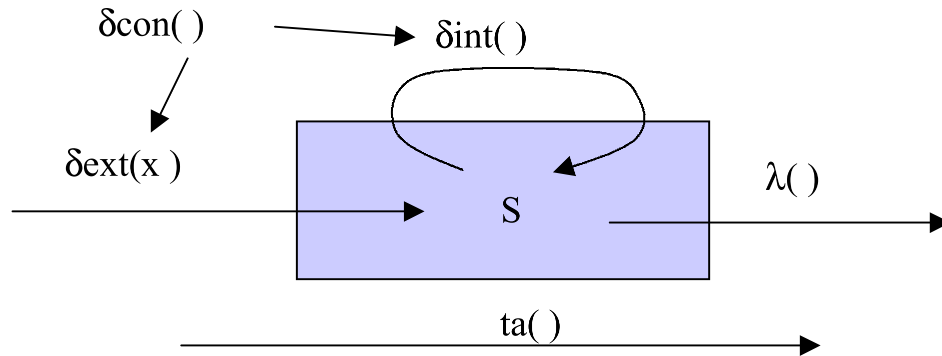


Figure 3 Systematic representation of a basic(*atomic*) model.

Figure 3 shows systematic representation of basic models. As an example, when an input packet arrives at the queueing system, the instance variable S which is the length of the queue should be incremented by 1. The function which carries the input and then changes the instance variable is the δ_{ext} function. After an elapsed time given by the *ta* function, the system checks the queue, it removes some packets causing the length of the queue to decrease by the amount of the removed packets. In other words, the instance variable, the length of the queue, changes internally from the previous state to the next state which is less by the number of the removed packets than

the previous one. Likewise, the internal function (δ_{int}) changes internal variables from previous state to the next. The δ_{con} function decides what to do when both external and internal events occure together. It might, for example, decide the order between δ_{ext} and δ_{int} functions. The λ function produces output Y from the instance variables. *ta* returns the time to the next internal event.

Consider a rather simplistic model of a processor as a basic model in Figure 4. The basic model has instance variables: sigma, phase, processing_time, and job_id. If the processor is idle("passive" phase) and a job arrives on the input port "in", it stores the job_id, changes the phase to "busy", and sigma(time left state variable) to processing_time. Such handling of incoming jobs is represented in the external transition function. When a job arrives while the processor is busy, "continue()" method updates sigma to reflect the passage of elapsed time, but leaves the next of the state unchanged. When the processor has finished processing, it places the job identity on port "out" and returns to the "passive" phase. This is done in the internal transition function. Sending of the job is done by the output function which is called just before the internal state transition function. Appendix A shows an example execution of single cycle of this model.

Figure 4 Modeling and trajectory for simple processor

3.2 DEVS Formalism for Coupled (Multi-component) Models

Two major activities involved in *coupled* models are specifying its component models, and defining the coupling which creates the desired communication links.

$$DN = \langle D, \{M_i\}, \{I_i\}, \{Z_{i,j}\}\rangle$$

Where

D is a set of components names;

for each i in D,

M_i is a component model

I_i is the set of influencees for i

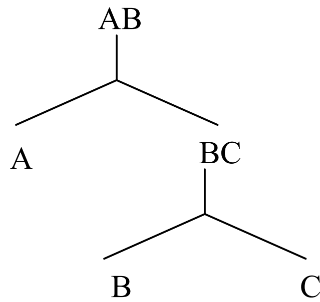
for each j in I_i,

$Z_{i,j}$ *is* the i-to-j output translation function

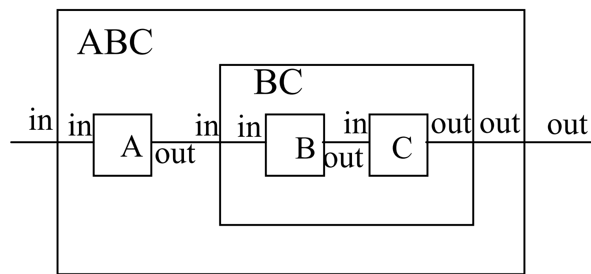
A *coupled* model contains the following information

- the set of components
- for each component, its influencees
- the set of input ports through which external events are received
- the set of output ports through which external events are sent
- the coupling specification consisting of
    - the external input coupling connects the input ports of the coupled to one or more of the input ports of the components
    - the external output coupling connects the output ports of the components to one or more of the output ports of the *coupled* model
    - internal coupling connects output ports of components to input ports of other components

Figure 5 depicts *coupled* models in terms of a hierarchical tree, the coupling relation, and the structure of coupled model. In Figure 5-a, *coupled* model ABC has, as its compoments, an *atomic* model A and a *coupled* model BC. In turn, BC's components are two *atomic* models, B and C. The Coupling relation, data path between models shown in Figure 5-b, is depicted by a line. Figure 5-c shows that Components is a container which holds items that can be atomic or coupled models. Pairs in the Coupling relation hold coupling information with port-to-port connections.

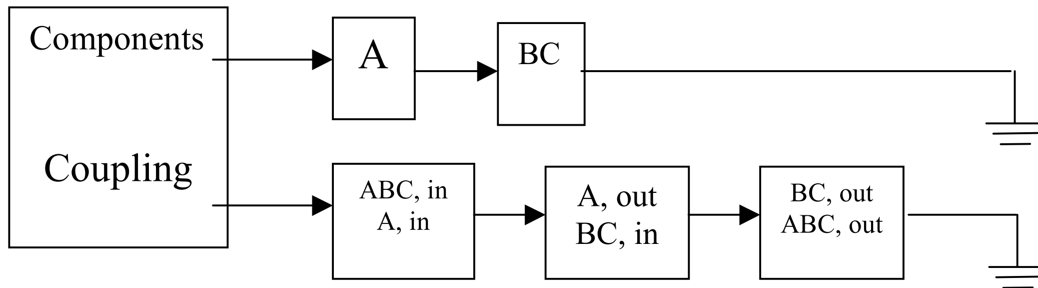


5-a) Hierarchical tree



5-b) Coupling relation

Coupled model : ABC



5-c) Coupled model structure

Figure 5 Various representation of coupled models

Figure 6 describes the GPT(generator/processor/transducer) architecture. Coupled model GPT has 3 components(D) which are G, T, and *P*. Note how the external input coupling connects the "*start*" port of GPT model to the "start" port of the enclosed G model. The external output coupling similarly connects the "*out*" port of *T(transducer)* to the "*out*" port of GPT. The output port of *G(generator)* is connected to the "in" port of *P(processor)* and to the "*ariv*" port of the *T* atomic model; the “out” port of *P* is connected to the "solved" port of *T*.

Once an external input signal “start” is injected, *G* generates its output periodically. The output of *G* goes to the input ports of *P* and *T*. The output of *P* goes to “*solved*” port of *T*. *T* gathers all information from *G* and *P*, and produces statistical information such as throughput and average processing time of *P*. Appendix B shows the source code of this GPT example model.

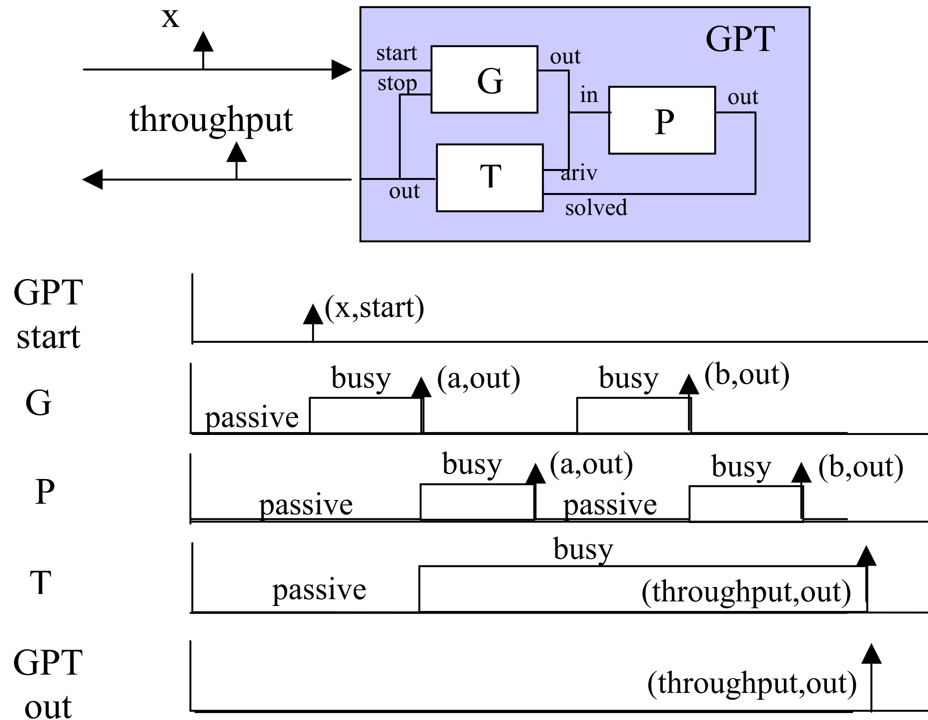


Figure 6 An example of coupled models: GPT model

4 DEVS Environment

There are two libraries, *container* and *devs*, to provide easy access to the DEVS environment. The *container* library, based on set theory, supports basic services to manipulate sets of entities. Container, bag, set, relation, and function classes are in the *container* library, including ordered lists such as stacks, queues, and lists.

The *devs* library supports several classes of basic models including atomic, cell, coupled, digraph, and block models. Using those predefined models, we can easily construct complex models in a hierarchical way, and build specific(user defined) models which would be expanded versions of basic models.

4.1 *Container* Library

The *Container* class, a generalized form of linked lists based on set theory, provides methods which can store, retrieve and organize interacting objects. A container object is an object containing other objects.

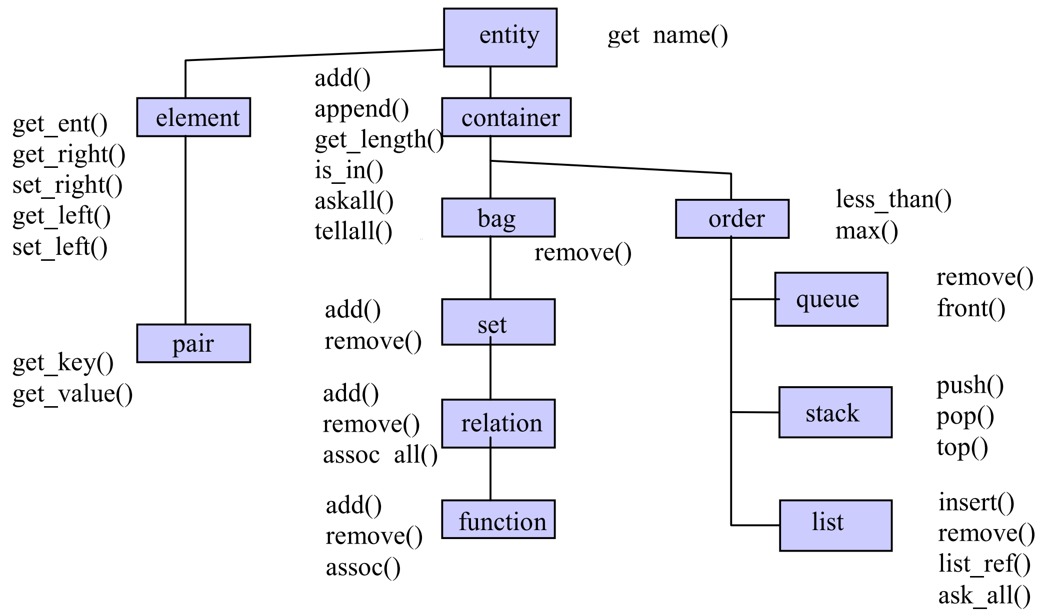


Figure 7 Class Hierarchy for Containers

Figure 7 depicts the *Container* class hierarchy. An *Entity* class will be the base class for all user-defined classes. It provides basically two methods, object name and a test of equality. The *Container* class provides basic services for the derived classes. The *Bag* class counts numbers of object occurrences. Only one occurrence of any object is allowed in the *Set* class. The *Relation* class consists of sets of pairs (key, value). The *Function* class is like the *Relation* except that at most one occurrence of any key is allowed. *Queues*, *stacks,* and *lists* maintain items in FIFO, LIFO, and ordered list respectively.

Figure 8 shows an example of a container class, specifically a message. Each content is in an element, and the elements are linked in a container. In DEVS-C++, the add method of a container,"add(entity **ent*)", automatically creates a new element having entity, *ent*, and puts the element into the container. The code for printing contents in the message illustrates how a message is implemented as a linked list.

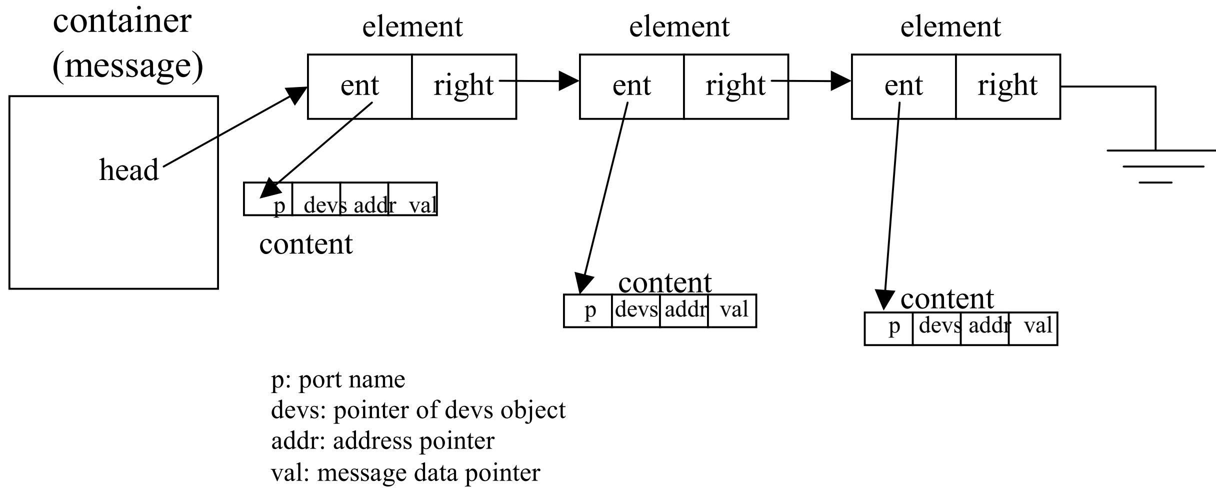


Figure 8 A message (container) structure with 3 contents

```
Element *el, *elnext;
For (el=mail->get_head(); el!=NULL; el=elnext) {
    Elnext = el->get_right();
    Content *con = (content *)el->get_ent();
    /* print message in the container */
    printf("port name: %s,", con->p);
    printf("DEVS name: %s,", con->devs->get_name());
    printf("address: %s,", con->addr->print());
    printf("message data: %s,", con->val->print());
}
```

4.2 *Devs* Library

The d*evs* class is the basic class to provide methods for the DEVS formalism. Such functions as δ_{ext}, δ_{int}, and λ are implemented as virtual methods to be defined by user. The method for time advance (*ta*) and port information is also provided by this class. The *inject* method is useful to send external input events to a model.

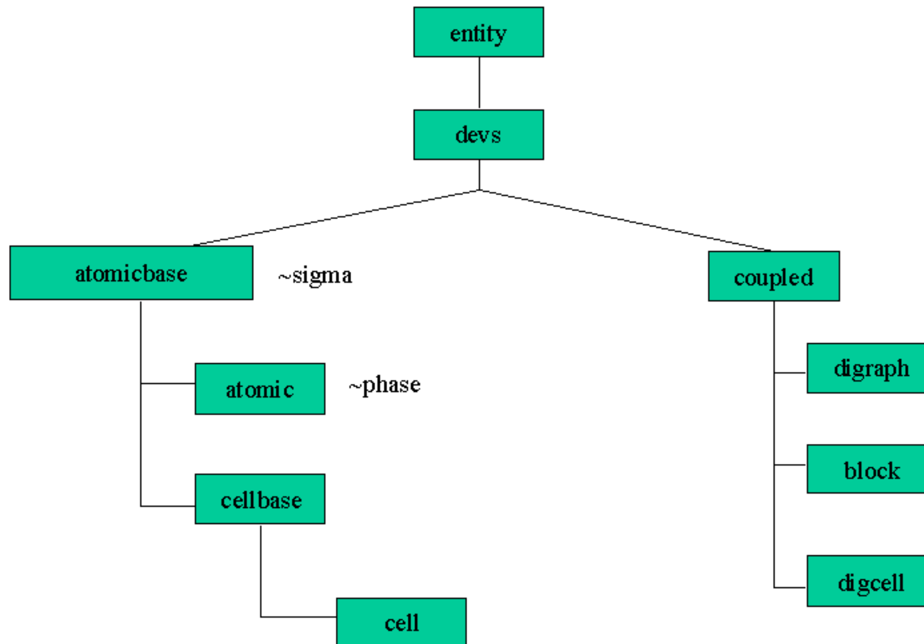


Figure 9 Class Hierarchy for DEVS models in DEVS-C++

As shown in Figure 9, an *atomicbase* has the *sigma* instance variable which holds the time remaining to the next internal event. This is the time-advance value to be produced by the time-advance function. The destination for sending output message is determined by coupling methods which manipulate coupling information. An *atomicbase* class realizes the atomic level of the underlying model formalism. It has variables corresponding to each of the parts of this formalism: internal transition function (δ_{int}), external transition function (δ_{ext}), confluent function (δ_{con}), output (out) function, and time advance function (*ta*). These methods are applied to the state of the model. The atomic class has a *phase* variable to describe model's phase.

The *cellbase* and *cell* classes are similar to *atomicbase* and *atomic* respectively. These classes are for cellular models and have address information (which *atomic* models do not). In this way, large numbers of *cellbase* or *cell* models can be manipulated in the same manner. Addresses are used for finding influencees.

The *coupled* class is the major class to support the hierarchical model composition constructs of the DEVS formalism. The d*igraph, block* and *digraphcell* are specializations which enable specification of *coupled* models in specific ways.

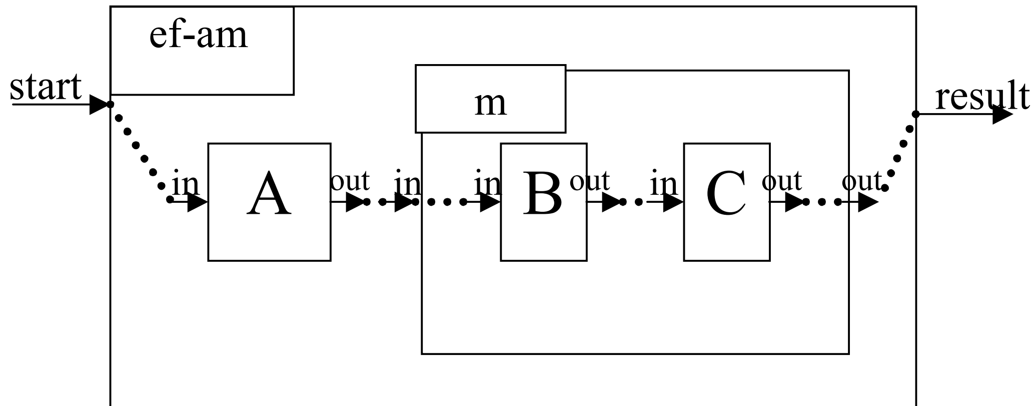


Figure 10 The Structure of Hierarchical Model

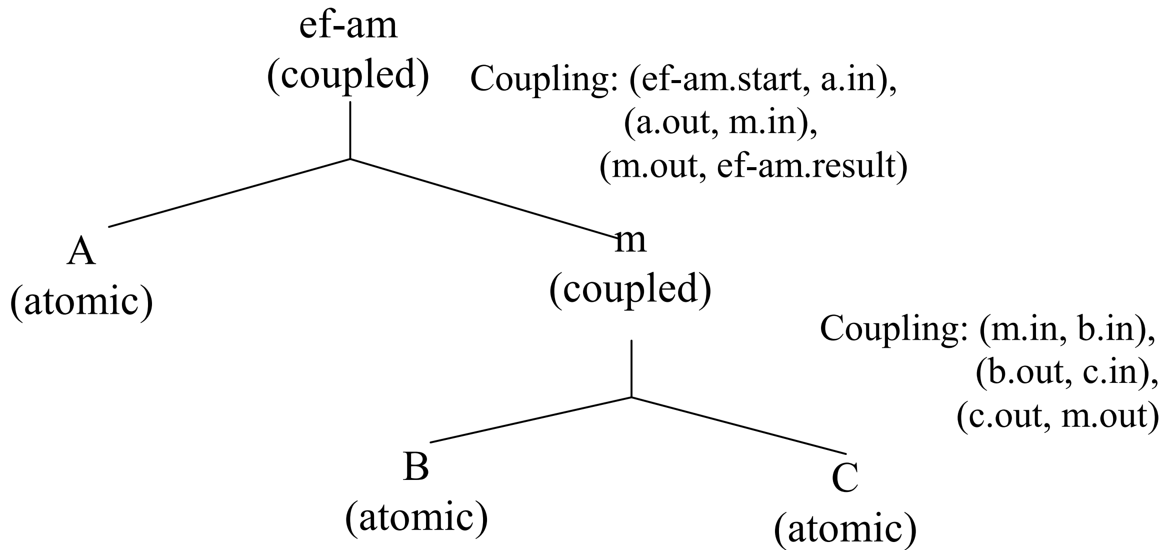


Figure 11 Composition Tree

A *coupled* model can become a hierarchical model. The structure of a hierarchical model as shown in Figure 10 is exhibited in a composition tree of Figure 11. The root model *ef-am* in Figure

11, is a *coupled* model. It has two components, an atomic model *a* and a coupled model *m.* The coupled model m also has atomic models b and c as its components. Appendix 3 gives source code for an example of hierarchical model.

A *coupled* model can have *coupled* or *atomic* models as its components, while *atomic* models become leaves in the tree. The coupling information needed to construct a *coupled* model is shown as Figure 11, attached to vertical line descending from the parent node to its children nodes.

4.3 Simulation Engine

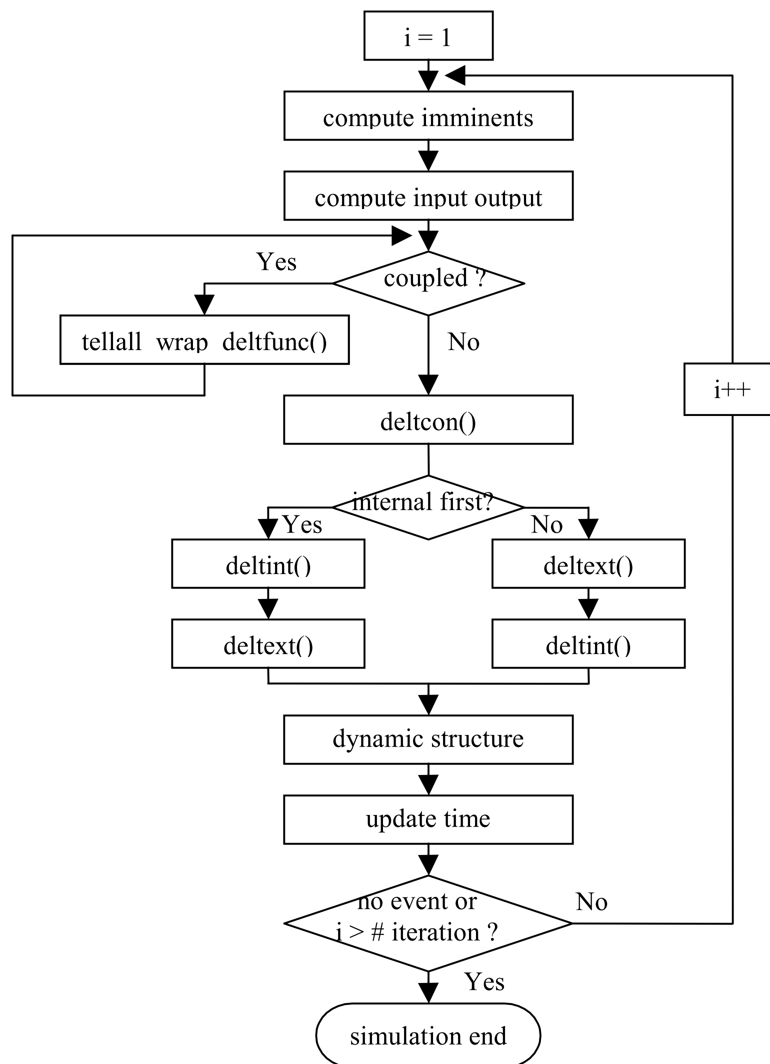


Figure 12 Simulation Cycle of Coupled model

Figure 12 shows the flow chart of a simulation cycle in a *coupled* model. The first step of a simulation cycle is to identify which components are imminent. Each imminent *atomic* component then generates its output in the compute_input_output phase. A *coupled* model accumulates all the outputs of its components, and also determines whether each output message of its components is for the internal use (input messages) or outgoing (output messages). After all imminents produce their outputs, the *coupled* model tells components to execute their delta functions.

Since all the components work simultaneously, an external input may arrive at a component at the very moment when its internal transition function should be executed. Via the confluent function a user can write tie-breaking rules such as selecting the order of the external transition or the internal transition function. By default, the internal transition fuction occurs first.

The internal transition function specifies to which next state a component will transit. For an atomic model, the effect is to place the component in a new *phase* and *sigma,* thus scheduling it for a next internal transition. Other state variables may be changed as well. The external transition function ($_{ext}$) with inputs from its influencees also specifies how the system changes state. For an atomic model, the next state is computed on the basis of the present state, the port and value of the input (external event), and the time that has elapsed in the current state.

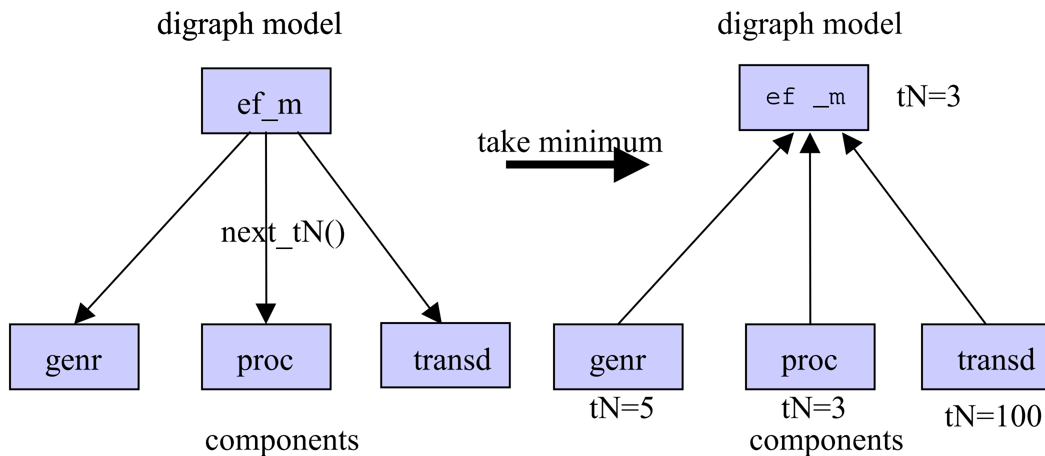


Figure 13 Time Advance Function of Coupled model

After all messages are transferred to their destinations and every model has changed its state, structural change to add, delete, or move models can be done by a dynamic_structure function. Next, the *coupled* model components update their *tL* (time of last event), *tN* (time of next event) values. The *coupled* model iterates the simulation cycle until the termination condition is met. Figure 13 illustrates how the time advance function works in coupled models.

5 Basic Models

5.1 Devs Model

A *devs* class has basic methods for handling time information (last time, next time), output message, and port information such as input ports and output ports.

```
timetype tL, tN;
message *output;
set *inports, *outports;
devs *parent;
```

Since this class provides many services needed in *devs* environment, numerous methods are defined to provide polymorphism for the children classes. The inject method is useful to send input to specific models. The make_content method creates the content for message.

```
inject(char *p, entity *val);
inject(char *p, entity *val, timetype e);
inject_address (char * p, addrclass * addr, entity * val, timetype e);
make_content(char *p, entity *val);
```

5.2 Atomicbase and Atomic Models

An *atomicbase* model has a simulation engine for atomic models, and also has the sigma variable. An atomic model which inherits from class *atomicbase* includes the phase variable. In million cell models, when we don't need a phase variable, we can use *atomicbase* model. The following methods handle sigma and phase variables.

```
hold_in(timetype sigma);
hold_in(phasetype phase, timetype sigma);
passivate();
passivate_in (phasetype phase);
```

5.3 Cellbase and Cell Models

Cellbase and *cell* classes have an address data structure as an attribute. They are used to represent large numbers of similar models called cells which communicate with each other in the same, uniform manner, such as in cellular spaces. Coupling information is applied all the cells in the same way. For example, the coupling "*out*" port to "*in*" port applies any time one cell sends to another. These models use methods handling the address to recognize source and destination.

```
addrclass my_location;
make_content_address(char *p, addrclass *addr, entity *val);
```

5.4 Coupled Models

Coupled models is the major class which embodies the hierarchical model composition constructs of the DEVS formalism. *Digraph* models, *block* models, and *digcell* models are specializations which enable specification of *coupled* models in specific ways. A *coupled* model is defined by specifying its component models, called its *components*, and the coupling relations which establish the desired communication links.

A *coupled* model tells how to couple several component models together to form a new model. It can itself be employed as a component in a larger *coupled* model, thus giving rise to hierarchical construction. A *coupled* model contains the following information:

- the set of components
- the set of input ports through which external events are received.
- the set of output ports through which external events are sent
- the external input, external output, and internal coupling

5.4.1 Instance Variables

Instance variables in *coupled* models are defined as follows and are initialized by the constructor:

```
set *components;
couprel *Coupling;   //couprel is derived from class relation
set *imminents;
function *comp_map;
```

- *Components*

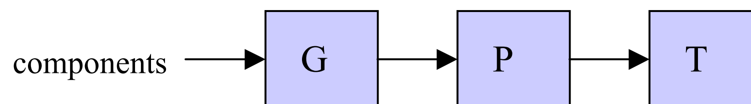


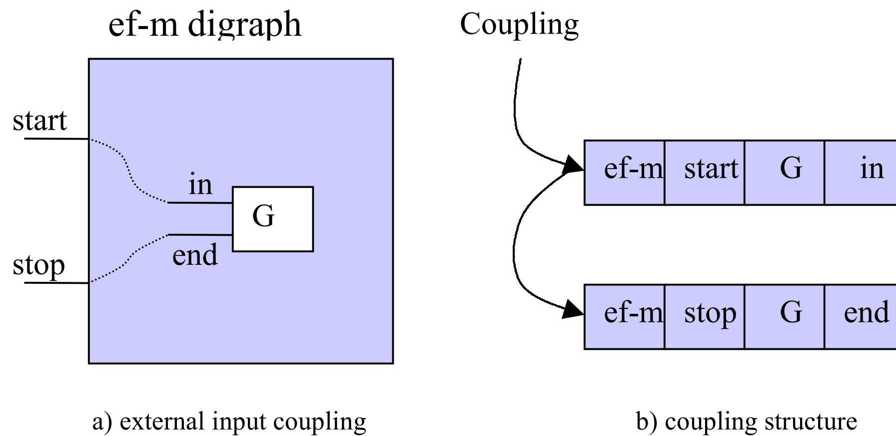
Figure 14 Components data structure

The *components* variable keeps component information. In Figure 14, the *coupled* model (*ef-m*) has 3 components: *genr*, *proc*, and *transd*. To add compoments we use "add_component";

```
ef-m->add_component(genr);
ef-m->add_component(proc);
ef-m->add_component(transd);
```

- Coupling

As the Coupling variable holds an instance of *relation* (derived class, couprel), each pair has a set of source and destination pairs to keep the coupling relationships among a *coupled* model and its components. Figure 15 represents the data structure for the Coupling variable. The source part, the first two columns, keeps the source (*ef-m*) followed by port ("*start"* or "*stop"*) of the source object. The destination part, remaining two columns, consists of the destination (*genr*) and port ("*in"* or "*end"*) of the destination object.



a) external input coupling b) coupling structure

Figure 15 coupling data structure

External input coupling couples the input ports of a *coupled* model to the input ports of its components. Likewise, external output coupling couples the output ports of the components to output ports of the *coupled* model. Internal coupling couples output ports of components to input ports of other components. We use "add_coupling" to provide coupling information;

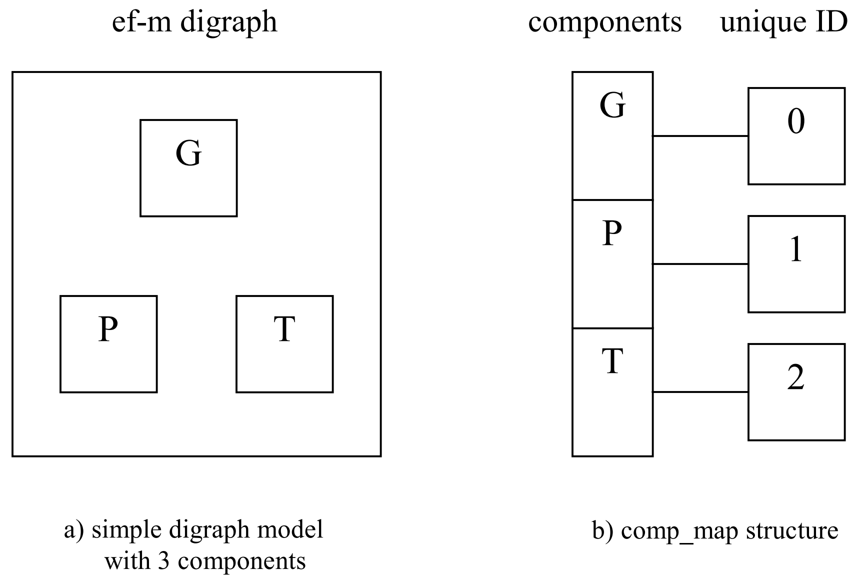
add_coupling (*ef-m, start, genr, in*);
add_coupling (*ef-m, stop, genr, end*);

- Comp_map

Comp_map, shown in Figure 16, is a function having (component, unique_id) pairs. Unique_id is assigned to each component in the *coupled* model so that its mailbox has unique integer id.

- Imminents

The imminents variable keeps the information of the components in which next output events should occur. To get the imminent information, the *coupled* model asks which of the components have the next global event time (Figure 13).



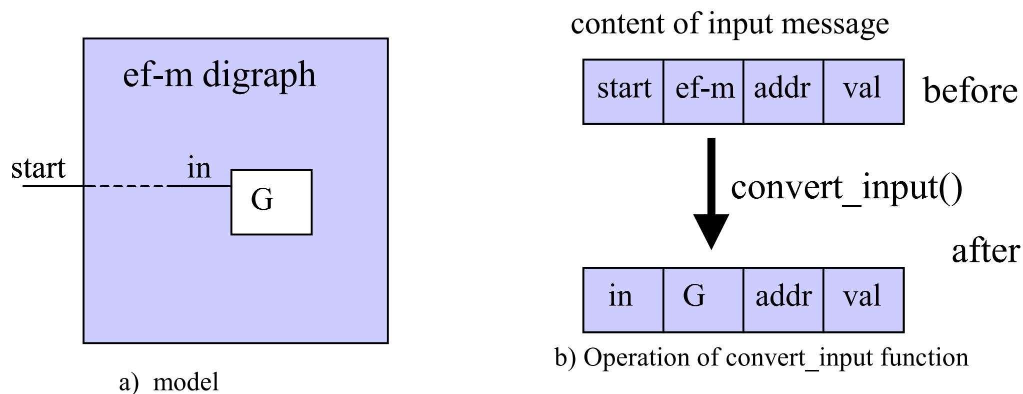
a) simple digraph model with 3 components

b) comp_map structure

Figure 16 comp_map structure

5.4.2 Convert_input()

When messages come into the system from outside, the destination of the messages must be changed every time they descend down the hierarchical instances in the model. Convert_input() uses the external input coupling (in which the external input of the *coupled* model goes to external input of the components). An input port of a *coupled* model can be connected to one or more input ports of the components. Using the information in the *coupling* relation, convert_input() translates the destination object and port name to those of each related component (Figure 17).



a) model

b) Operation of convert_input function

Figure 17 Convert Input Function of Coupled model

6 Derived Classes of coupled class

6.1 Digraph Models

Class *digraph* is a derived class of the class *coupled.* A digraph model is composed of a finite set of explicitly given components with explicitly specified coupling. Methods are used to specify the components of a *digraph* model and to specify the external and internal coupling relationships.

An example of a digraph model is an experimental frame model. When coupled to a model, it generates input external events, monitors model's behavior, and processes its output. An experimental frame module usually consists of a generator, transducer, and acceptor.

6.1.1 Instance Variables

The *Coupling* and components variables are inherited from the *coupled* class. During initialization, these variables are built from the digraph class definition. Method *get_components*() accesses the *components* variable.

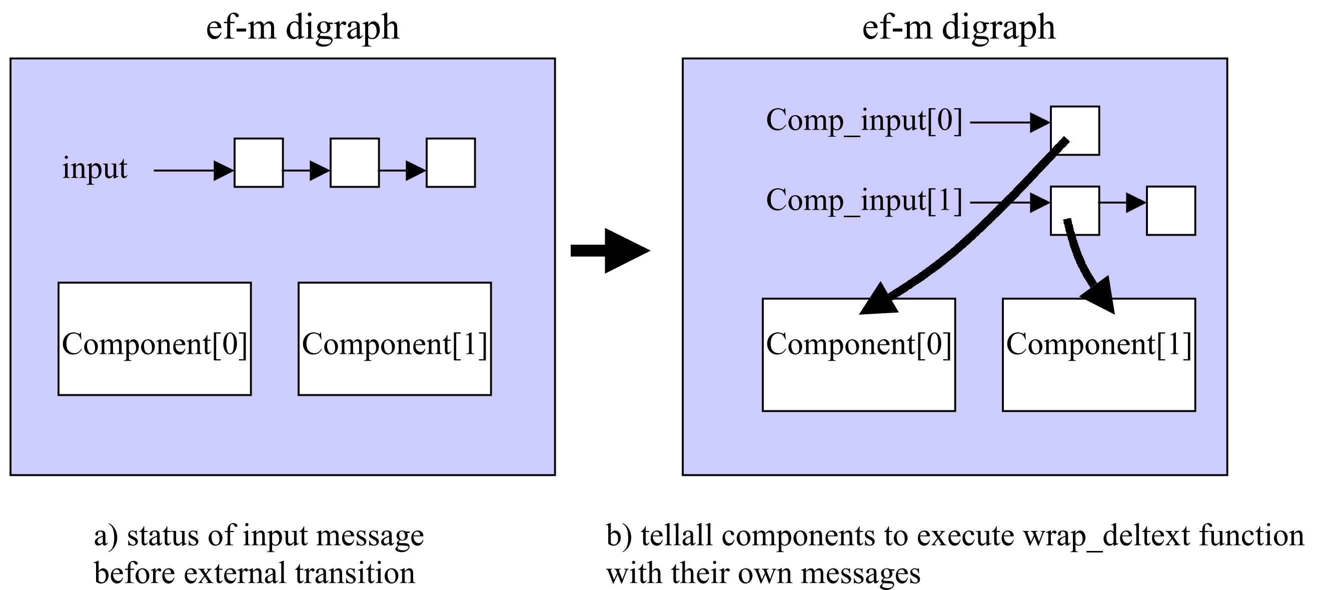


Figure 18 External Transition Function of Digraph Model

6.1.2 wrap_deltfunc Function

The wrap_deltfunc function sends input messages down to its components (children). In Figure 18, input messages of the *digraph* can come from outside of the system or from its components inside. These input messages are divided into mailboxes based on unique_ids of its components according to comp_map structure. Then the contents of each mailbox are unified as a message (container) and sent to the associated component. Finally, the wrap_deltfunc function of each component is called by the tellall method. This repeats recursively until the atomic level is reached.

6.1.3 Output (compute_input_output) Function

The output function should generate an external output just before an internal transition takes place. Figure 19 shows how a *digraph* model handles messages in the compute_input_output function. Since imminents are selected just before this method, the *digraph* model tells all its imminent components to execute their compute_input_output method. Then each imminent component produces its outputs and classifies them into two parts: input for internal messages and output for outgoing messages. The *digraph* model gathers all outputs from the imminent components and puts them into the input mailbox or output mailbox. Input messages for the components remain in the model, while output messages go up the hierarchical structure.

6.2 Block Models

Block models is specialization of *coupled* models which provides for coupling of a fixed or variable set of geometrically located *cells*, each of which is connected to other *cells* in a uniform way.

6.2.1 Instance Variables

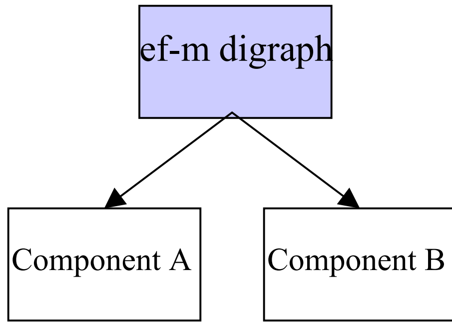
The *Coupling* and c*omponents* variables keep coupling information and components information respectively just as in digraph models. The *size* variable is an array which has boundary information for the region a block covers.

size[0] : lowest i address

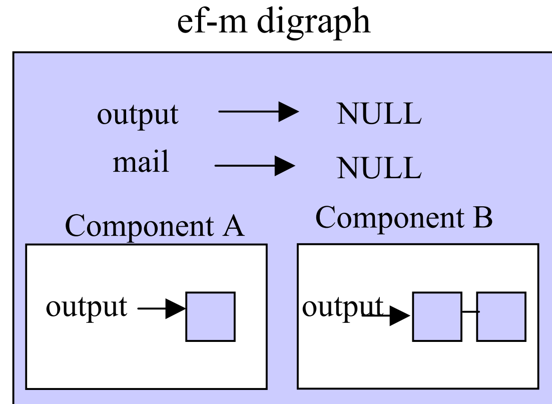
size[1] : lowest j address

size[2] : highest i address

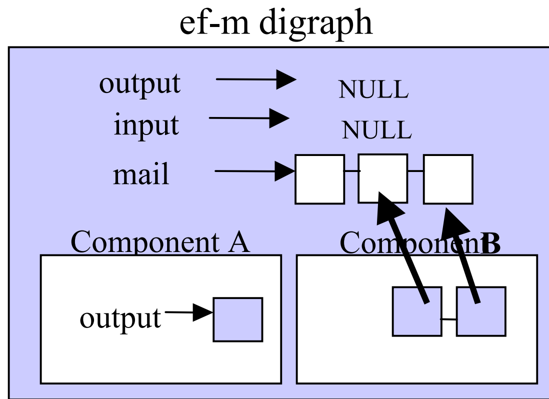
size[3] : highest j address



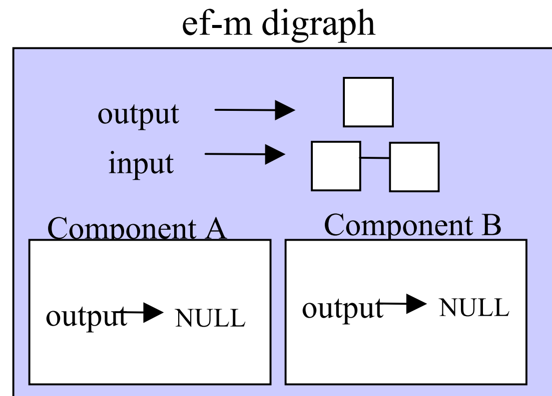
a) Tellall compute_input_output()



b) each component output



c) collect all output messages to a mail box



d) separate input and output messages from mail box

Figure 19 Compute Input Output Function

Whenever a model is added into the block, the block checks its boundary and updates the *size* variable if necessary. In reverse, whenever a model is deleted from, or moves outside of, the block, the block checks its boundary again and diminishes the appropriate *size* variable if it is in the boundary. The size information can be accessed by these methods--get_size0(), get_size1(), get_size2(), get_size3() respectively.

6.2.2 Coupling method of the Block

As a *block* model can have huge numbers of *cells* as its components, *cells* can be built in easy and uniform ways. The coupling methods in *block* models propagate down to its components so that every component in the block area has the same coupling information due to its uniformity. The add_coupling method (with two arguments) provides the coupling relationship between two ports of the *cells*. For example, add_coupling("*out*", "*in*") specifies connection from the "out" port of any *cell* to the "in" port of any other *cell* in the block:

```
add_coupling(char * p1, char * p2);
```

6.3 Digcell models

Digcell models combine both cell and digraph properties. A *digcell* model is almost the same as a *digraph* model except for having its own address and its way of converting output. As shown in Figure 20, *digcell* models work like cell models from *block* models point of view, and work like *digraph* models to their components. For example, the d*igcell* model, with address(1,n), has one *atomic* model (layer A) and one *digraph* model (*m-bc*). The digraph model, *m-bc,* has 2 *atomic* models (layer B, layer C).

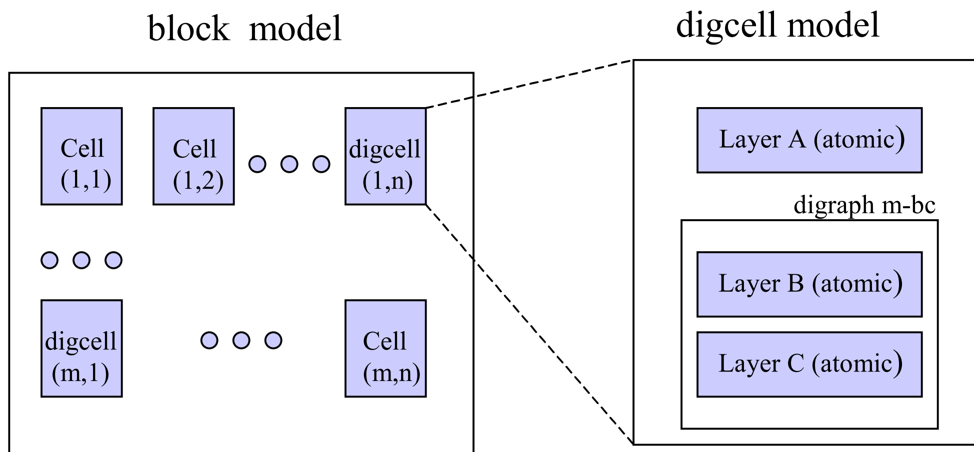


Figure 20 *Digraphcell* structure

6.4 Comparison of *digraph, block*, and *Digraphcell* models

Table 3 shows the differences among *digraph*, *block*, and *digcell* models. *Block* models use an address based messaging scheme. *Digraph* models use a port based scheme with coupling information. *Digcell* models use both the address-based messaging scheme and the port based scheme.

Table 3 Differences among digraph, block, and digraph-cell models

	Digraph	Block	Digcell
Messaging scheme	Port based	Address based	mixed (port + address)
Children information	Variable: "components"	Variable: "components"	Variable: "components"
Boundary Information	None	Variable "size"	None
Coupling information	Variable: "Coupling"	Variable: "Coupling"	Variable: "Coupling"
Coupling method	**One-to-one** add_coupling(d1,p1,d2,p2)	**One-to-many** add_coupling(p1,p2)	**One-to-one** add_coupling(d1,p1,d2,p2) **One-to-many** add_coupling(p1,p2)

In terms of Coupling relationship between influencees and receivers, both *digraph* and *digcell* models map one source to one destination, while *block* models support many-to-many coupling. As shown in Table 4, both *digraph* and *digcell* models can keep *atomic* models, *block* models, and *digraph* models as their components in hierarchical fashion; while c*ell*, *block* and *digcell* models, each having its own address (location), can become components of a *block* model.

Table 4 component relationship among digraph, block, and digcell models

component of / class	digraph model	block model	digcell model
Atomic	Yes	No	Yes
Cell	No	Yes	No
Digraph	Yes	Yes	Yes
Block	Yes	No	Yes
Digcell	No	Yes	No

7 Hierarchical Dynamic Structure

Dynamic structure is an important capability, related to hierarchical structure, for simulating complex systems. One form of structure change is an autonomous transformation in which a single component changes from one structural state to another. When the problem calls for creation, destruction, or modification of components during simulation runs, a much more complex situation occurs. In fact, two levels of the dynamic structure can be developed in which each level is a modular, hierarchical component in the DEVS environment. The lower level represents the current structure of a model, while the upper level is the decision-maker directing the structural change. The basic 2 level dynamic structure can be hierarchically expanded to a multi-level dynamic structure. The control of multi-level dynamic structure then can be transferred to the decision-maker of each level. Therefore, the multi-level dynamic structure can be easily built on the hierarchical structure.

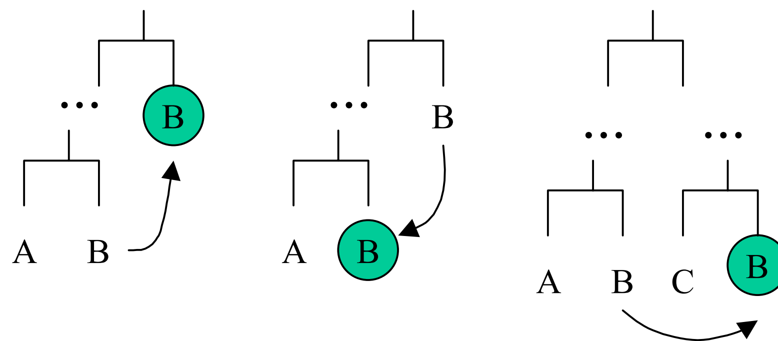


Figure 21 hierarchical dynamic movement

Figure 21 shows the three types of movement--move upward, move downward, and move sideward. On the other hand, a hierarchical structure--a tree structure for building dynamic structure--is a modular construction in which components are coupled together to form larger ones in order or one above another.

Figure 22 depicts a detailed example of a sidward dynamic structure. Figures 22-a and 22-c represent a block diagram and a decomposition tree before the moving event, while Figures 22-b and 22-d are after the moving event. A block class, blk, has two block components: blk1 and blk2. Blk1 has 2 cells which addresses are (1,1) and (1,2), while blk2 has also 2 cells (1,3) and (1,4) respectively. Whenever a model moves from a source block to a destination block, it is removed from the components list of the source block and is added to the components list of the destination block. In this case, if the address of the model is in the boundary of a block, each *size* variable of ancestor blocks of the source block is decreased by removing the component, while that of ancestor blocks of the destination block is increased by adding the components. The following codes show a simple example of move sideward case in Figure 22.

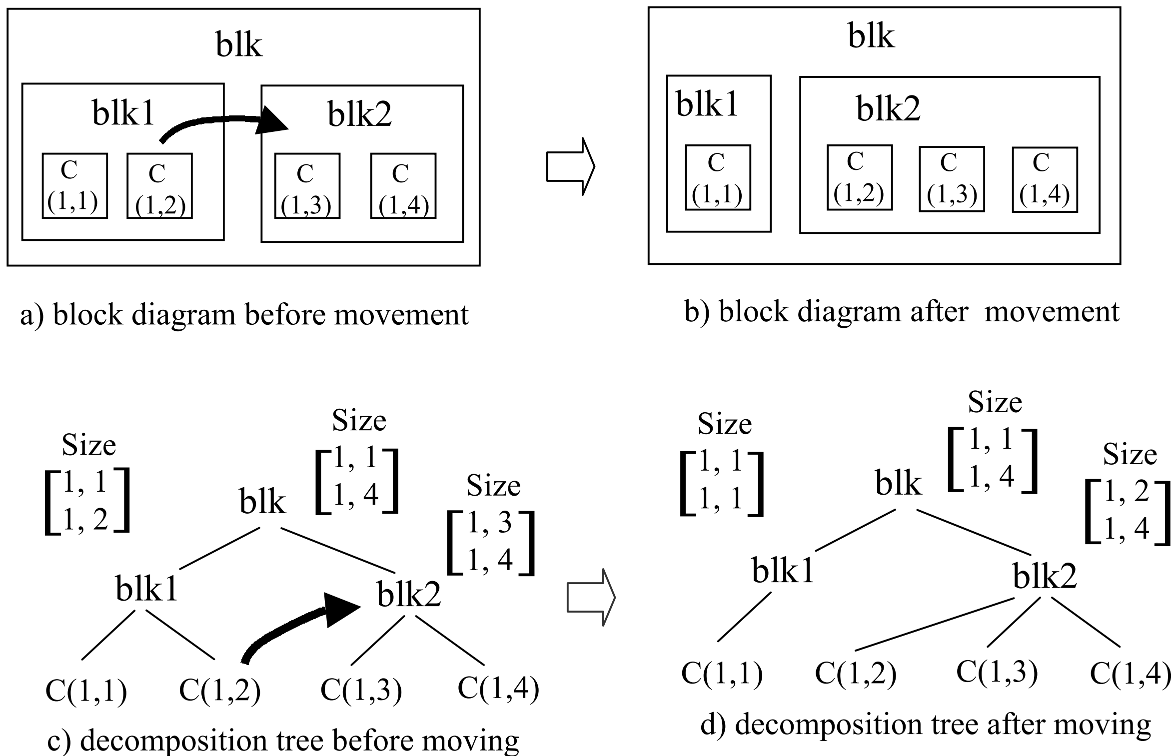


Figure 22 Dynamic structure : move sideward case

```
block *blk = new block(“blk”);
block *blk1 = new block(“blk1”);
block *blk2 = new block(“blk2”);
blk->add(blk1);
blk->add(blk2);

cell *c11 = new cell(“c11”, 1, 1);
cell *c12 = new cell(“c12”, 1, 2);
cell *c13 = new cell(“c13”, 1, 3);
cell *c14 = new cell(“c14”, 1, 4);

blk1->add(c11);
blk1->add(c12);
blk2->add(c13);
blk2->add(c14);

// in variable_structure function
element * e12 = blk1->remove(c12);
blk2->add(e12);
```

An example of variable structure is available at URL: www-ais.ece.arizona.edu/….

8 Test Operations

8.1 Control Flow of DEVS

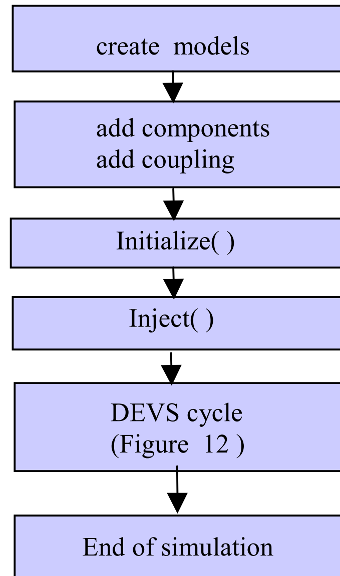


Figure 23 Control Flow of DEVS

Figure 23 represents control flow of DEVS-C++. At the beginning, the user creates appropriate models, inserting components into coupled models, and adding coupling information. After initializing the system, messages from outside of the system are inserted by injection. The DEVS cycle in figure 12 is then executed.

8.2 Initialize Operation

During initialization, timing variables are set. Last time variable (*tL*) is set to zero, and next time variable (*tN*) is set to infinity. After that, the hierarchical structure is constructed.

8.3 Inject Operation

To insert input to a model, the inject command inserts a message and calls *wrap_deltext* method with zero elapsed time. In *deltext*() of the *coupled* model, *convert_input*() and *tellall_wrap_deltext*() methods are executed. Descending, the structure hierarchy, eventually *wrap_dext*() method of the *atomic* model with the component's mailbox (*comp_input[i]*)

information calls the *deltext* method. After changing the state of the *atomic* model, the time advance function is executed.

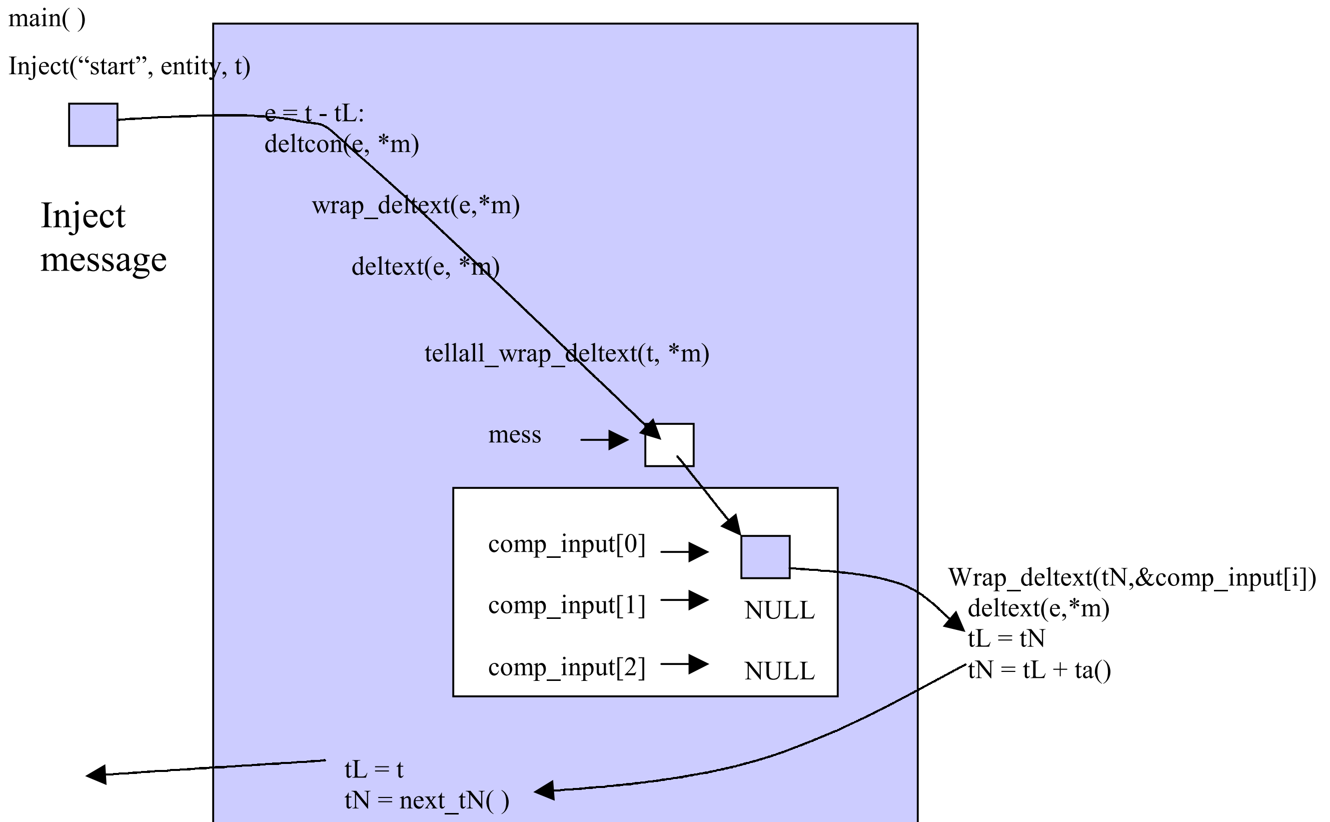


Figure 24 Inject Operation of Digraph Model

9 Installation of DEVS-C++ in UNIX

9.1 Structure of Directories

There are 4 directories in the source code of DEVS-C++ environment: *con, devs, lib*, and *examples*. The Con directory is for container library, while the *devs* for *devs* library. The archive files of two libraries are placed in the *lib* directory named libcon.a and libdevs.a respectively. The *Examples* directory has several example programs. Figure 26 shows the structure of directories of DEVS-C++.

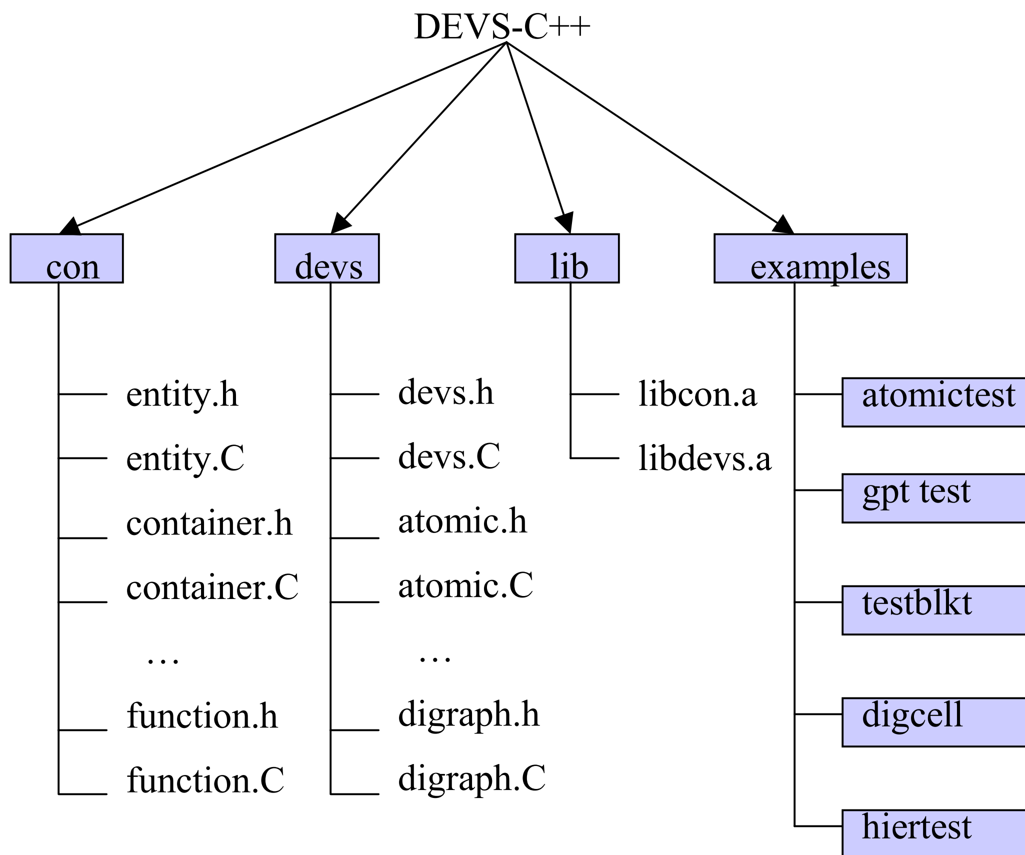


Figure 25 Directory structure of DEVS-C++

9.2 How to run test models

UNIX command "tar" creates an archive file from files or directories, and , in reverse, files from archive file. When we need to create an archive file named devsc++, tar with devsc++ directory, then type "*tar -cvf devsc++.tar devsc++*" "*tar -xvf devsc++.tar* ." command recreates the whole devsc++ directory.

The sequence to run test models is illustrated as follows:

% cd con

% make

% cd ../devs

```
% make
% cd ../examples/gpt
% make gpt
% gpt
```

References

[1] Bernard. P. Zeigler. *Theory of Modeling and Simulation*. Wiley-Interscience, New york, 1976.

[2] Bernard. P. Zeigler. *Multifacetted Modelling and Discrete Event Simulation*. Academic Press, London, 1984.

[3] Bernard. P. Zeigler. *Object-Oriented Simulation with Hierarchical, Modular Models.* Academic Press, San Diego, CA, 1990.

[4] Bernard. P. Zeigler and G. Zhang. "Mapping hierarchical discrete event models to multiprocessor systems: Concepts, algorithm and simulation." Parallel and Distributed Computing, 10:271-281, July, 1990.

[5] Alex Chunghen Chow and Bernard P. Zeigler. "Parallel DEVS : A parallel, hierarchical, modular modeling formalism." In Winter Simulation conference Proceedings, Orlando, Florida, 1994.

[6] Averill M. Law and W. David Kelton. *Simulation Modeling and Analysis.* McGraw-Hill Press, 1991.

Appendix

A1. Basic(atomic) model example

A2. GPT test example

A3. Hierarchical block model example

A4. Digraph cell(digcell) model example

A1. Source code for Basic (*atomic*) model example

```
//
//testatomic.C
//

#include "hold.h"

int main(int argc,char ** argv)
{

   cout << endl;
   cout << "TEST FOR ATOMIC: " << endl;
   cout << endl;
   hold* p  =  new hold("p",2);

    p ->initialize();

    entity * e = new entity("a");

    p ->inject("in",e);

    p->simulate(2);
    message * m = p->get_output();
    if (m->is_in_value(e) ) {
        cout << endl;
        cout << "TEST SATISFIED" << endl;
        cout << endl;
    }
    else {
        cout << endl;
        cout << "test NOT satisfied" << endl;
        cout << endl;
    }

   return 0;
}


//
//  hold.h
//
include "../devs/atomic.h"

class hold:public atomic{

protected:
   timetype hold_time;
   entity * store;

public:
   hold(char * name,timetype hold_time);
   void  deltext(timetype e,message * x) ;
   void  deltint( );
   message *  out( );
```

```
   void show_state();
   void show_output();
};


//
// hold.C
//
#include "hold.h"

hold::hold(char * name,timetype hold_time):atomic(name){
   inports->add("in");
   inports->add("start");
   outports->add("out");
   phases->add("busy");
   this->hold_time = hold_time ;
}

void  hold::deltext(timetype e,message * x)
{

   Continue();

   element *el, *elnext;
   for (el=x->get_head(); el!=NULL;el=elnext) {
      elnext = el->get_right();
      x->remove(el);
      content *con;
      con = (content *)el->get_ent();

      if(strcmp(con->p,"stop")==0) {
      }
      if(strcmp(con->p,"start")==0) {
         store = con->val;
         hold_in("busy",hold_time);
      }
   }
}

void  hold::deltint( )
{
   passivate();
}

message *  hold::out( )
{
   message * m = new message();
   content * con = make_content_address("out",store, new addrclass(1,1));
   store = NULL;
   m->add(con);
   m->show_message();

   return m;
}
```

A2. Source code for GPT Test Programs

```
//
//   gpt.C
//

include <stdio.h>

int main(int argc, char ** argv)
{
    genr * g = new genr("g",5);
    transd * t  = new transd("t",9);
    proc * p  = new proc("p",2);  //try also 5,6 and INFINITY

    digraph * efsim  = new digraph("efsim");

    efsim->add(g);
    efsim->add(t);
    efsim->add(p);

    g->add_coupling(g, "out", t, "ariv");
    g->add_coupling(g, "out", p, "in");

    t->add_coupling(t,"out",g,"stop");

    p->add_coupling(p, "out", t, "solved");

    efsim->get_inports()->add("start");
    efsim->get_outports()->add("stop");
    efsim->get_outports()->add("result");

    efsim->add_coupling(efsim, "start", g, "start");
    t->add_coupling(t, "out", efsim, "result");

    efsim->initialize();

    entity * ent = new entity("a");
    efsim->inject("start", ent);

    efsim->simulate(4);

    return 0;
}
```

```
//
// gpt.h
//


#ifndef _GENRH_
#define _GENRH_

#include <atomic.h>
#include <strio.h>

class genr:public atomic{

protected:

   timetype int_arr_time;
   int count;
   entity * ent;

public:

   genr(char * name,timetype int_arr_time) ;
   void  deltext(timetype e,message * x) ;
   void  deltint(timetype e);
   message *  out(timetype e);
};
#endif


/*
 *    genr.C
 */

#include "genr.h"

genr::genr(char * name,timetype int_arr_time):atomic(name){
   inports->add("stop");
   inports->add("start");
   outports->add("out");
   phases->add("busy");
   this->int_arr_time = int_arr_time ;
   count = 0;
}

void  genr::deltext(timetype e,message * x)
{
   Continue();

   int i;
   for (i=0; i< x->get_length();i++)
      if (message_on_port(x,"start",i))
      {
         ent = x->get_val_on_port("start",i);
         hold_in("busy",int_arr_time);
      }
```

```
   for (i=0; i< x->get_length();i++)
      if (message_on_port(x,"stop",i))
         passivate();
}

void  genr::deltint(timetype e)
{
   count = count +1;
   hold_in("busy",int_arr_time);
}

message *  genr::out(timetype e)
{

   message * m = new message();
   entity * p = new entity(name_gen(ent->get_name(),count));

   content * con = make_content("out", p);
   m->add(con);

  m->show_message();
  return m;
}


/*
 *  proc.h
 */


#ifndef _PROC_H
#define _PROC_H

#include <atomic.h>

class proc:public atomic{

protected:
     timetype processing_time;
     entity * job;

public:

      proc(char * name,timetype proc_time);
      void  deltext(timetype e,message * x) ;
      void  deltint(timetype e);
      message *  out(timetype e);
};
#endif
```

```
/*
 *  proc.C
 */


#include "proc.h"


proc::proc(char * name,timetype processing_time):atomic(name){
   inports->add("in");
   outports->add("out");
   phases->add("busy");
   this->processing_time = processing_time ;
}


void  proc::deltext(timetype e,message * x)
{
   Continue();

   int i;
   if (phase_is("passive"))
       for (i=0; i< x->get_length();i++)
          if (message_on_port(x,"in",i)) {
            job = x->get_val_on_port("in",i);
            hold_in("busy",processing_time);
          }
}

void  proc::deltint(timetype e)
{
   passivate();
}

message *  proc::out(timetype e)
{
   message * m = new message();
   entity *val = job;

   content * con = make_content("out",val);
   m->add(con);

   m->print_message();
   return m;
}
```

```
/*
 *    transd.h
 */


#ifndef _TRANSDH_
#define _TRANSDH_

#include <atomic.h>

#include <relation.h>
#include <num_ent.h>


class transd:public atomic{

protected:
   relation *arrived, *solved;
   timetype clock,total_ta,observation_time;

public:
   transd(char * name,timetype observ_time);
   void deltext(timetype e,message * x);
   void  deltint(timetype e);
   message *  out(timetype e);
};
#endif



/*
 *    transd.C
 */

#include "transd.h"


class number:public entity{
private:
   float NUMBER;
public:
   number(float i):NUMBER(i){}
   float get_number(){return NUMBER;}
   void print(){cout << "number: " << NUMBER;}
};

transd::transd(char * name,timetype observation_time):atomic(name){
   inports->add("ariv");
   inports->add("solved");
   outports->add("out");
   phases->add("active");
   arrived = new relation();
   solved = new relation();

   phase = "active";
   this->observation_time = observation_time;
   sigma = observation_time;
```

```
   clock = 0.0;
   total_ta = 0.0;
}

void transd::deltext(timetype e,message * x)
{
   clock = clock + e;
   Continue();
   entity * val;
   for (int i=0; i< x->get_length();i++)
   {
    if (message_on_port(x,"ariv",i))
       {
           val = x->get_val_on_port("ariv",i);
           arrived->add(val,new number(clock));
       }
    if (message_on_port(x,"solved",i))
      {
       val = x->get_val_on_port("solved",i);
       if (arrived->key_name_is_in(val->get_name()))
        {
            entity * ent = arrived->assoc(val->get_name());
            number * num = (number *)ent;
            timetype arrival_time = num->get_number();
            timetype turn_around_time = clock - arrival_time;
            total_ta = total_ta + turn_around_time;
            solved->add(val, new number(clock));
        }
       else {
             cerr << "arriving job " ; val->print();cout << " did not depart!";
             exit(1);
             }
     }
   }
}

void  transd::deltint(timetype e){
   clock = clock + sigma;
   passivate();
}

message *  transd::out(timetype e)
{
   float throughput;
   timetype avg_ta_time = 0;
   if (!solved->empty()) avg_ta_time = total_ta/solved->get_length();
   if (clock > 0)
   throughput = solved->get_length()/clock;
   else throughput = 0;

   char a[10], b[10];
   strcpy(a, arrived->get_head()->get_ent()->get_name());
   strcpy(b, solved->get_head()->get_ent()->get_name());

   if ( strcmp(a,b)==0 ) {
        cout << endl;
        cout << "***  TEST SATISFIED  ***" << endl;
```

```
          cout << endl;
     }
     else {
          cout << endl;
          cout << "test NOT satisfied" << endl;
          cout << endl;
     }

     cout << "jobs arrived :" ;
              arrived->print();

     cout << "jobs solved :" ;
              solved->print();

     cout << endl;
     cout << "AVERAGE TA = " << avg_ta_time <<endl;
     cout << "THROUGHPUT = " << throughput <<endl;
     cout << endl;

     message * m = new message();
     content * con = make_content("out",new float_ent(throughput));
     m->add(con);
     return m;
}
```

A3. Source code for Hierarchical Block Model Test Programs

```
/*
 *  testblk
 */


#include <stdio.h>
#include <stdlib.h>
#include <iostream.h>

#include <strio.h>
#include <num_ent.h>
#include <block.h>
#include <digraph.h>
#include "genr.h"
#include "transd.h"
#include "scell.h"

//#define   DEBUGG

int main(int argc, char ** argv)
{
   int i, j;
   float amount;
   scell *nc;
   digraph *tmpdig;

   int maxiter, maxsimtime, pause;

   // *************************************************
   //  relationship between genr & transd
   // *************************************************

    genr * g;
    transd * t;
    g = new genr("g",5);   // int_arr_time
    t = new transd("t",20);   // int_arr_time


    block * top1, top2;
    top1 = new block("top1");
    top2 = new block("top2");

    // printf("number of row and coulmn(rowmin,rowmax,colmin,colmax: ");
    // scanf("%d %d %d %d",&rowmin,&rownum,&colmin,&colnum);
    int rowmin, rownum, colmin, colnum;
    rowmin = 1;
    rownum = 4;
    colmin = 1;
    colnum = 4;

    for (i=rownum;i>=rowmin;i--) {              // left half cells
      //cout << "adding " << i << endl;
      for (j=(colnum+colmin-1)/2;j>=colmin;j--) {
```

```
            top1->add(new scell(name_gen("c",i,j),new addrclass(i,j)));
      }
    }

    for (i=rownum;i>=rowmin;i--) {              // right half cells
      //cout << "adding " << i << endl;
      for (j=colnum;j>=(1+colnum+colmin)/2;j--) {
            top2->add(new scell(name_gen("c",i,j),new addrclass(i,j)));
      }
    }


    // **************************************************
    //  relationship between ef and genr & transd
    // **************************************************

    digraph *efsim, *ef;
    efsim  = new digraph("efsim");
    ef  = new digraph("ef");


    ef->add(g);
    ef->add(t);
    efsim->add(ef);
    efsim->add(top1);
    efsim->add(top2);

    top1->add_coupling("out", "in");
    top1->add_coupling("in", "in");
    top2->add_coupling("out", "in");
    top2->add_coupling("in", "in");

    efsim->get_inports()->add("start");

    ef->get_inports()->add("start");

    g->get_inports()->add("stop");

    efsim->add_coupling(efsim, "start", ef, "start");
    ef->add_coupling(ef, "start", g, "start");

    g->add_coupling(g, "out", t, "ariv");
    g->add_coupling(g, "out", ef, "out");
    t->add_coupling(t, "out", g, "stop");

    ef->add_coupling(ef, "out", top1, "in");
    top1->add_coupling(top1, "out", top2, "in");

    top2->add_coupling(top2, "out", ef, "in");
    ef->add_coupling(ef, "in", t, "solved");

    efsim->initialize();
    efsim->inject("start", new entity("a"));
    efsim->start_sim(argc, argv);

    return 0;
}
```

```
/*
 * scell.h
 */


#ifndef _SCELL_H_
#define _SCELL_H_

#include <cell.h>

class scell:public cell{

  protected:

    /*
     * state variables
     */

    int processing_time;
    entity * store;


  public:

    scell(char * name, addrclass * location);

    void deltint(timetype e);

    void deltext(timetype e,message * x);

    message * out(timetype e );

};

#endif


/*
 * scell.C
 */


#include <string.h>
#include <num_ent.h>
#include "scell.h"


extern int rownum,colnum, rowmin, colmin;
```

```
scell::scell(char * name,addrclass * location)
            :cell(name,location)
{
      phases->add("busy");
      this->processing_time = 1 ;
        passivate();
}



//
// External Transition Function
//
void scell::deltext(timetype e,message * x)
{
   #ifdef DEBUGG
     cout << "SCELL: [" << get_name() << "] deltext()" << endl ;
   #endif

   if(x->empty()) return;           // for timing sync

   Continue();

   element *el, *elnext;
   for (el=x->get_head(); el!=NULL;el=elnext) {
      elnext = el->get_right();
      x->remove(el);
      content *con;
      con = (content *)el->get_ent();

      if(strcmp(con->p,"stop")==0)
         passivate();
      if(strcmp(con->p,"in")==0) {
         store = con->val;
           hold_in("busy",processing_time);
      }
   }
}
```

```
/*
 * Internal transition Function
 */
void scell::deltint()
{

   #ifdef DEBUGG
     cout << "SCELL: [" << get_name() << "] deltint()" << endl ;
   #endif

   passivate();
}


message * scell::out()
{
   #ifdef DEBUGG
     cout << "SCELL: [" << get_name() << "] out()" << endl ;
   #endif

   message * m;
   addrclass * addr;
   content * con;

   m = new message();
   addr = new addrclass(my_location->i + 1, my_location->j);
   con = make_content_address("out",(entity *)store, addr);
   store = NULL;
   m->add(con);

   #ifdef DEBUGG
   #endif
   m->show_message();

   return m;
}
```

```
/*
 *    genr.h
 */

#ifndef _GENRH_
#define _GENRH_

#include <atomic.h>

class genr:public atomic{

protected:

   timetype int_arr_time;
   entity * ent;
   int num;

public:

   genr(char * name,timetype int_arr_time) ;

   void  deltext(timetype,message *) ;
   void  deltint(timetype e) ;

   message *  out(timetype e );

};

#endif
```

```
/*
 *    genr.C
 */

#include "genr.h"
#include <element.h>
#include <bag.h>
#include <string.h>
#include <strio.h>

genr::genr(char * name,timetype int_arr_time):atomic(name){
   this->int_arr_time = int_arr_time ;
   num = 0;
   phases->add("active");
   passivate();
}

void  genr::deltext(timetype e,message * x)
{
   Continue();

   element *el, *elnext;
```

```
   for (el=x->get_head(); el!=NULL;el=elnext) {
      elnext = el->get_right();
      x->remove(el);
      content *con;
      con = (content *)el->get_ent();

      if(strcmp(con->p,"stop")==0)
         passivate();
      if(strcmp(con->p,"start")==0)
           hold_in("active",int_arr_time);
   }
}

void  genr::deltint(timetype e)
{
//cout << "genr deltint" << endl;
  hold_in("active",int_arr_time);
}

message *  genr::out(timetype e )
{

   message * m = new message();
   char * s;
   content *con;

   num++;
   con = make_content_address("out", new entity(name_gen("job",num)),
               new addrclass(1,1));
   m->add(con);
   //m->print();
   return m;
}

/*
** transd.h
*/

#ifndef _TRANSDH_
#define _TRANSDH_

#include <atomic.h>

#include <relation.h>


class transd:public atomic{

protected:

   relation *arrived, *solved;

   timetype clock,total_ta,observation_time;

public:
```

```
   transd(char * name,timetype observ_time);

   void deltcon(timetype,message *);
   void deltext(timetype,message *);
   void deltint(timetype e);

   message *  out(timetype e );

};

#endif


/*
 *    transd.C
 */

#include "transd.h"

//#define DEBUGG

class number:public entity{

  private:
      float NUMBER;
  public:
      number(float i):NUMBER(i){}
      float get_number(){return NUMBER;}
      void print(){cout << "number: " << NUMBER;}
};


transd::transd(char * name,timetype observation_time):atomic(name){

   inports->add("ariv");
   inports->add("solved");
   outports->add("stop");

   phases->add("active");
   arrived = new relation();
   solved = new relation();
   this->observation_time = observation_time;

   clock = 0.0;
   total_ta = 0.0;

   hold_in("active",observation_time);
}



void transd::deltcon(timetype e,message * x)
{
   deltint(e);
   deltext(e, x);
}
```

```
void transd::deltext(timetype e,message * x)
{
   #ifdef DEBUGG
   cout << "transd "; x->print();
   #endif
   clock = clock + e;

   element *el, *elnext;
   content *con;
   for (el = x->get_head();el != NULL;el = elnext) {
      elnext = el->get_right();
      con = (content *) el->get_ent();
      if (strcmp(con->p,"ariv") == 0) {
         arrived->add(con->val,new number(clock));
      }
      else if (strcmp(con->p,"solved") == 0) {
        if (arrived->key_name_is_in(con->val->get_name())) {
           entity * ent;
           number * num;
           timetype arrival_time, turn_around_time;

           ent = arrived->assoc(con->val->get_name());
           num = (number *) ent;
           arrival_time = num->get_number();
           turn_around_time = clock - arrival_time;
           total_ta = total_ta + turn_around_time;
           solved->add(con->val, new number(clock));
        }
        else {
           cerr << "arriving job " ; con->val->print();cout << " did not
arrived!";
           exit(1);
        }
      }
      else {
        cout << "port error" << endl;
        exit(1);
      }
      delete (addrclass *)con->address;
      delete con;
      //delete x;
   }

   Continue();
}

void transd::deltint(timetype e)
{
  passivate();
}



message *  transd::out(timetype e )
{
   timetype avg_ta_time;
```

```
   avg_ta_time = 0;

   if (!solved->empty())
       avg_ta_time = total_ta/solved->get_length();

   float throughput;
   if (clock > 0)
      throughput = solved->get_length()/clock;
   else
      throughput = 0;

   char a[10], b[10];
   strcpy(a, arrived->get_head()->get_ent()->get_name());
   strcpy(b, solved->get_head()->get_ent()->get_name());

   if ( strcmp(a,b)==0 ) {
        cout << endl;
        cout << "***  TEST SATISFIED  ***" << endl;
        cout << endl;
    }
    else {
        cout << endl;
        cout << "test NOT satisfied" << endl;
        cout << endl;
    }

   cout << "jobs arrived :" ;
            arrived->print();

   cout << "jobs solved :" ;
            solved->print();

   cout << endl;
   cout << "AVG TA = " << avg_ta_time <<endl;
   cout << "THROUGHPUT = " << throughput <<endl;
   cout << endl;

   message * m;
   content * con;

   m = new message();
   con = make_content("out",new entity("finished"));
   m->add(con);

   return m;

}
```

A4. Source code for digraph cell (digcell) model

```
//
// dctest.C
//


#include <stdio.h>
#include <stdlib.h>
#include <iostream.h>

#include <strio.h>
#include <num_ent.h>
#include <block.h>
#include <digraph.h>
#include <digcell.h>
#include "genr.h"
#include "transd.h"
#include "pcell.h"
#include "layer.h"



int main(int argc, char ** argv)
{
    int i, j;
    float amount;
    pcell *nc;
    digcell *tmpdig;

// **************************************************
//  relationship between genr & transd
// **************************************************

    genr * g;
    g = new genr("g",1);   // int_arr_time

    transd * t;
    t  = new transd("t",10);

    block * top;
    top = new block("top");

    digraph * efsim  = new digraph("efsim");

    efsim->add(g);
    efsim->add(t);
    efsim->add(top);

    int rowmin, rownum, colmin, colnum;
    rowmin = 1;
    rownum = 1;
    colmin = 1;
    colnum = 2;
```

```
    for (i=rownum;i>=rowmin;i--) {
      for (j=colnum;j>=colmin;j--) {
        if ((i+j)%2 ==0) {
            top->add(nc = new pcell(name_gen("c",i,j,1),new addrclass(i,j,1),
1));
      }
      else {
          top->add(tmpdig=new digcell(name_gen("dc",i,j,1), new
addrclass(i,j,1)));
            tmpdig->get_inports()->add("in");
            tmpdig->get_inports()->add("in");
            tmpdig->get_outports()->add("out");
            tmpdig->get_outports()->add("out");

            int k=1;
            layer *tl_1, *tl_2;

            tmpdig->add(tl_1 = new layer(name_gen("layer", k),1));      // 1:
proc time
            k++;
            tmpdig->add(tl_2 = new layer(name_gen("layer", k),1));      // 1:
proc time

            tmpdig->add_coupling(tmpdig,"in",tl_1,"in");
            tl_1->add_coupling(tl_1,"out",tl_2,"in");
            tl_2->add_coupling(tl_2,"out",tmpdig,"out");
      }
      }
    }

    top->add_coupling("out", "in");

    g->add_coupling(g, "out", t, "ariv");
    g->add_coupling(g, "out", top, "in");

    t->add_coupling(t,"out",g,"stop");

    top->add_coupling(top, "out", t, "solved");

    efsim->get_inports()->add("start");
    efsim->get_outports()->add("stop");
    efsim->get_outports()->add("result");

    efsim->add_coupling(efsim, "start", g, "start");
    t->add_coupling(t, "out", top, "stop");

    efsim->initialize();

    amount = 1000;
    efsim->inject("start", new float_ent(amount));

    efsim->simulate(10);


    return 0;
}
```

```
//
//  genr.h
//

#ifndef _GENRH_
#define _GENRH_

#include <atomic.h>


class genr:public atomic{

protected:

   timetype processing_time;
   float amount;
   int count;

public:

   genr(char * name, timetype int_arr_time) ;

   void  deltext(timetype e,message * x) ;

   void  deltint();

   message *  out();

   void show_state();

   void show_output();
};

#endif



//

//genr.C
//

#include "genr.h"
#include <num_ent.h>

//#define DEBUGG

genr::genr(char * name,timetype int_arr_time):atomic(name){
   inports->add("stop");
   inports->add("start");
   outports->add("out");
   phases->add("busy");
   this->processing_time = int_arr_time ;
   count = 0;
   hold_in("busy", processing_time);
}
```

```
void  genr::deltext(timetype e,message * x)
{
#ifdef DEBUGG
  cout << "GENR DELTEXT" << endl;
#endif

   if(x->empty()) return;           // for timing sync

   Continue();

   element *el, *elnext;
   for (el=x->get_head(); el!=NULL;el=elnext) {
      elnext = el->get_right();
      x->remove(el);
      content *con;
      con = (content *)el->get_ent();

      if(strcmp(con->p,"stop")==0)
           passivate();
      if(strcmp(con->p,"start")==0) {
           amount = ((float_ent *)con->val)->getv();
           hold_in("busy",processing_time);
      }
      delete con;
      delete el;
   }
}


void  genr::deltint()
{

#ifdef DEBUGG
   cout << "[genr] DELTINT" << endl;
#endif
   count++;
   //hold_in("busy", processing_time);
   passivate();
}

message *  genr::out()
{

   message * m = new message();
   float_ent * amt = new float_ent(amount);

   content * con = make_content_address("out", (entity *) amt,
                  new addrclass(1,1,1));
   m->add(con);
   m->show_message();

   return m;
}
```

```
void genr::show_state()
{
   cout << "\nstate of  " << name << ": " ;
   cout << "phase, sigma,count : "
         <<phase << " " << sigma << " " << count <<endl;
}



//
//   layer.h
//

#ifndef _LAYER_H_
#define _LAYER_H_

#include <atomic.h>

class layer:public atomic{

  protected:

    entity *store;
    int processing_time;

  public:

    layer(char * name, int pt);

    void deltint();

    void deltext(timetype e, message * x);

    message * out();

};

#endif



//
//   layer.C
//

#include "layer.h"
#include <atomic.h>
#include <num_ent.h>
#include <digcell.h>

//#define DEBUGG


layer::layer(char * name, int pt) :atomic(name)
{
   inports->add("in");
   inports->add("stop");
```

```
   outports->add("out");
   this->processing_time = pt ;
   store = NULL;
}


/*
 * External Transition Function
 */
void layer::deltext(timetype e, message * x)
{

#ifdef DEBUGG
  cout << "LAYER: [" << get_name() << "] deltext()" << endl ;
#endif

   if(x->empty()) return;           // for timing sync

   Continue();

   element *el, *elnext;
   for (el=x->get_head(); el!=NULL;el=elnext) {
      elnext = el->get_right();
      x->remove(el);
      content *con;
      con = (content *)el->get_ent();

      if(strcmp(con->p,"stop")==0)
           passivate();
      if(strcmp(con->p,"in")==0) {
           store = con->val;
           hold_in("busy",processing_time);
      }

      delete con;
      delete el;
   }
}


/*
 * Internal transition Function
 */
void layer::deltint()
{

#ifdef DEBUGG
  cout << "LAYER: [" << get_name() << "] deltint()" << endl ;
#endif

   passivate();
}


message * layer::out()
{
#ifdef DEBUGG
```

```
   cout << "LAYER: [" << get_name() << "] out()" << endl ;
#endif

   message * m = new message();

   addrclass * addr;
   content * con;

   int ii, jj, kk;
   ii = ((digcell *)get_parent())->get_my_location()->i;
   jj = ((digcell *)get_parent())->get_my_location()->j;
   kk = ((digcell *)get_parent())->get_my_location()->k;

   addr = new addrclass(ii, jj+1, kk);
   con = make_content_address("out",(entity *)store, addr);
   m->add(con);

   store = NULL;

   m->show_message();

   return m;
}


//
//   pcell.h
//

#ifndef _PCELL_H_
#define _PCELL_H_

#include <cell.h>

class pcell:public cell{

  protected:

    /*
     * state variables
     */

    entity *store;
    int processing_time;

  public:

    pcell(char * name, addrclass * location, int pt);

    void deltint();

    void deltext(timetype e,message * x);

    message * out();

};
```

```
#endif


//
//   pcell.C
//

#include "pcell.h"
#include <atomic.h>
#include <num_ent.h>

extern int rownum,colnum, rowmin, colmin;

//#define DEBUGG

/*
 * class pcell
 */

pcell::pcell(char * name, addrclass * location, int pt) :cell(name,location)
{
   inports->add("in");
   inports->add("stop");
   outports->add("out");
   this->processing_time = pt ;
   store = NULL;
}


/*
 * External Transition Function
 */
void pcell::deltext(timetype e,message * x)
{

#ifdef DEBUGG
  cout << "PCELL: [" << get_name() << "] deltext()" << endl ;
#endif

   if(x->empty()) return;           // for timing sync

   Continue();

   element *el, *elnext;
   for (el=x->get_head(); el!=NULL;el=elnext) {
      elnext = el->get_right();
      x->remove(el);
      content *con;
      con = (content *)el->get_ent();

      if(strcmp(con->p,"stop")==0)
           passivate();
      if(strcmp(con->p,"in")==0) {
           store = con->val;
           hold_in("busy",processing_time);
      }
```

```
   }
}


/*
 * Internal transition Function
 */
void pcell::deltint()
{

#ifdef DEBUGG
  cout << "PCELL: [" << get_name() << "] deltint()" << endl ;
#endif

   passivate();
}


message * pcell::out()
{
#ifdef DEBUGG
   cout << "PCELL: [" << get_name() << "] out()" << endl ;
#endif

   message * m = new message();

   addrclass * addr;
   content * con;
   addr = new addrclass(my_location->i, my_location->j+1, 1);
   con = make_content_address("out",(entity *)store, addr);
   m->add(con);

   store = NULL;
#ifdef DEBUGG
#endif
   m->show_message();

   return m;
}



//
//   trnasd.h
//

#ifndef _TRANSDH_
#define _TRANSDH_

#include <atomic.h>
#include <function.h>


class transd:public atomic{

protected:
```

```
   function *arrived, *solved;
   timetype clock,total_ta,observation_time;

public:

   transd(char * name,timetype observation_time);

   void deltext(timetype e,message * x);
   void  deltint();
   message *  out();
   void show_state();
   void show_output();
};

#endif



//
//   transd.C
//

#include "transd.h"
#include <mess.h>
#include <num_ent.h>

//#define DEBUGG

transd::transd(char * name,timetype observation_time):atomic(name){
   inports->add("null");
   inports->add("ariv");
   inports->add("solved");
   outports->add("out");
   phases->add("active");
   arrived = new function();
   solved = new function();

   phase = "active";
   this->observation_time = observation_time;
   hold_in("active", observation_time);
   clock = 0.0;
   total_ta = 0.0;
}



void transd::deltext(timetype e,message * x)
{
   timetype clock;
   clock = tL + e;

   #ifdef DEBUGG
   cout << "[transd] start DELTEXT" << endl;
   cout << "TRANSD [" << get_name() << "] deltext  clock=" << clock << endl;
   #endif
```

```
   if(x->empty()) return;               // for timing sync

   Continue();

   element *el, *elnext;
   for (el=x->get_head(); el!=NULL;el=elnext) {
      elnext = el->get_right();
      x->remove(el);
      content *con;
      con = (content *)el->get_ent();

      if(strcmp(con->p,"ariv")==0) {
           arrived->add(con->val,new float_ent(clock));
      }
      else if(strcmp(con->p,"solved")==0) {
          entity * ent = arrived->assoc(con->val);
          timetype arrival_time = ((float_ent *)ent)->getv();
          timetype turn_around_time = clock - arrival_time;
          total_ta = total_ta + turn_around_time;
          solved->add(con->val, new float_ent(clock));
       }
       delete con;
       delete el;
   }
}

void  transd::deltint(){
   #ifdef DEBUGG
   cout << "[transd] start DELTINT" << endl;
   #endif
   passivate();
}

message *  transd::out()
{
   #ifdef DEBUGG
   cout << "[transd] start OUT" << endl;
   #endif
   timetype avg_ta_time = 0;
   if (!solved->empty())
      avg_ta_time = total_ta/solved->get_length();
   float throughput;
   throughput = solved->get_length()/tN;

   char a[10], b[10];
   strcpy(a, arrived->get_head()->get_ent()->get_name());
   strcpy(b, solved->get_head()->get_ent()->get_name());
   if ( strcmp(a,b)==0 ) {
        cout << endl;
        cout << "***  TEST SATISFIED  ***" << endl;
        cout << endl;
    }
    else {
        cout << endl;
        cout << "test NOT satisfied" << endl;
        cout << endl;
    }
```

```
   cout << "jobs arrived :" ;
            arrived->print();

   cout << "jobs solved :" ;
            solved->print();

   cout << endl;
   cout << "AVG TA = " << avg_ta_time <<endl;
   cout << "THROUGHPUT = " << throughput <<endl;
   cout << endl;

   message * m = new message();

   content * con = make_content("out",new entity("finished"));
   m->add(con);

   m->show_message();

   return m;
}

void transd::show_state()
{
   cout << "state of  " << name << ": " ;
   cout << "phase, sigma,arrived,solved: " <<phase << " " << sigma << " ";
}
```