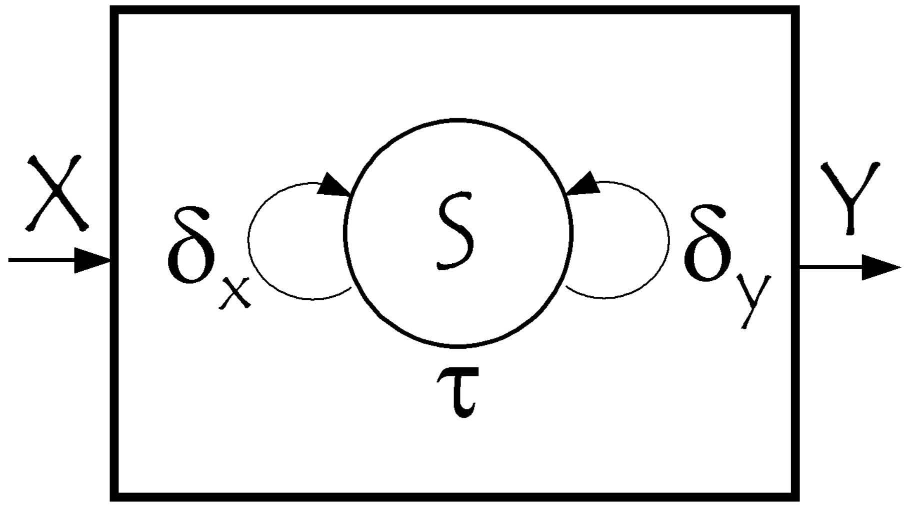

DEVS++
Open Source

Moon Ho Hwang

DEVS++ verion 1.4.2

You can cite this manual in the form of BibTex as follows.

```
@MANUAL{DEVSpp,
  TITLE =        "{DEVS++: C++ Open Source Library of DEVS Formalism}",
  author =       "Moon Ho Hwang",
  address =      "http://odevspp.sourceforge.net/",
  edition =      "v.1.4.2",
  month =        "April",
  year =         "2009",
}
```

Preface

Perfection is achieved, not when there is nothing more to add, but when there is nothing left to take away.

- Antoine de Saint Exupery

DEVS++ is an open source library that is an implementation of discrete event system specification (DEVS) formalism in C++ language. More than 30 years ago, Dr. Zeigler introduced DEVS to the public through his first book [Zei76], and its second edition [ZPK00] became available in 2000 due to the help of other two authors, Dr. Praehofer and Dr. Kim.

In 1994 when I was a Ph.D. student at the Korea Advanced Institute of Science and Engineering (KAIST), I was taught the DEVS theory by Dr. Kim who had been taught it by Dr. Zeigler. At that time, Dr. Kim used a C++ library, called DEVSim++© [Kim94] in one of his courses. I became fascinated with it even at the first glance because I had been struggling with developing a simulator without any theory for a while. DEVSim++ was so neat and well-organized as is DEVS inherently.

After seeing the header files of DEVSim++, I developed several versions of DEVS-based C++ kernels. One of them has been used in the VMS Lab., directed by Dr. Byoung Kyu Choi, IE Dept. at KAIST, and some of them are used in commercial packages of Cubiteck Ltd. Co., Seoul, Korea.

I had a chance to meet Dr. Ziegler in the DEVS standardization session of the 2005 DEVS Symposium. At that time, Dr. Zeigler suggested that I open my C++ DEVS library (called DEVS++), and I accepted his suggestion. I released the implementation as an open source project at http://odevspp.sourceforge.net in 2005. However, I were not able to finish writing its user manual over a couple years. Finally, the first version of the DEVS++ manual was released in May, 2007 when DEVS++ has evolved up to version 1.4.1.

The main objective of this document is to introduce the DEVS++ library. Since it is a C++ implementation of DEVS formalism, we need to understand what DEVS is first. Chapter 1 provides a belief review of DEVS formalism by introducing DEVS structures and their behaviors. Chapter 1 also gives sample codes for a ping-pong game using DEVS++ so we can see what the DEVS++ codes look like.

Chapter 2 explains the DEVS++ Library in terms of the object oriented programming paradigm of C++. We will see the class hierarchy and some of the virtual functions the user is supposed to override to make a concrete class. In addition, this section introduces a menu that DEVS++ provides when we run DEVS++ from a console.

Chapter 3 demonstrates several simple examples from atomic DEVS models to a coupled DEVS network. In these examples, we can check the knowledge learned from the previous chapters.

Chapter 4 deals with one of major goals of simulation study, that is, how to measure some performance indices. To do this, the mathematical definitions of throughput, cycle time, utilization and average queue length are addressed first, then their implementations in DEVS++ are introduced using practical examples. I hope the readers will have insight to modify these simple examples for their own purposes.

In addition to these main chapters, there are two appendixes. Appendix A explains how to build DEVS++ library from its source codes. Appendix B summaries the revision history and development plans of DEVS++.

Acknowledgements

I would thank Dr. Tag Gon Kim and Dr. Bernard. P. Zeigler for introducing me the world of DEVS.

Many thanks to

- Dr. Russ Mayers who read the entire document for version 1.4.1, corrected some of my not so excellent English expressions, and suggested some interesting systems engineering ideas. Without his devoted help, this manual could never have been completed.

- Shi Zhaoxiang who pointed out computation error of Example 4.5 so I were able to fix it when releasing the document of verion 1.4.2.

Special thanks are also due to my wife, Su Kyeon Cho, my mom Kyoung-Ai Kim, and my dad, Seung Hun Hwang who passed away in 2005 when the first version of DEVS++ was born.

Troy, MI
May 3, 2009

Moon Ho Hwang

Contents

List of Figures

List of Tables

Chapter 1

DEVS Formalism and DEVS++ code

In DEVS formalism the *time base* denoted by $\mathbb{T}$, is the *non-negative real numbers*, i.e. $\mathbb{T} = [0, \infty)$. Even though time can not reach the transfinite number, infinity (∞), sometimes it is useful to include ∞ in our consideration so we use the extend set, denoted by $\mathbb{T}^\infty = [0, \infty]$.

This chapter introduces DEVS formalism in terms of the *atomic DEVS* to define the dynamic behavior, and the *coupled DEVS* to build the hierarchical network structure.

1.1 Atomic DEVS

An atomic DEVS model is defined by a 7-tuple structure

$$A =< X, Y, S, s_0, \tau, \delta_x, \delta_y >$$

where

- X is a set of *input events*.
- Y is a set of *output events*.
- S is a set of *partial states*.
- $s_0 \in S$ is *the initial partial state*.
- $\tau : S \rightarrow \mathbb{T}^\infty$ is the *time advance function*. This function is used to determine the lifespan of a state.

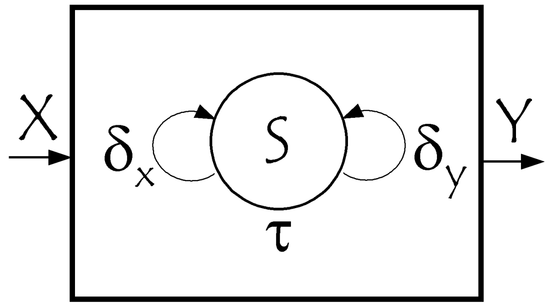

Figure 1.1: Symmetric Structure of Atomic DEVS

- $\delta_x : Q \times X \to S \times \{0, 1\}$ is the *external transition function* where

$$Q = \{(s, t_s, t_e)|s \in S, t_s \in \mathbb{T}^{\infty}, t_e \in (\mathbb{T} \cap [0, t_s])\}$$

  is *the set of total states* where t_s and t_e are the *lifespan* of the state, s, and the *elapsed time* since last reset of t_e, respectively. $\delta_x(s, t_s, t_e) = (s', b)$ defines how an input event, x, changes the state, s, as well as the lifespan, t_s, and the elapsed time, t_e.

- $\delta_y : S \to Y \times S$ is the *output and internal transition function* that defines how a state generates an output event and, at the same time, how it changes the state internally. This function can be invoked when the elapsed time reaches the lifespan. [1]

■

Figure 1.1, also used as the cover illustration, shows the symmetric structure of DEVS in the sense that the input event set (X) and the external transition function (δ_x) are on the input side; the output event set (Y) and the output and internal transition function (δ_y) are on the output side; and a set of states (S) and its time advance function (τ) are in the middle.

Definition 1.1 (Deterministic and Nondeterministic Functions) *Let A and B be two arbitrary sets. Then function $f : A \to B$ is called* deterministic *if give an $a \in A$, the values of callings $f(a)$ at different times are identical. Otherwise, f is called* non-deterministic. □

Definition 1.2 (Deterministic and Nondeterministic DEVSs) *A DEVS model, M, is called* deterministic *if s_0, τ, δ_x, and δ_y are deterministic. Otherwise, M is called* non-deterministic. □

[1] In [ZPK00], δ_y is split into two functions: the output function $\lambda : S \to Y$ and the internal transition function $\delta_{int} : S \to S$.

Behavior of Atomic DEVS models

Suppose that $\mathcal{A} =< X, Y, S, s_0, \tau, \delta_x, \delta_y >$is an atomic DEVS model. Then behavior of $\mathcal{A}$ is a sequence of total state transitions

$$(s, t_s, t_e) \to (s', t'_s, t'_e)$$

where total states (s, t_s, t_e) and $(s', t'_s, t'_e) \in Q$ are respectively defined at time $t_l, t_u \in \mathbb{T}$ such that $t_l \leq t_u$ in the following three different cases.

1. **Change by Time Passage** If there is no event until time t_u,
   (a) $(s' = s) \wedge (t'_s = t_s)$, i.e, partial states and life spans are preserved, but
   (b) the new elapsed time increases such that $t'_e = t_e + t_u - t_l$.

   We call move q to q' *total state change by time passage*.
2. **Change by an external transition** When $\mathcal{A}$ receives an input event $x \in X$,
   (a) $(s', \tau(s'), 0)$ if $\delta_x(q, x) = (s', 1)$,
   (b) (s, t_s, t_e) if $\delta_x(q, x) = (s', 0)$.
3. **Change by an internal transition** If there is no input event when t_e reaches at t_s,[2] then new state is defined as $(s', \tau(s'), 0)$ if $\delta_y(s) = (y, s')$.

For a formal definition of atomic DEVS behaviors, readers can refer to [DEV08a].

Example 1.1 (Ping-Pong Player) Figure 1.2 shows an atomic DEVS model for a ping-pong player. This model has an input event "?receive" and an output event "!send". And it has two states: "Send" and "Wait". Once the player gets into "Send", it will generates "!send" and backs to "Wait" after the sending time which is a random variant in the uniform probability distribution function (pdf) of [0.1, 1.2]. When staying at "Wait" and if it gets "?receive", it changes into "Send" again.

Formally we can rewrite this player as $M_{Player} =< X, Y, S, s_0, \tau, \delta_x, \delta_y >$ where X={`?receive`}; Y={`!send`};S={`Send`, `Wait`}; s_0=`Send`; τ(`Send`)$\in$ [0.1, 1.2], τ(`Wait`)=∞; $\delta_x(s, t_s, t_e, x) = \delta_x$(`Send`,$\infty$,[0,$t_s$],`?receive`) =(`Send`,1), $\delta_x(s, t_s, t_e, x) = \delta_x$(`Send`,[0.1, 1.2],[0,$t_s$],`?receive`)=(`Send`,0); $\delta_y(s) = \delta_y$(`Send`)=(`!send`,`Wait`);

Notice that this player model is not deterministic because the lifespan value of `Send` decided by τ(`Send`) is uniformly distributed in the interval of [0.1, 1.2]. □

[2] Recall that $t_e \in \mathbb{T} = [0, \infty)$ and t_s can be ∞. Thus when $t_s = \infty$, it is impossible that $t_e = t_s$.

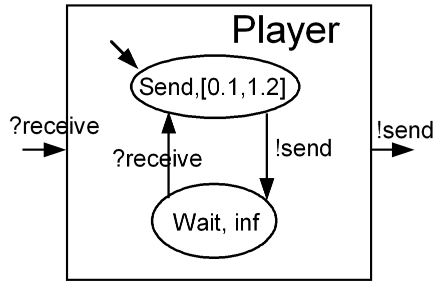

Figure 1.2: State Transition Diagram of Ping-Pong Player

1.2 Coupled DEVS

The coupled DEVS provides the hierarchical and modular structure necessary to describe system networks. Formally, a coupled DEVS is defined by

$$N =< X, Y, D, \{M_i\}, EIC, ITC, EOC >$$

where

- X is a set of *input events.*
- Y is a set of *output events.*
- D is a set of *names of sub-components*
- $\{M_i\}$ is a set of DEVS *models* where $i \in D$. M_i can be either an atomic DEVS model or a coupled DEVS model.
- $EIC \subseteq X \times \bigcup_{i \in D} X_i$ is a set of *external input couplings* where X_i is the set of input events of M_i.
- $ITC \subseteq \bigcup_{i \in D} Y_i \times \bigcup_{i \in D} X_i$ is a set of *internal couplings* where Y_i is the set of output events of M_i.
- $EOC \subseteq \bigcup_{i \in D} Y_i \times Y$ is a set of *external output couplings.* ■

Practically, we can see an event as a pair of (*port*, *value*) and the coupling as a pair of ($port_{source}$, $port_{destination}$) [Zei90, ZPK00]. The basic assumption of the *port coupling* is that the value of $port_{source}$ is casted to that of $port_{destination}$. We can find that the realistic example of the port coupling in the VHDL language [Ska96] and the language of programmable logic controller (PLC) [Lew98]. DEVS++ implements the (port,value) view for events (we will see it in Section 2.1.4). However, it does not mean that the event should be a pair of a port and a value.

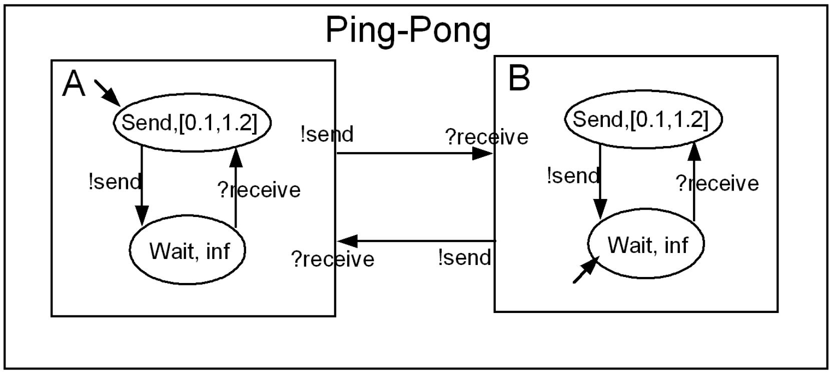

Figure 1.3: DEVS Model of Ping-Pong Game

Behavior of Coupled DEVS models

The coupled DEVS's behavior is described verbally as follows.

1. **Change by Time Passage** If there are no events in time duration t_l to $t_u(t_l \leq t_u)$, all sub-components' total states are changed by time passage of $dt = t_u - tl$.

2. **Change by an external transition** When N receives an input event, the coupled DEVS transmits the input event to the sub-components through the set of external input couplings.

3. **Change by an internal transition** When a sub-component produces its output event when the internal transition occurs, the coupled DEVS transmits the output event to the other sub-components through the set of internal couplings. The coupled DEVS also produces an output event of N through the set of external output couplings.

Theoretically speaking, DEVS is closed under the coupling which means that the behavior of any coupled DEVS model can be explained by an atomic DEVS model.[ZPK00]. For a formal definition of coupled DEVS behaviors, readers can refer to [DEV08b].

Example 1.2 (Ping-Pong Game) Consider a ping-pong game with two players that each represented by the `Player` model introduced in Example 1.1 except the initial state.

This block diagram can be modeled by a coupled DEVS such as $N_{PPGame} =< X, Y, D, \{M_i\}, EIC, ITC, EOC >$ where $X = \{\}; Y = \{\}; D$={`A`,`B`}; $\{M_i\}$ ={$\texttt{Player}_i$} where $\texttt{Player}_i$ is the atomic DEVS introduced in Example 1.1 with initial states `Send` for i=`A`, `Wait` for i=`B`, respectively; EIC={ }, ITC={ (`A.!send`, `B.?receive`), (`B.!send`, `A.?receive`)}, EOC = { }. □

1.3 Building Ping-Pong Game using DEVS++

This section shows *how DEVS++ codes look like* using the ping-pong game introduced in Example 1.2. All source codes below are available in `DEVSpp/Examples/Ex_PinPong` folder. If you want to build and run this example by yourself, Appendix A will be helpful for you.

```
#include "Atomic.h" //--- (1)
#include "Coupled.h"
#include "SRTEngine.h"
#include "RNG.h"
#include <iostream>
#include <math.h>

using namespace std;
using namespace DEVSpp; //--- (2)

const string WAIT = "Wait";
const string SEND = "Send";

//---- definition of atomic DEVS for Player --- (3)
class Player: public Atomic {
public:
    OutputPort* send;  //-- associated ports --- (4)
    InputPort* receive;
protected:  //-- associated internal state variables ----(5)
    string    m_phase;
    bool      m_width_ball;
public:
    Player(const string& name="", bool with_ball=false): Atomic(name),
        m_phase(WAIT), m_width_ball(with_ball)
    {
        send = AddOP("send");       //--- add ports --- (6)
        receive = AddIP("receive");
    }
    //---- four characteristic functions ------- (7)
    /*virtual*/ void init()
    {
        if(m_width_ball)
            m_phase = SEND;
        else
            m_phase = WAIT;
```

```
    }

    /*virtual*/ TimeSpan tau() const
    {
        static rv urv;

        if(m_phase == SEND)
            return urv.uniform(0.1, 1.2); //---- (8)
        else
            return DBL_MAX;
    }
    /*virtual*/ bool delta_x(const PortValue& x)
    {
        if(x.port == receive)
        {
            if(m_phase == WAIT) {
                m_phase = SEND;
                return true;
            }
        }
        return false;
    }
    /*virtual*/ void delta_y(PortValue& y)
    {
        if(m_phase == SEND) {
            y.Set(send);
            m_phase = WAIT;
        }
    }
    //------ end of four characteristic functions -------
    /*virtual*/ string Get_s() const //------(9)
    {
        return m_phase;
    }
};


Coupled* MakePingPongGame(const string& name) {
    Coupled* PingPong =  new Coupled(name);// ----(10)
    Player* A = new Player("A", true);  //--- (11)
    Player* B = new Player("B", false);
```

```
    A->CollectStatistics(true); //-- (12)
    B->CollectStatistics(true);

    PingPong->AddModel(A); //-- (13)
    PingPong->AddModel(B);

    //-- Internal Coupling --------  (14)
    PingPong->AddCP(A->send, B->receive);
    PingPong->AddCP(B->send, A->receive);

    PingPong->PrintCouplings(); //---- (15)
    return PingPong;
}

void main(void) {
    Coupled* PingPong = MakePingPongGame("PingPong");
    SRTEngine simEngine(*PingPong);//-- (16)
    simEngine.RunConsoleMenu(); //-- (17)
    delete PingPong;
}
```

Above example codes contain comments in the forms of "`//--- (#)`". Each "`(#)`" has the following explanation.

(1) Include Files

First of all, we should include the associated header files. In this example, we define the class `Player` derived from the class `Atomic` (`Atomic.h`); we create a ping-pong game as an instance of the class `Coupled` (`Coupled.h`); we will simulate the ping-pong game using a scalable simulation engine: `SRTEngine` (`SRTEngine.h`); and the time advance of the state `Send` is a random variable of the uniform pdf (`RNG.h`).

(2) Using Name Space

For convenience, we use the name space "`DEVSpp`" as well as "`std`". Without this, we should add a scope operator like `DEVSpp::` or `std::` in front of all classes and global APIs that are defined in `DEVSpp` and `std`.

(3) Player derived from Atomic

In this example, `Player` is a concrete class derived from `Atomic` which is an abstract class. We will see the class `Atomic` in Section 2.2.2.

(4) Interfacing Ports

The port pointers are useful to identify the added ports. Without these pointers, we would have to search for each pointer by its name, and that can be a burden. For more information of the class `Port`, the reader can refer to Section 2.1.2

(5) State Variables

The derived and concrete class of atomic DEVS will have its state variables to describe its dynamic situations. In DEVS++, we use member data of C++ for the state variables.

(6) Adding Interfacing Ports

The interfacing port pointers mentioned in (5) are assigned by calling either the `AddIP` or the `AddOP` function in which memory allocations and parent assignments are performed. A set of port related functions defined at `Atomic` can be referred to Section 2.2.2.

(7) Defining Four Characteristic Functions

The characteristic functions such as τ, δ_x, δ_y plus `init()` are pure virtual, and so we should override them when defining a concrete class of `Atomic`. These characteristic functions describe the behavior of the state transition diagram of Figure 1.3.

(8) Random Number

The lifespan of `Send` is a random variable with uniform pdf of [0.1,1.2], where elements of the domain denote time-units. To generate the random number, the random variable class `rv` is used as a static local variable for the output of the function `tau()`. The pdfs available in DEVS++ are addressed in Section 2.4.

(9) Displaying the current state

To show the current state, we will override the `Get_s()` function which is supposed to return the current state in a `string`.

(10) Making the Ping-Pong Game

We make an instance of coupled DEVS for the ping-pong game.

(11) Creating Two Players

The ping-pong game has two sub-components that are instances of `Player` having different initial states.

(12) Collecting Statistics

If we want to collect statistics about the two players, we turn the flag on by calling `CollectStatistics(true)`. Chapter 4 will introduce performance measures and how we can collect statistics in detail.

(13) Adding Sub-components

We add two players `A` and `B` by calling the function `AddModel` of the class `Coupled`.

(14) Adding Couplings

We add couplings between players `A` and `B` calling the function `AddCP` of the class `Coupled` .

(15) Print Couplings

Even though it is not necessary, we can call the function `PrintCouplings()` of `Coupled` to check the coupling status. The couplings of the ping-pong game are displayed as follows.

```
Inside of PingPong

 -- External Input Coupling (EIC) --
 ------ # of EICs: 0-----

 -- Internal Coupling (ITC) --
A.send --> B.receive
B.send --> A.receive
 ------ # of ITCs: 2-----

 -- External Output Coupling (EOC) --
 ------ # of EOCs: 0-----
```

(16) Making a simulation engine

Instancing a scalable simulation engine `SRTEngine` can be done by calling its constructor that needs the model supposed to be simulated. In this example the model is the coupled model of the ping-pong game.

(17) Running the console menu

We can use the console menu of `SRTEngine` by calling `RunConsoleMenu()`. After that, we will see the following screen on the selected console.

```
DEVS++: C++ Open Source of DEVS Implementation, (C) 2005~2009,
http://odevspp.sourceforge.net
The current date is 04/09/09
The current time is 11:42:26

scale, step, run, mrun, [p]ause, pause_at, [c]ontinue, reset,
rerun, [i]nject, dtmode, animode, print, cls, log, [e]xit
>
```

The first part shows the header of DEVS++ and current data and time. The second part shows the available command set. Even we don't have clear idea of each command, let's try" `run`" and then "`exit`".

The detailed information of each command will be provided in Section 2.3.

Chapter 2

Structure of DEVS++

DEVS++ is an C++ open source of DEVS formalism. Thus, there are two features: one comes from C++ language, the other from the formalism. Figure 2.1 shows the hierarchy relation among classes used in DEVS++.

As we reviewed in Chapter 1, two DEVS models called atomic DEVS and coupled DEVS have common features such as input and output event interfaces as well as time features such as current time, elapsed time, schedule time and so on. In DEVS++, these common features have been captured by a base class, called `Devs` from which the class `Atomic` (for atomic DEVS) and the class `Coupled` (for coupled DEVS) are derived.

In DEVS++, an event is a pair of (*port*, *value*) where *port* can be an instance of either `InputPort` class or `OutputPort` class, while *value* is an instance of a derived class of `Value` class such as `bValue` and `tmValue`. `SRTEngine` is a scalable real-time engine which runs a DEVS instance inside. `rv` is a class for a random variable.

In Figure 2.1, a *gray* box indicates a concrete class which can be created as an instance, while a *white* box is an abstract class which can not be created as

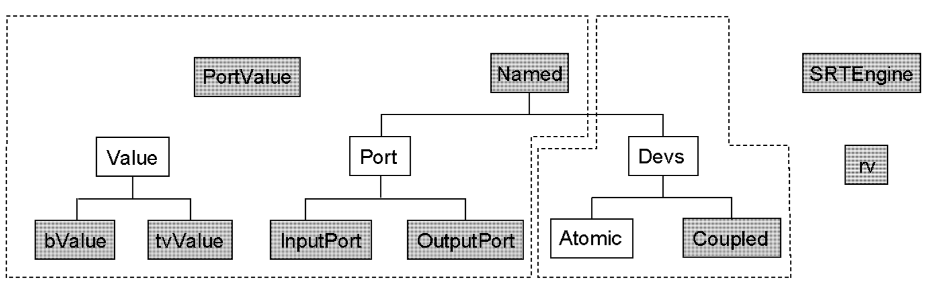

Figure 2.1: Classes in DEVS++

an instance.

We will first go through the PortValue class and its related classes in Section 2.1. Next, Devs class and its derived two classes: Atomic and Coupled will be investigated in Section 2.2, Section 2.3 will introduce a simulation engine class, called SRTEngine. And finally, We will see the random number generator rv in Section 2.4.

2.1 Event=PortValue

An event will be modeled by an instance of PortValue class which is a pair of Port and Value. We will first see the top-most base class, called "Named". Then we will look at Port-related classes and Value-related classes. And finally, the PortValue class will be seen in the last part of this section.

2.1.1 Named

Named is defined in a header file Named.h as a concrete class. The class provides its constructor whose argument is a string, and has a public Name field as a string.

```
class Named {
public:
    Named(const string& name):Name(name){}
    string Name;
};
```

2.1.2 Port, InputPort, and OutputPort

The Port.h file defines three classes Port, InputPort and OutputPort as follows.

```
class Port: public Named {
public:
    Devs* Parent;
    vector<Port*> ToP;   // Successor
    vector<Port*> FromP; // Predecessor
    ...
};

class InputPort: public Port {
    ...
};
```

```
class OutputPort: public Port {
    ...
};
```

Port is an abstract class derived from Named. It has a parent pointer whose type is Devs pointer and which is automatically assigned when we call the AddIP() and AddOP() functions of Devs. Port has "vector<Port*> ToP" as a set of successors as well as "vector<Port*> FromP" as a set of predecessors which are changed when we call AddCP() and RemoveCP() of Coupled.

InputPort and OutputPort are concrete and derived classes from Port.

2.1.3 Value, bValue and tmValue

In Value.h, there are three classes: Value, bValue and tmValue. Value is the base abstract class for the other two classes. Value has two *virtual* functions: Clone() makes a copy, STR() returns a string of a derived class's status.

```
class Value {
protected:
    Value(){}
public:
    virtual Value* Clone() const {return NULL;}
    virtual string STR() const {return string(); }
};
```

bValue is a concrete class derived from Value. bValue is a template class, and it has a field v whose type is the template augment V. Thus we can define bValue<bool>, bValue<char>, bValue<int>, bValue<double> whose value types are bool, char, int, and double, respectively.

```
template<class V>
class bValue: public Value {
public:
    V v;  //-- public value field
    ...
};
```

tmValue is a concrete class derived from Value. This class has a map from a string to a double-precision floating number that can be used to identify an *event as a string* and to specify its *occurrence time*.

```
class tmValue: public Value {
public:
    map<string, double> TimeMap;
```

```
    ...
};
```

You can see how to use this `tmValue` in Section 4.2.1 and 4.3.

2.1.4 PortValue

As mentioned before, an event in DEVS++ is modeled by a pair of associated classes: `Port` and `Value` by using the `PortValue` class.

```
class PortValue {
public:
    Port*  port; //-- either an output port or an input port
    Value* value;//-- typecast it to a concrete derived class!

    PortValue(const Port* p=NULL, Value* v=NULL);
    ...
 };
```

2.2 DEVS

As introduced in Chapter 1, DEVS has two basic structures: atomic DEVS and coupled DEVS. In DEVS++, these two structures are implemented as the classes `Atomic` and `Coupled`, respectively, and are derived from the base class `Devs`. Thus `Devs` has the common member data and functions of both `Atomic` and `Coupled`.

2.2.1 Base DEVS: Devs

`Devs` defined in `Devs.h` is an abstract class derived from `Named`.[1] And it has the parent pointer assigned by `AddModel()` of `Coupled` as we will see in Section 2.2.3.

```
class DEVSpp_EXP Devs: public Named {
public:
    Coupled*  Parent; // parent pointer
    ...
```

There are adding, getting, removing, and printing functions for the input ports denoted as `AddIP`, `GetIP`, `RemoveIP`, and `PrintAllIPs`. Similarly, `AddOP`, `GetOP`, `RemoveOP`, and `PrintAllOPs` are available functions for the output ports.

[1] The macro DEVSpp_EXP can be compiled several different ways according to the set of preprocessor. For compiling dynamic linking library, we should add DLL in the preprocessor definitions. For more information, the reader can refer to Chapter A.

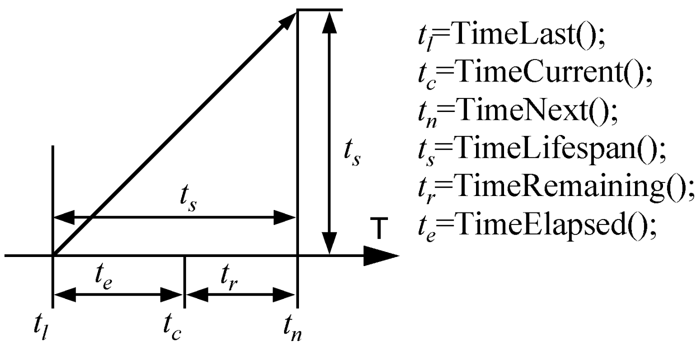

Figure 2.2: Relations of Times

```
//-- Input Port related functions --
InputPort* AddIP(const string& ipn);
InputPort* GetIP(const string& ipn) const;
InputPort* RemoveIP(const string& ipn);
void PrintAllIPs() const;

//-- Output Port related functions --
OutputPort* AddOP(const string& opn);
OutputPort* GetOP(const string& opn) const;
OutputPort* RemoveOP(const string& opn);
void PrintAllOPs() const;
```

Implementations of Times

Recall that the behavior of DEVS needs times notions, lifespan t_s and elapsed time t_e (see section 2.2.2). To capture these two times, DEVS++ implements two other time notions: *last event time* t_l and *next event time* t_n instead. If we have a *current time*, t_c, relationships among them are

$$t_s = t_n - t_l,$$

$$t_e = t_c - t_l$$

and

$$t_r = t_s - t_e = t_n - t_c$$

where t_r is *remaining time* to t_n. Figure 2.2 illustrates the relationships among these times.

The user doesn't have to set the values of times during simulation since that will be done by DEVS++. However, if users need to access the values of them, there are following APIs defined at `Devs` class :

```
    Time TimeLast() const;
    Time TimeNext() const ;
    Time TimeLifespan() const ;
    static Time TimeCurrent() ;
    Time TimeElapsed() const ;
    Time TimeRemaining() const ;
```

Notice that `TimeCurrent()` is a *static* function which means that all DEVS instances will have the same value of `TimeCurrent()`, while they can have different values for `TimeLast()`, `TimeNext()`, etc.

2.2.2 Atomic DEVS: Atomic

The atomic DEVS is implemented as `Atomic` in the files of `Atomic.h` and `Atomic.cpp`. `Atomic` is an *abstract* class that is derived from the abstract base class `Devs`.

```
class DEVSpp_EXP Atomic: public Devs {
protected:
    Atomic(const string& name): Devs(name), m_cs(false) {}
    ...
```

Characteristic Functions

There are four public characteristic functions that are *pure virtual*. Thus the user *must* override them to define a concrete class from `Atomic`.

The function `init()` is used when the model needs to be reset, such as in the case of an initialization for a simulation run.

```
    virtual void init() = 0;
```

The function `tau()` returns the lifespan of the current state.

```
    virtual TimeSpan tau() const = 0;
```

The function `delta_x(const_PortValue& x)` defines the input state transition caused by an input event `x`. The return value `true` indicates that the next schedule needs to be updated by calling `tau()`. Contrarily, the return value `false` indicates that the time for the next schedule needs to be preserved.

```
    virtual bool delta_x(const PortValue& x) = 0;
```

The function `delta_y(PortValue& y)` defines the output transition by generating an output event `y`. Recall that the schedule will be updated right after this occurs, based upon the value of `tau()`.

```
    virtual void delta_y(PortValue& y) = 0;
```

Displaying State as a string

There is an other public virtual(but not pure) function `Get_s()`, that will return the current status in a string for display purposes.

```
virtual string Get_s() const { return string();}
```

Collecting Performance Functions

If we want to trace the performance of an atomic DEVS model, we need to set the flag on by using `CollectStatistics(true)`. We can also get the flag's status by calling `CollectStatisticsFlag()`. The virtual function `Get_Statistics_s()` will return a string which represents the status in terms of *collecting statistics*. Also, the user can override the `GetPerformance()` function to collect the performance index.

```
void CollectStatistics(bool flag = true) { m_cs = flag; }
bool CollectStatisticsFlag() const { return m_cs; }
virtual string Get_Statistics_s() const { return Get_s(); }
virtual map<string, double> GetPerformance() const;
```

We will see the theoretical background of performance indices and how we collect them using DEVS++ in Chapter 4.

2.2.3 Coupled DEVS: Coupled

The coupled DEVS is implemented as the class `Coupled` derived from the class `Devs`. `Coupled` class is concrete and it has a constructor. It also has a destructor in which all sub-components are deleted.

```
class DEVSpp_EXP Coupled: public Devs {
public:
    Coupled(const string& name=""): Devs(name) {}
    virtual ~Coupled();
```

Sub-components Related

There are three main functions associated with modeling of sub-components as follows.

```
void AddModel(Devs* md);
Devs* GetModel(const string& name) const;
void RemoveModel(Devs* md);
```

Couplings Related

Related to couplings, there are three constructing functions, one each for the external input couplings (EICs), the internal couplings (ITCs), and the external output couplings (EOCs).

```
void AddCP(InputPort* spt, InputPort* dpt); // EIC
void AddCP(OutputPort* spt, InputPort* dpt); // ITC
void AddCP(OutputPort* spt, OutputPort* dpt);// EOC
```

In addition, we can print out the coupling information by calling `PrintEICs()`, `PrintITCs()`, `PrintEOCs()`, and `PrintCouplings()` for printing EICs, ITCs, and EOCs, and all of them, respectively.

```
void PrintEICs() const;
void PrintITCs() const;
void PrintEOCs() const;
void PrintCouplings() const;
```

The corresponding removing functions are as follows.

```
void RemoveCP(InputPort* spt, InputPort* dpt); // EIC
void RemoveCP(OutputPort* spt, InputPort* dpt); // ITC
void RemoveCP(OutputPort* spt, OutputPort* dpt);// EOC
```

2.3 Scalable Real-Time Engine: SRTEngine

DEVS++ provides a simulation engine class, called `SRTEngine` which is a concrete class. When we make an instance of `SRTEngine`, its constructor creates an independent simulation thread from the main thread.

2.3.1 Constructor

```
SRTEngine(Devs& modl, Time ending_t = DBL_MAX, CallBack cbf=NULL);
```

The constructor needs three arguments: the first argument is the `Devs` model to be simulated, the second is the simulation terminating time, the last is a callback function that is used to inject a user-input into the simulation model.

`Callback` function's type is defined as

```
PortValue (*CallBack)(Devs& md).
```

It returns a `PortValue` which can be a pair of an *input port* and a *value*. The associated input port should belong to `Devs md`. The following example shows that `InjectMsg` returns a `PortValue` whose port is `vm`'s `ip` input port.

```
PortValue InjectMsg(Devs& md) {
    VM& vm = (VM&) md;
    return PortValue(vm.ip);
}
```

We can pass the function pointer of a callback function to an instance of `SRTEngine` as follows.

```
SRTEngine simEngine(vm, 10000, InjectMsg);
```

2.3.2 Console Menu

If we call the `RunConsoleMenu()` function of `SRTEngine`, it provides a console menu as follows.

```
scale, step, run, mrun, [p]ause, pause_at, [c]ontinue, reset,
rerun, [i]nject, dtmode, animode, print, cls, log, [e]xit
>
```

Let's take a look at each menu item.

scale *f*

`scale` controls the speed of time flow by the scale factor f

- 0.1 for 10 times slower than real time
- 1 as fast as real time;
- 10 for 10 times faster than real time;
- 0 or greater than 1000,000 for as fast as possible;

step

`step` executes a simulation run until one internal transition is fired. After that it pauses the run automatically unless the user inputs commands such as `step`, `continue`, `run`, `mrun`. This command can be useful when we try a step-by-step run to see the model behavior.

run

`run` executes a simulation run which continues until it reaches the simulation ending time, which is set by the second argument of the `SRTEngine` constructor or by the command `pause_at` *et*.

mrun *n*

`mrun` executes *n* simulation runs. Each simulation run stops when it reaches the simulation ending time. When trying `mrun n`, where 2≤ `n` ≤ 20, SRTEingine calculates the 95% confidence interval of the average values of each statistical items.

[p]ause

`pause` or `p` pauses a simulation run immediately.

pause_at *et*

`pause_at` sets the simulation ending time as *et*.

[c]ontinue

`continue` or `c` resumes a simulation run which has been paused. It continues the previous simulation mode that had been determined by `step`, `run`, or `mrun`.

reset

`reset` initializes the associated simulation model.

rerun

`rerun` combines `reset` and `run`.

[i]nject

`inject` or `i` injects an *user-input event* into the simulation model. This command invokes the callback function whose type is `PortValue callback(Devs& md)`. This is the third argument of the SRTEngine constructor.

dtmode

`dtmode` sets the print mode of discrete transition, both for in the console and in the log file (whose file name is `devspp_log.txt`). The choice can be one of the following options:

- `none` displays no discrete state transition.
- `rel` displays relative mode in which lifespan and elapsed time are displayed.
- `abs` displays absolute mode in which last event time and n ext event time are displayed.

- `nc` no change.

animode

`animode` sets the animation interval. The choice can be either one of the following options.

- `none` displays no animation state transition.
- `ani` is the number of animation interval > 1.0E-2.
- `nc` no change.

print

`print` displays information according to the following option.

- `q` prints the total state of the model.
- `cpl` prints the couplings information if the model is a coupled DEVS.
- `s` prints all settings. The following screen shot is made by `print s`.

```
scale factor: 1
run-through mode
current time: 0
simulation ending time: 1.79769e+308
current dt_mode: absolute
current animation mode:  on and interval= 0.25
current log setting: on, p00
```

- `p` prints the performance indices at the current time.

cls

`cls` clears the screen.

log

`log` sets the logging option which generates the log file `devspp_log.txt`. After the `log` command, DEVS++ shows the current log settings and waits for the user input as follows.

```
current log setting: on, p00
options: {on,off}, {+,-}{pqt} nc >
```

The user options are `on` or `off` or `{+,-}{pqt}` or `nc`. Their meanings are:

- {on, off} is the main log options. Use on for turning log on or off for turning log off. If the mode is on, three independent options are selectable.
  - p is for logging performance indices at the end of a simulation run.
  - q is for logging the total state of the model at the end of a simulation run.
  - t is for logging every single discrete event transition.

  If all of three are on, it is shown as pqt. If p is on, q and t are off, the display is shown as p00, etc.
- {+,-}{pqt} can be interpreted that + stands for setting the following options on, while - stands for turning the following options off. For example +qt means to set q and t on, while -p means to set p off.
- nc no change.

[e]xit

exit or e exits the console menu.

Table 2.1 summarizes API functions of SRTEngine related to menu items that we have introduced so far.

2.4 Random Variable

DEVS++ modified the random number generator that was provided with the ADEVS engine [Nut00] in 2004. rv class defined in RNG.h is a random variable whose default constructor rv() sets its seed number as the current time. There are four probability density functions that can be used to select a random number: uniform, triangular, exponential, normal.

1. uniform(a,b) returns a random number having a uniform PDF over the closed interval [a,b].
2. triangular(a,b,c) returns a random number having a triangular PDF over the closed interval [a,b] with mode c, where c in [a,b].
3. exp(m) returns a random number having an exponential PDF with mean m.
4. normal(m,s) returns a random number having a normal PDF with mean m and standard deviation s.

Table 2.1: APIs related to Menu Items

Menu Item	SRTEngine's APIs
scale	`double GetTimeScale() const;` `void SetTimeScale(double ts);`
step	`void Step();`
run	`void MultiRun(1);`
mrun n	`void MultiRun(unsigned n);`
pause	`void Pause();`
pause_at	`Time GetEndingTime() const;` `void SetEndingTime(Time et);`
continue	`void Continue();`
reset	`void Reset();`
rerun	`void Rerun();`
inject	`void Inject(PortValue x);`

dtmode	`void Set_dtmode(PrintStateMode flag);`
	`void Get_dtmode(PrintStateMode& flag) const;`
	where
	`enum PrintStateMode {P_NONE, P_relative, P_absolute};`

animode	`void SetAnimationFlag(bool flag);`
	`bool GetAnimationFlag() const;`
	`void SetAnimationInterval(TimeSpan ai);`
	`TimeSpan GetAnimationInterval() const;`

print	`void PrintTotalState() const;`
	`void PrintCouplings() const;`
	`void PrintSettings() const;`
	`void PrintPerformanceOfaRun() const;`

log	`static void SetLogOn(bool flag=true);`
	`static void SetLogPerformance(bool flag=true);`
	`static void SetLogTotalState(bool flag=true);`
	`static void SetLogTransition(bool flag=true);`
	`static bool GetLogOn();`
	`static bool GetLogPerformance();`
	`static bool GetLogTotalState();`
	`static bool GetLogTransition();`

2.5 Miscellaneous

2.5.1 Time Span and Time

Sometimes, we are confused with two concepts: time span and time. A *time span* means the time duration between a starting time and an ending time in which the starting time and the ending time are specific values within the time horizon. In general, the time horizon consists of all the non-negative real numbers. But a time value is for a specific value within the time horizon.

In DEVS++, both `TimeSpan` and `Time` are defined as `double` in `Devs.h`.

```
typedef double TimeSpan ;
typedef double Time;
```

When we want to check if a pair `a` and `b` are the same in terms of a tolerance `tol`, we can use the following function defined in `Devs.h`.

```
bool DEVSpp::IsEqual(double a, double b, double tol=1E-3);
```

Since both types of `a` and `b` are `double`, we can be for checking for `Time` and `TimeSpan`, respectively.

We can also check if a given real number is equal to infinity by the following function

```
bool DEVSpp::IsInfinity(double a, double tol);
```

in which it calls `IsEqual(a, DBL_MAX, tol)`.

2.5.2 String Handling

String handling functions inside of the `DEVSpp` name space are available in `StrUtil.h` and `StrUtil.cpp`.

```
string STR(int v);// return int v as a string
string STR(conststring& s,int v);//s+::STR(v);
string STR(unsigned v); // return int v as a string
string STR(const string& s, unsigned v);//s+::STR(v);
string STR(double v);// return int v as a string
string STR(const string& s, double v);//s+::STR(v);

//-- split string s using delimiter c by n times if possible
vector<string> Split(const string& s, char c);

//-- return the merged string with s from f to t with delimiter c
string Merge(const vector<string>& s, unsigned f, char c);
```

```
//-- return string(pt->Parent->Name+"."+pt->Name);
string NameWithParent(DEVSpp::Port* pt);

//-- hierarchical name of child from the view of under.
//-- if under=NULL, the hierarchical name starts from the root model
string HierName(const Devs* child, const Coupled* under=NULL);
```

Chapter 3

Simple Examples

In this chapter, we will see DEVS++ examples of atomic DEVS as well as coupled DEVS.

3.1 Atomic DEVS Examples

3.1.1 Timer

An example, `Ex_Timer`, shows how to define a concrete atomic and deterministic DEVS from `Atomic`. In this example, we define a class `SimplestTimer` which generates an output, `op`, every 3.3 seconds as illustrated in Figure 3.1.

`SimplestTimer` has one output port, `op`. In the constructor, `op` is assigned by calling `AddOP`. The function `init()` does nothing because the class has no internal variable. The function `tau()` returns 3.3 all the time.

```
class SimplestTimer: public Atomic {
public:
    OutputPort* op;

    SimplestTimer(const string& name=""): Atomic(name), n(0)
    { op = AddOP("op"); }

    /*virtual*/ void init(){}
    /*virtual*/ Time tau() const {
        return 3.3;
    }
```

Since there is no input transition defined, `delta_x` has the null body. However, `delta_y` returns the output `op`.

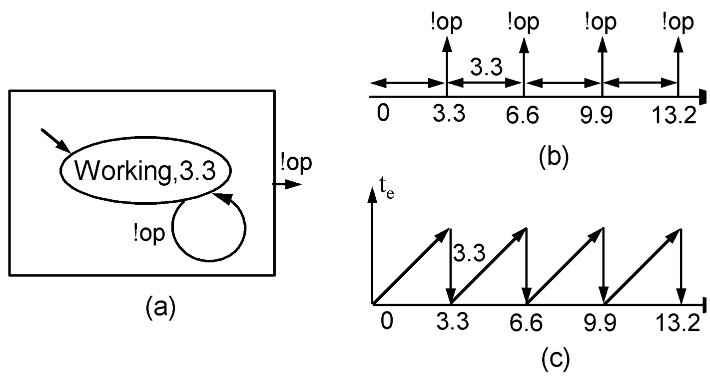

Figure 3.1: SimplestTimer (a) State Transition Diagram (b) Event Segment (c) t_e Trajectory

```
    /*virtual*/ bool delta_x(const PortValue& x) {return false;}
    /*virtual*/ void delta_y(PortValue& y)
    {
        y.Set(op);
    }
```

The display function `Get_s()` returns the current status, which is constantly `Working`.

```
    /*virtual*/ string Get_s() const {
        return string("Working");
    }
};
```

If you try `step`, you can see the animation is increasing the elapsed time. The following display shows the state at time 2.188 where the schedule time `t_s`=3.3 and the elapsed time `t_e`=2.188.

```
(STimer:Working, t_s=3.300, t_e=2.188) at 2.188
```

The simulation run will stop at 3.3 because its run mode is step-by-step when using `step`. At that time, it will display the discrete state transition as follows.

```
(STimer:Working, t_s=3.300, t_e=3.300)
 --({!STimer.op},t_c=3.3)-->
(STimer:Working, t_s=3.300, t_e=0.000)
```

The first state is the source of state transition. An arc shows a triggering event which is the output `op` of `STimer` at the current time=3.3. The second state is the destination of the state transition in which the lifespan is also 3.3 but the elapsed time has been reset to zero.

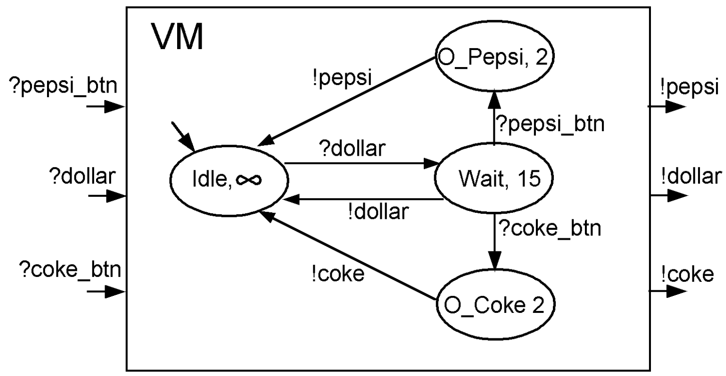

Figure 3.2: State Transition Diagram of Vending Machine

Exercise 3.1 Consider the example `Ex_Timer`.

a. Let's change the display mode from `rel` to `abs` by applying the command `dtmode`. Then preset the simulation ending time to "5" by `pause_at` 5. Now `run` until the simulation stops. When it stops at `t_c`=5, print the total state using `pinrt` with option `q`. What are the values of `t_s` and `t_e`, respectively? Guess the remaining time that `t_e` becomes `t_s` (or `t_c` becomes `t_n`) at this moment.

b. Add one more state variable `int n` in `SimplestTimer` class. `n` should be set = zero in `init()`, and it should increase by one in `delta_y()`. `Get_s()` shows `n` in the C print format of `"Working, n=%d"`.

3.1.2 Vending Machine

Consider a simple vending machine (`VM`) from which we can get Pepsi and Coke. Figure 3.2 illustrates the state transition diagram of `VM` we are considering.

There are three input events such as `?dollar` for "input a dollar", `?pepsi_btn` for "push the Pepsi button", `?coke_btn` for "push the Coke button". Similarly, we can model three output events such as `!dollar` for "a dollar out (because of timeout of menu selection)", `!pepsi` for "Pepsi out" and `!coke` for "Coke out'. [1] The state of `VM` can be either `Idle` for "Idle", `Wait` for "Wait"(that is waiting for selection of Pesi or Coke), `O_Pepsi` for "output Pepsi" and `O_Coke` for "output Coke". And their life times are: 15 time units for `Wait`, 2 time unites for both `O_Pepsi` and `O_Coke`, ∞ for `Idle` which is denoted by `inf` in Figure 3.2. [2]

[1] We use symbol ? and ! for indicating an input event and an output event, respectively.

[2] we call a state s *passive* if $\tau(s) = \infty$ or *active* otherwise ($0 \le \tau(s) < \infty$). In Figure 3.2, the state Idle is passive, the rest states are active.

At the beginning (t=0), VM is at Idle. If we put ?dollar in, it changes the state into Wait simultaneously updating $t_s = 15$ and $t_e = 0$ for the state. While in the state, if VM receives ?pepsi_btn (resp. ?coke_btn), it enters into the state O_Pepsi (resp. O_Coke) and simultaneously updates $t_s = 2$ and $t_e = 0$. While in the state O_Pepsi or O_Coke, VM ignores any input and preserves the state. Similarly, while in the state Wait, VM ignores ?dollar input.

After staying at Wait for 15 time unites, VM returns to Idle state and outputs the dollar if we don't select Pepi or Coke within the 15 time units. However, if we had selected one of them, VM changes its state into O_Pepsi (resp. O_Coke). Then after 2 time unites, VM outputs !pepsi (resp. !coke) and returns to Idle.

The example of Ex_VendingMachine shows an atomic DEVS model of VM. First of all, there are some constant strings we use for describing states as follows.

```
    const string IDLE="Idle";
    const string WAIT="Wait";
    const string O_PEPSI="O_Pepsi";
    const string O_COKE="O_Coke";
```

The class VM has three input port pointers idollar, pepsi_btn and coke_btn; three output port pointers odollar, pepsi, coke, all assigned by returning values of the AddIP and AddOP functions in the constructor.

```
class VM: public Atomic {
public:
    InputPort * idollar, *pepsi_btn, *coke_btn;
    OutputPort * odollar, *pepsi, *coke;

    VM(const string& name=""): Atomic(name)
    {
        idollar = AddIP("dollar");
        pepsi_btn = AddIP("pepsi_btn");
        coke_btn = AddIP("coke_btn");

        odollar = AddOP("dollar");
        pepsi = AddOP("pepsi");
        coke = AddOP("coke");
        init();
    }
```

VM's initial state is set to IDLE in init(). The lifespan of each state is defined in tau() as 15, 2, 2, and infinity for WAIT, O_PEPSI, O_COKE, and IDLE, respectively.

```
/*virtual*/ void init()
{
    m_phase = IDLE;
}
/*virtual*/ Time tau() const
{
    if(m_phase == WAIT)
        return 15;
    else if(m_phase == O_PEPSI)
        return 2;
    else if(m_phase == O_COKE)
        return 2;
    else
        return DBL_MAX;
}
```

The input transition function `delta_x` defines every arc triggered by an input event in Figure 3.2 and returns `true` for each such arc. If the input event `idollar` arrives while `VM` is not in state `Idle`, or if the input events `pepsi_btn` or `coke_btn` arrive while `VM` is not in state `Wait`, `delta_x` returns `false`, and the input is ignored.

```
/*virtual*/ bool delta_x(const PortValue& x)
{
    if(m_phase == IDLE && x.port == idollar){
        m_phase = WAIT;
        return true;
    } else if(m_phase == WAIT && x.port == pepsi_btn) {
        m_phase = O_PEPSI;
        return true;
    } else if(m_phase == WAIT && x.port == coke_btn) {
        m_phase = O_COKE;
        return true;
    }else
        return false;
}
```

The output transition function `delta_y` defines every arc generating an output event in Figure 3.2.

```
/*virtual*/ void delta_y(PortValue& y)
{
    if(m_phase == WAIT)
```

```
            y.Set(odollar);
        else if(m_phase == O_PEPSI)
            y.Set(pepsi));
        else if(m_phase == O_COKE)
            y.Set(coke);
        m_phase = IDLE;
    }
```

The virtual function `Get_s()` is also overridden and returns an `m_phase` variable that is a `string`.

```
    /*virtual*/ string Get_s() const
    {
        return m_phase;
    }
protected:
    string m_phase;
};
```

The following example demonstrates the use of a callback function to inject a user-input into an instance of `VM`.

```
PortValue InjectMsg(Devs& md)
{
    VM& vm = (VM&)md;
    string input;
    cout << "[d]ollar [p]epsi_botton [c]oca_botton > " ;
    cin >> input;
    if(input == "d")
        return PortValue(vm.idollar);
    else if(input == "p")
        return PortValue(vm.pepsi_btn);
    else if(input == "c")
        return PortValue(vm.coke_btn);
    else {
        cout <<"Invalid input! Try again! \n";
        return PortValue();
    }
}
```

The callback function `InjectMsg` casts the type of `Devs& md` to `VM&vm`. And the user-input of either `d`, `p`, or `c` is mapped to `PortValue(vm.idollar)`, `PortValue(vm.pepsi_btn)`, or `PortValue(vm.coke_btn)`, respectively.

The last part the the code in Ex_VendingMachine runs the simulation engine. First we make vm as an instance of VM, and plug vm into an instance of SRTEngine with the simulation ending time=10000 using the above callback function.

```
void main( void ) {
    VM* vm = new VM("VM") ; //-- simulation model
    SRTEngine simEngine(*vm, 10000, InjectMsg); // see above function
    simEngine.RunConsoleMenu();
    delete vm;
}
```

Let's try the first step. Observe that since tau(IDLE)=∞ and the initial t_s=∞ also, the elapsed time t_e cannot ever reach t_s. Thus this command step doesn't stop until the t_e becomes 1000 which is the simulation ending time (unless the user interrupts the simulation).

In this case, we can stop the simulation run using pause or p, followed by Enter key. The following screen shows the situation if we make it pause at 8.859.

```
(VM:Idle, t_s=inf, t_e=8.859) at 8.859
```

Let's try inject or i. Then we can see the console output which is produced by the above InjectMsg(Devs& md) as follows.

```
[d]ollar [p]epsi_botton [c]oca_botton >
```

If we input d, we can see the input causes the state to transition from Idle to Wait as follows.

```
(VM:Idle, t_s=inf, t_e=8.859)
 --({?dollar,?VM.dollar}, t_c=8.859)-->
(VM:Wait, t_s=15.000, t_e=0.000)
```

Now, we use continue or c to resume stepping again. If we want to pause again and inject a menu selection such as pepsi_btn or coke_btn, we can do that just like before.

Exercise 3.2 Consider modifying the VM model in EX_VendingMachine in order to add the behavior of *rejecting* a second dollar input when VM is the state Wait. To model this, let's add a state Reject whose lifespan is 0. We define the output transition δ_y at Reject as delta_y(Reject) = (!dollar, Wait). However there are two ways of rescheduling of t_s and t_e of the the state Wait when VM comes back to the state. Let's try each of the following two ways.

1. Reset t_s=15 and t_e=0.
2. Make t_s and t_e back to the values they had right before the input of the additional dollar.

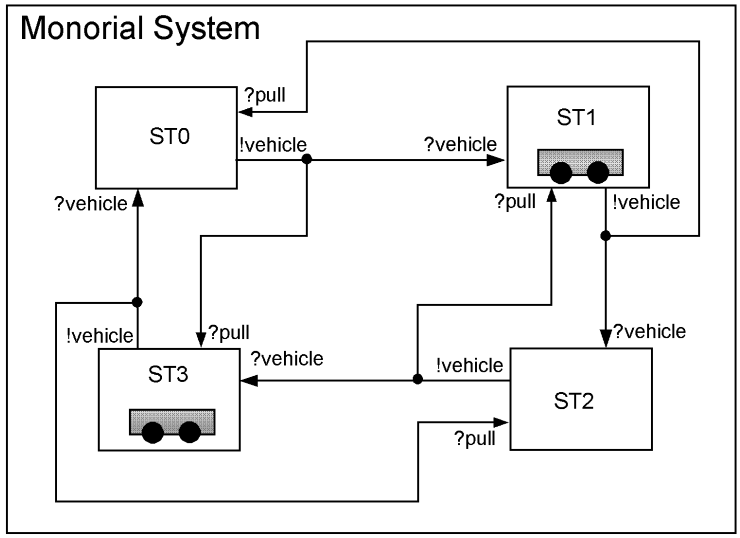

Figure 3.3: Monorail System

3.2 Coupled DEVS Examples

3.2.1 Monorail System

Figure 3.3 illustrates the configuration of a monorail system which consists of four stations whose names are `ST0`, `ST1`, `ST2` and `ST3`, respectively.

Each station, `ST0`, `ST1`, `ST2` and `ST3`, is an instance of `Station` class derived from `Atomic` such that it has an input event set $X=$ {`?vehicle`, `?pull`} and an output event set $Y=${`!vehicle`, `!pull`} and two state variables: `phase` $\in$ {Empty (`E`), Loading (`L`), Sending (`S`), Waiting (`W`), Collided (`C`)}, and `nso` $\in$ {false(`f`), true(`t`)} indicating "next station is NOT occupied" for `nso=f` or "next station is occupied" for `nso=t`.

To avoid collisions that can occur when more than one vehicle attempts to occupy a station (let's call it A) at the same time, the station prior to A (let's call it B) should dispatch the vehicle ONLY when B's `nso` = `f`. The phase transition diagram of a single station is shown in Figure 3.4 where an arc is augmented by *(pre-condition),(post-condition)*. For example, when a station receives `?p` at `phase=E`, it makes `nso=f`; if `phase=L` and `nso=f`, then when it receives `?p`, it changes into `phase=S` internally without any output indicated by `!`ϵ. The symbols `?v`, `?p`, and `!v` in Figure 3.2 stand for `?vehicle`, `?pull`, and `!vehicle`, respectively.

The loading time lt is assigned as $lt = 10$ for `ST0`, `ST2`, `ST3`; $lt = 30$ for `ST1` (because `ST1` is bigger than the rest other three stations). The initial state for

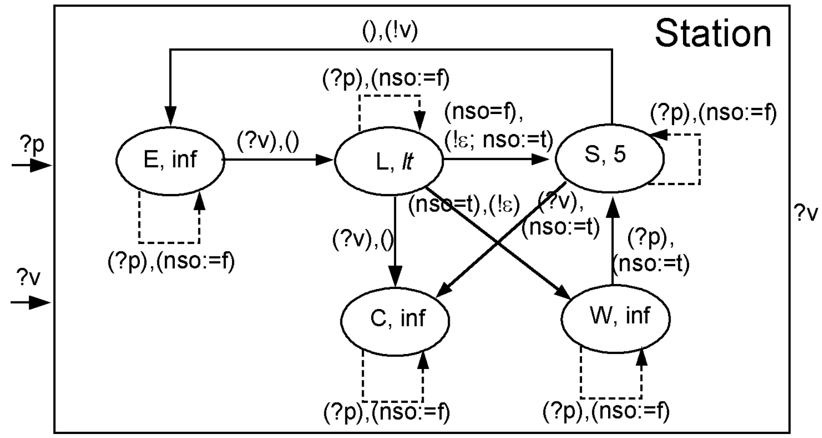

Figure 3.4: Phase Transition Diagram of Station
(A dashed line indicates $\delta_x(s, t_s, t_e, x) = (s', 0)$.)

each station is $s_0 = (\mathtt{E}, \mathtt{t})$ for `ST0` and `ST2`, $s_0 = (\mathtt{L}, \mathtt{f})$ for `ST1` and `ST3`.

To model and simulate this monorail system, we build `Station` as follows.

Station

First of all, we define several constant strings for indicating the phase of `Station`. And a macro `REMEMBERING` is defined for testing the effect of monitoring the next station's status using `nso`.

```
const string EMPTY="E";
const string LOADING="L";
const string SENDING="S";
const string WAITING="W";
const string COLLIDED="C";

#define REMEMBERING // for testing the effect of using nso
```

The class `Station` has several state variables: a string `m_phase`; `bool init_occupied` indicating the initial occupation state of the station, `bool nso` which indicates if the next station is occupied or not; and the constant variable `TimeSpan loading_t` indicating the lifespan of a state when its phase is `LOADING`.

`Station` has two input port pointers `ipull` and `ivehicle`, one output port pointer `ovehicle`. These variables, including ports, are assigned in the constructor as follows.

```
class Station: public Atomic {
```

```
public:
    string m_phase;
    bool   init_occupied;
    bool   nso;//next_state_occpied
    const TimeSpan loading_t;

    InputPort* ipull, *ivehicle;
    OutputPort* ovehicle;

    Station(const string& name, bool occupied, TimeSpan lt):
        Atomic(name), init_occupied(occupied), loading_t(lt), nso(true)
        {
            ipull = AddIP("pull"); ivehicle = AddIP("vehicle");
            ovehicle = AddOP("vehicle");
            init();
        }
```

`Station::init()` initializes `m_phase` depending on `init_occupied` such that `m_phase` = `SENDING` if `init_occupied` is `true`, otherwise, `m_phase` = `EMPTY`.

```
    /*virtual*/ void init()
    {
        if(init_occupied == true)
            m_phase = SENDING;
        else
            m_phase = EMPTY;
        //cout << Name << ":" << Get_s()<<endl;
    }
```

`Station::::tau()` returns the lifespan of each state; 10 for `SENDING`; `loading_t` for `LOADING`; ∞ otherwise.

```
    /*virtual*/ TimeSpan tau() const
    {
        if (m_phase == SENDING)
            return 10;
        else if (m_phase == LOADING)
            return loading_t;
        else
            return DBL_MAX;
    }
```

`Station::delta_x` defines the input transition such that if it receives an input through `ipull`, it sets `nso = false`. At that time, if the station's phase is

WAITING, then nso had previously been set by true for remembering that the next station had been occupied, delta_x then changes the phase to SENDING and returns true.

When a station receives a vehicle through ivehicle port, if phase is EMPTY, its phase changes into LOADING; otherwise the phase changes into COLLIDED.

```
/*virtual*/ bool delta_x(const PortValue& x)
    {
        if( x.port == ipull) {
            nso = false;
            if( m_phase == WAITING){
#ifdef REMEMBERING
                nso = true;
#endif
                m_phase = SENDING;
                return true;
            }
        }
        else if(x.port == ivehicle) {
            if(m_phase == EMPTY)
                m_phase = LOADING;
            else // rest cases lead to Colided!
                m_phase = COLLIDED;
            return true;
        }
        return false;
    }
```

Station::delta_y defines the output transition behavior such that, at the end of LOADING phase, if nso=true, then delta_y changes the stations' phase into WAITING. But if nso=false, delta_y marks nso=true for remembering the next station's occupation and changes the station's phase to SENDING. At the end of SENDING phase, it sends out the vehicle through ovehicle port and changes the station's phase to IDLE.

```
     /*virtual*/ void delta_y(PortValue& y) {
        if(m_phase == LOADING){
            if(nso == true)
                m_phase = WAITING;
            else {
#ifdef REMEMBERING
                nso = true;
#endif
```

```
                m_phase = SENDING;
            }
        }  else if(m_phase == SENDING) {
            y.Set(ovehicle);
            m_phase = EMPTY;
        }
    }
```

The displaying function `Get_s()` is overridden to return a string containing information about `m_phase` and `nso` as follows.

```
    /*virtual*/ string Get_s() const
    {
        string str = "phase="+m_phase +",nso=";
        if(nso) str +="true";
        else    str +="false";
        return str;
    }
```

Monorail System

To construct the monorail system, we will make four instances from `Station`. `Stations` `ST1` and `ST3` each have one vehicle initially, the other two have none, while the loading time of `ST1` is 30 time-units, the other three each have a loading time of 10.

Each station will collect its own performance data. All couplings are connected as shown in Figure 3.3.

```
Coupled* MakeMonorail(const char* name) {
    Coupled* monorail = new Coupled(name);
    //-- Add Station 0 to 3 ----
    Station* ST0 = new Station("ST0", false, 10);
    ST0->CollectStatistics();

    Station* ST1 = new Station("ST1", true, 30);
    ST1->CollectStatistics();

    Station* ST2 = new Station("ST2", false, 10);
    ST2->CollectStatistics();

    Station* ST3 = new Station("ST3", true, 10);
    ST3->CollectStatistics();
```

```
    monorail->AddModel(ST0);
    monorail->AddModel(ST1);
    monorail->AddModel(ST2);
    monorail->AddModel(ST3);
    //-------------------------------------------
    //-------- Add internal couplings ------------
    monorail->AddCP(ST0->ovehicle, ST1->ivehicle);
    monorail->AddCP(ST1->ovehicle, ST0->ipull);

    monorail->AddCP(ST1->ovehicle, ST2->ivehicle);
    monorail->AddCP(ST2->ovehicle, ST1->ipull);

    monorail->AddCP(ST2->ovehicle, ST3->ivehicle);
    monorail->AddCP(ST3->ovehicle, ST2->ipull);

    monorail->AddCP(ST3->ovehicle, ST0->ivehicle);
    monorail->AddCP(ST0->ovehicle, ST3->ipull);
    //-------------------------------------------
    return monorail;
}
```

If you try the command `run`, DEVS++ will simulate system performance until it reaches the simulation ending time of 1000 time units. The default simulation speed of DEVS++ is the real time so it will take 1000 seconds in reality. However, the user don't have to wait until the simulation ending time. Don't forget to use the command `pause` to stop a simulation run any time you want.

We can change the simulation speed as maximum by `scale 0` . If you don't care of animation output, you can set `animode none`. In addition, if you don't want to see the status of discrete state transitions, you can set `dtmode none` too.

The following screen is the results of the command `print p`.

```
CPU Run Time: 12.375000 sec.
mr.ST0
phase=E,nso=false: 0
phase=E,nso=true: 0.59
phase=L,nso=false: 0.01
phase=L,nso=true: 0.19
phase=S,nso=true: 0.2
phase=W,nso=true: 0.01

mr.ST1
phase=E,nso=true: 0.21
```

```
phase=L,nso=false: 0.4
phase=L,nso=true: 0.19
phase=S,nso=true: 0.2

mr.ST2
phase=E,nso=false: 0.2
phase=E,nso=true: 0.4
phase=L,nso=false: 0.2
phase=L,nso=true: 0
phase=S,nso=true: 0.2

mr.ST3
phase=E,nso=false: 0.19
phase=E,nso=true: 0.41
phase=L,nso=false: 0.2
phase=S,nso=true: 0.2
```

The performance index for each station is the ratio of the total time the station stays in each state divided by the simulation run time of 1000. In the example above, for `mr.ST3`, `phase=L,nso=false: 0.2` indicates that the total time ST3 spent in the `LOADING` state was about 20% of the length of simulation run time of 1000. That means that station 3 spent about 200 time-units in the LOADING phase.

It is not hard to find that since `ST1::loading_t`=30 is three times longer than other stations' `loading_t`, `ST1` stays at `LOADING` about 59% of the simulation time. This causes `ST0` to transition into `WAIT` because `ST1` stays so long at `LOADING`.

Exercise 3.3 Let's comment out the line of "`#define REMEMBERING`" in `Station.h` of `Ex_Monorial` example. Build it again and try `run`. When the run stops, try `print q` and `print p`. Is there a station which gets into `COLLIDED`?

Chapter 4

Performance Evaluation

This section introduces several performance indices in Section 4.1 and shows how to calculate them in Section 4.2.

4.1 Performance Measures

This section introduces four performance indices: Throughput, Cycle Time, Utilization, and Average Queue Length.

4.1.1 Throughput

It is not hard to imagine that a system produces products. In this context, we can think of a performance index for the system that answers the question "how may products does this system produce?" This performance index can be measured by *counting the number of products* produced by the system over particular time period.

If we have $x \in \mathbb{N}$ jobs produced by the system over an observational time span t_o, then the system throughput $thrp$ is

$$thrp = \frac{x}{t_o} \tag{4.1}$$

and its unit of measurement is jobs/time-unit.

Example 4.1 (Throughput) If the number of products produced by a system is 2500 during 100 minutes, then its throughput is $thrp = 2500/100 = 25$ jobs/min. □

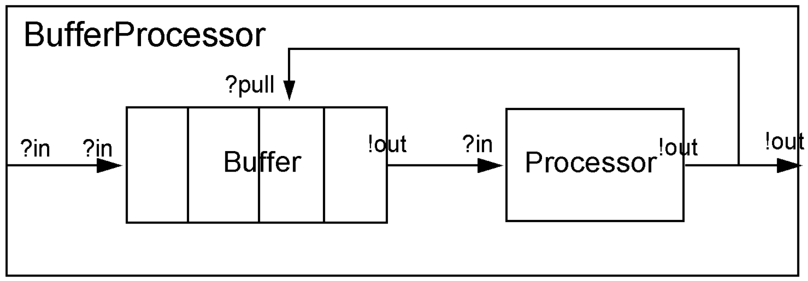

Figure 4.1: A System having a Buffer and a Processor

4.1.2 Cycle Time

A system performs a set of activity cycles so its performance can be measured by how long it has taken to perform an activity cycle. The unit of this measure is time-unit/activity.

Given a event set Z, let a *timed event* be a pair of an event $z \in Z$ and its occurrence time $t \in \mathbb{T}$. Then an *activity* consists of a pair of $((z_l, t_l), (z_u, t_u))$ such that $t_l \leq t_u$. Given an activity $a = ((z_l, t_l), (z_u, t_u))$, its *duration* or *cycle time*, denoted $d(a)$ is defined

$$c(a) = t_u - t_l. \tag{4.2}$$

Given an activity set A, the (average) *cycle time* of A is

$$t_{cyc}(A) = \frac{\sum\limits_{a \in A} c(a)}{|A|}. \tag{4.3}$$

Since cycle time is defined over a given activity set, it can be interpreted differently depending on contexts of activity sets. For example, in the system which consists of a buffer and a processor as shown in Figure 4.1, the *system time* can be measured over the entire processing activity from arrival to departure of the `BufferProcessor` system. Also *waiting time* can be considered as the time duration for the waiting activity in `Buffer`, while *processing time* can be the time duration between arrival to and departure from `Processor`.

Without loss of generality, we normally consider an activity set A that has a homogenous events pair, i.e. for $a_1 = ((z_{l1}, t_{l1}), (z_{u1}, t_{u1}))$ and $a_2 = ((z_{l2}, t_{l2}), (z_{u2}, t_{u2})) \in A$, then $z_{l1} = z_{l2}$ and $z_{u1} = z_{u2}$. In this kind of activity set A, events themselves are not critical to compute the cycle time (even though it makes much clear our understanding when events are available.). Thus, we can see the average cycle time of an activity set A as the average of length of two times t_l and t_u

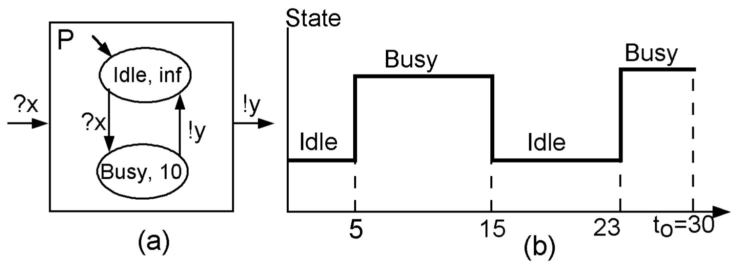

Figure 4.2: State Trajectory of a Processor

$$t_{cyc}(t(A)) = \frac{\sum_{(t_l, t_u) \in t(A)} t_u - t_l}{|t(A)|} \tag{4.4}$$

where $t(A) = \{(t_l, t_u) : ((z_l, t_l), (z_u, t_u)) \in A\}$.

Example 4.2 (Cycle Time as System Time) Assume we have the set of time pairs $A = \{((a, 5), (d, 17)), ((a, 7), (d, 29)), ((a, 15), (d, 41)), ((a, 50), (d, 62))\}$ where a is for "arrival" event, and d for "departure" `BufferProcessor` system in Figure 4.1. Since $t(A) = \{(5, 17), (7, 29), (15, 41), (50, 62)\}$, the system time is $t_{cyc}(A) = t_{cyc}(t(A)) = (12 + 21 + 26 + 12)/4 = 17.75$. □

4.1.3 Utilization

Conventionally the definition of *utilization* is the percentage of the *working time of a machine compared to its total running time*. Let's consider a processor `P` as shown in Figure 4.2(a) which has two states: `Busy`, which is defined as working time, and `Idle`, which is defined as "running, but not working" time. Once it receives an input `?x`, it processes the input and then generates output `!y` after 10 time units. Figure 4.2(b) illustrates a state trajectory of the processor terminating at $t_o = 30$. In this trajectory, the total time span of `Busy` is (15-5)+(30-23)=17, so utilization of the processor is 56.7%=(17/30)*100, while `idle`'s percentage is 100-56.7=43.3%.

We can generalize this concept to more than two states. Let's consider the vending machine introduced in Section 3.1.2. Suppose that we have a state trajectory of the vending machine as shown in Figure 4.3. This state trajectory can be seen as a sequences of piece-wise constant segments. The time it takes to transition between states is assumed to be zero.

The *time duration* at a piece-wise constant segment is defined by

$$td : S \times \mathbb{N} \rightarrow \mathbb{T} \tag{4.5}$$

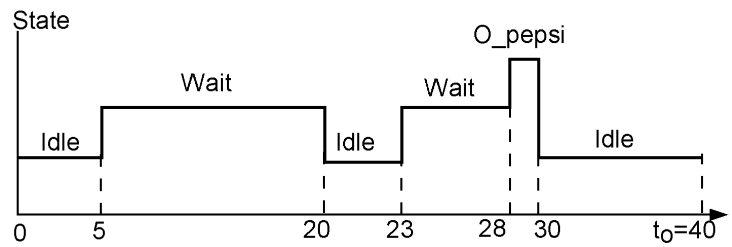

Figure 4.3: A State Trajectory of Vending Machine

where $\mathbb{N}$ is a set of natural numbers. The natural number $i \in \mathbb{N}$ of this function $td(s, i)$ indicates the order i of the segment whose state is s. For example, in the state trajectory of Figure 4.3, $td(\texttt{Idle}, 1) = 5 - 0 = 5$, $td(\texttt{Idle}, 2) = 23 - 20 = 3$, $td(\texttt{Idle}, 3) = 40 - 30 = 10$ and $td(\texttt{Idle}, n) = 0$ for $n = 4, 5, \ldots$.

Let C be the current state. Then the probability that the current state is $s \in S$ over time from 0 to t_o, denoted by $P(C = s)$, is

$$P(C = s) = \frac{\sum_{i \in N} td(s, i)}{t_o}. \tag{4.6}$$

It is true that

$$\sum_{s \in S} \sum_{i \in N} td(s, i) = t_o. \tag{4.7}$$

So it is also true that

$$\sum_{s \in S} P(C = s) = \sum_{s \in S} \left(\frac{\sum_{i \in N} td(s, i)}{t_o} \right) = \frac{t_o}{t_o} = 1. \tag{4.8}$$

Example 4.3 Consider the state trajectory of Figure 4.3. Then $P(C = \texttt{Idle})$ = (5+3+10)/40 = 0.45, $P(C{=}\texttt{Wait})$ = (15+5)/40 =0.5, $P(C{=}\texttt{O\_pepsi})$=2/40 =0.05, $P(C{=}\texttt{O\_coke})$=0. □

Exercise 4.1 Assume that we have a processor as shown in Figure 4.2(a). From the processor, we have an event segment $\omega_{[0,50]} = (?x, 10)(!y, 20)(?x, 35)(!y, 45)$ where (z, t) means an event z occurs at $t \in \mathbb{T}$ and the observation was performed from 0 to 50. Calculate $P(C{=}\texttt{Idle})$ and $P(C{=}\texttt{Busy})$ over time [0,50]. □

To calculate $P(C = s)$, we need to keep track of $\sum_i td(s, i)$ by accumulating all time durations of piece-wise constant time segments when the system is in state s. We will see how to implement this in Section 4.2.2.

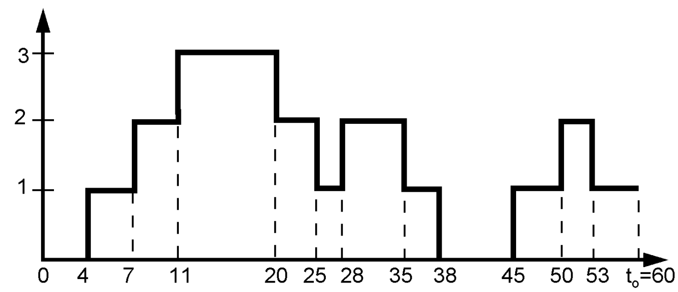

Figure 4.4: Trajectory of Queue

4.1.4 Average Queue Length

Once again, let's consider a system with a buffer and a processor that are serially connected as shown in Figure 4.1. To avoid collisions of multiple inputs at the processor, the buffer stores inputs while the processor is busy working on previous inputs.

Depending on inter-arrival times of between inputs and `Processor`'s processing time, the length of time an input waits in `Buffer` can vary widely. Thus the number of waiting inputs (queue size) can be a random number.

Recall how we developed the probability that the current state C is equal to a state s in Section 4.1.3. Let the current state C of `Buffer` be defined as the *number of inputs currently waiting in buffer*. Then the probability that the number of waiting parts C is equal to $x \in \mathbb{N}$, where $\mathbb{N}$ is a suitably defined subset of the natural numbers, over an observation time from 0 to t_o is

$$P(C = x) = \frac{\sum_{i \in \mathbb{N}} td(x, i)}{t_o} \tag{4.9}$$

The *mean* or *expected value* of C is defined by

$$E(C) = \sum_{x \in \mathbb{N}} xP(C = x) \tag{4.10}$$

The *Average Queue Length* is defined as Equation (4.10).

Example 4.4 Suppose that we have a state trajectory of a queue as shown in Figure 4.4. By Equation (4.9), we can get $P(C{=}0){=}(4{+}7)/60{=}0.183$, $P(C{=}1){=}(3{+}3{+}3{+}5{+}7)/60{=}0.35$, $P(C{=}2){=}(4{+}5{+}7{+}3)/60{=}0.317$, $P(C{=}3){=}9/60{=}0.15$. By Equation (4.10), the Average Queue Length is $E(C = x){=}0{*}0.183{+}1{*}0.35{+}2{*}0.317{+}3{*}0.15{=}1.434$. □

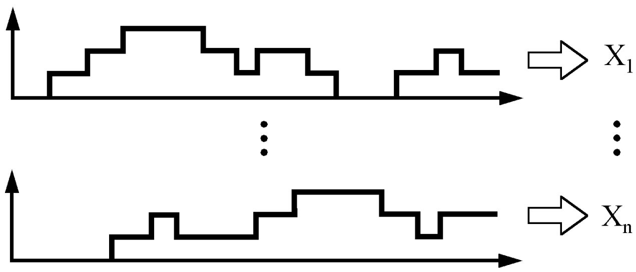

Figure 4.5: IID random variants $X_1 \dots X_n$ from n simulation runs

Since the natural number $x \in \mathbb{N}$ is the special case of a general state $s \in S$, if we can calculate $P(C = s)$ then we can also calculate $P(C = x)$ as well as $E(C)$. We will see how we implement this process in Section 4.2.3.

4.1.5 Sample Mean, Sample Variance, and Confidence Interval

If the internal components of a system behave stochastically or if its input events can occur at arbitrary times, the performance have randomness.

If we reset the model under study prior to each simulation run, the performance indices from each run are *independent* from those of all the other runs. Random variables are said to be *identically distributed* if the associated variables have identical measurement. For examples, the Utilization of `Processor` in `BufferProcessor` of Figure 4.1 from multiple simulation runs are independent and identically distributed (IID) random variable.

Suppose that we try to estimate the real mean μ of a random variable from a sample whose values are $X_1, X_2, \dots X_n$ from n simulation runs as illustrated in Figure 4.5. Then the *sample mean*

$$\hat{\mu} = \frac{\sum_{i=1}^{n} X_i}{n} \tag{4.11}$$

is an unbiased (point) estimator of the real mean μ. Similarly, the *sample variance*

$$\hat{\sigma}^2(n) = \frac{\sum_{i=1}^{n} [X_i - \hat{\mu}]^2}{n-1} \tag{4.12}$$

is an unbiased estimator of the real variance σ^2. For $n \geq 2$, a $100(1-\alpha)$ percent confidence interval for μ is given by

$$\hat{\mu} \pm t_{n-1,1-\alpha/2} \sqrt{\frac{\hat{\sigma}^2(n)}{n}} \tag{4.13}$$

where $t_{n-1,1-\alpha/2}$ is the upper $1-\alpha/2$ critical point for the t distribution with $n-1$ degree of freedom. It can be written

$$P\left[\hat{\mu} - t_{n-1,1-\alpha/2}\sqrt{\frac{\hat{\sigma}^2(n)}{n}} \leq \mu \leq \hat{\mu} + t_{n-1,1-\alpha/2}\sqrt{\frac{\hat{\sigma}^2(n)}{n}}\right] = 1-\alpha \qquad (4.14)$$

and we say that we are 100(1-a) percent confident that the real μ lies in the interval given by Equation (4.13).

Example 4.5 Suppose that 10 simulation runs produce system throughput data of 12.0, 15.0, 16.8, 18.9, 9.5, 14.9, 15.8, 15.5, 5.0, and 10.9. Our objective is to build the 90 % confidence interval for μ. We have t-distribution values of $t_{10,0.9}$=1.372, $t_{10,0.95}$=1.812, $t_{9,0.9}$=1.383, $t_{9,0.95}$=1.833.

Then $\hat{\mu}$=13.4 and $\hat{\sigma}^2$=16.75 and the 90% confidence interval for μ is $\hat{\mu} \pm t_{9,0.95}\sqrt{\frac{\hat{\sigma}^2(n)}{n}} = 13.4 \pm 1.83\sqrt{\frac{16.75}{10}} = [11.03, 15.77]$ □

The values of $t_{n-1,1-\alpha/2}$ of t pdf are available in many statistics books and simulation books [Zei76, LK91]. DEVS++ calculates the 100(1-α) confidence interval for μ when using `mrun n` for 2 ≤`n`≤ 20 in verion 1.4.2. We will see it in detail in Section 4.3.

4.2 Practice in DEVS++

This section addresses how we can calculate the performance indices using DEVS++. All classes used in this section are available in `DEVSpp/Examples/Ex_ClientServer` folder.

4.2.1 Throughput and System Time in DEVS++

Throughput can be collected by counting flow entities coming out of the system under study, while System Time can be collected by tracing the arrival time and the departure time of each flow entity. [1] To do this, we will use two atomic models: `Generator` and `Transducer`, which are key models in the experimental frame.

Counting flow entities coming from the system can be done by `Transducer`. For collecting system time, we will need the cooperation of both `Generator` and `Transducer`.

[1]Flow entities can be clients of a bank, products of a manufacturing system, airplanes of an airport, and messages of a communicating network.

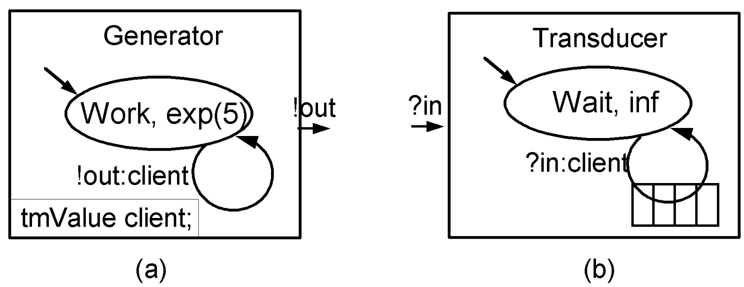

Figure 4.6: Generator and Transducer in Ex_ClientServer

Generator

The state transition diagram of `Generator` is shown in Figure 4.6(a). This model has an output port `out` and `tmValue`-type client which will be used for cloning the client every generating time.

```
class Generator: public Atomic {
public:
    OutputPort* out;
    tmValue client;
public:
    Generator(const string& name=""): Atomic(name),
        client() { out = AddOP("out"); }
```

`Generator::tau()` returns a random value from an exponential pdf with mean 5.

```
    /*virtual*/ Time tau() const
    {
        static rv erv;
        TimeSpan t = erv.exp(5);
        return t;
    }
```

`Generator::detla_y()` makes a clone of `client` and assigns it `pClient`. `pClient` is stamped by ("SysIn",CurrentTime) and it is sent out of `Generator` through `out` port.

```
    /*virtual*/ void delta_y(PortValue& y)
    {
        tmValue* pClient = (tmValue*) client.Clone();
        //-- (event, time) stamping
        pClient->TimeMap.insert(make_pair("SysIn",Devs::TimeCurrent()));
```

```
        y.Set(out, pClient);
    }
```

Generator's init() and delta_x are omitted here because they have no functionality.

Transducer

Transducer's behavior is pretty much opposite to that of Generator. Figure 4.6(b) shows its state transition diagram. It has an input port in and a buffer Collector as deque<tmValue*> to collect tmValues coming in. Transducer also contains a destructor called init().

```
class Transducer: public Atomic {
protected:
    InputPort* in;
    deque<tmValue*> Collector;
public:
    Transducer(const string& name=""): Atomic(name)
    {
        CollectStatistics(true);
        in = AddIP("in");
    }
    virtual ~Transducer() { init(); }
```

Transducer::init() clears all clients in Collector. Transducer::tau() returns ∞ all the time so it is passive.

```
    /*virtual*/ void init() {
        while(Collector.size()>0) // delete all pv in Collector
        {
            tmValue* pv = Collector[0];
            Collector.pop_front();
            delete pv;
        }
    }
    /*virtual*/ Time tau() const { return DBL_MAX; }
```

Transducer::delta_x() castes the input value x.value to pv of tmValue type. It stamps pv with ("SysOut",CurrentTime), and pushes pv into Collector. Since Transducer is always passive, it has no output, and so delta_y() is not needed here;

```
/*virtual*/ bool delta_x(const PortValue& x)
    {
```

```
        tmValue* pv = dynamic_cast<tmValue*>(x.value);
        if(pv)
        {
            //-- (event, time) stamping
            pv->TimeMap.insert(make_pair("SysOut", Devs::TimeCurrent()));
            Collector.push_back(pv); // delete contents later in int();
        }else
            THROW_DEVS_EXCEPTION("Type casting Failed!");
        return false;
    }
```

Recall that `Transducer` collects incoming `tmValues` stamped with ("SysIn",arrival-time) by `Generator`, ("SysOut",departure-time) by `Transducer`. Using these data, `GetPerformance()` of `Transducers` returns {("Throughput", value) and ("Average System Time", value) } as follows.

- Throughput value defined in Equation (4.1) is the number of `tmValues` in `Collector` divided by the current time.
- System Time defined in Equation (4.3) is the average value of all time durations (arrival-time, departure-time) for each `tmValue` in `Collector`.

The following `Transducer::GetPerformance()` returns these two indices.

```
    /*virtual*/ map<string, double> GetPerformance() const
    {
        map<string, double> statistics;
        if(m_cs) {
            string str = "Throughput";
            statistics.insert(make_pair(str,
                    Collector.size()/TimeCurrent()));

            TimeSpan average_st=0;
            for(int i=0; i<(int)Collector.size(); i++){
                tmValue* pv = Collector[i];
                TimeSpan system_t = pv->TimeMap["SysOut"] -
                                        pv->TimeMap["SysIn"];
                average_st += system_t;
            }
            average_st = average_st / (double)Collector.size();
            str = "Average System Time";
            statistics.insert(make_pair(str, average_st));

        }
```

```
        return statistics;
    }
```

4.2.2 Utilization in DEVS++

Recall that to get Utilization, we need to accumulate the time intervals of piece-wise constant time-segments associated with a state. Accumulating the time intervals can be done using the criterion of either "as long as possible" or "as short as possible". "Longer" is preferred over "shorter" because it requires less computational burden.

If we accumulate the time interval in cases

(1) when the constant segment might change at discrete event points, or

(2) when the simulation run stops

the "as long as" preference might be achieved. For example, in Figure 4.3, times at t =5, 20, 23, 28, 30 for case (1) (discrete state transitions) and also at t =40 for case (2) (simulation stop time).

DEVS++ calls the following function `when_receive_cs` for collecting the time interval of a state segment in cases of above (1) and (2).

```
void Atomic::when_receive_cs() {
    Time dT = TimeCurrent() - t_Lcs;// dT: accumulating time span
    if(CollectStatisticsFlag() == true)
    {
        string state_str = Get_Statistics_s();
        if(m_statistics.find(state_str) == m_statistics.end())
            m_statistics.insert(make_pair(state_str, 0.0));// new entry
        m_statistics[state_str] += dT;//add dT to staying time
    }
    t_Lcs = TimeCurrent(); // update t_Lcs as the current time.
}
```

The function description of `when_receive_cs()` shows that it records and accumulates the time interval `dT` from the last time we called `when_receive_cs()` to the current time if the flag of collecting statistics is `true`.

We are using `m_statistics` (defined as `map<string, double>`) to collect statistics. The key value of piece-wise constant segment will be a string returned from `Get_Statistics_s()`.

If the string of `Get_Statistics_s()` was not yet registered in `m_statistics`, the pair`(state_str,0.0)` will be newly registered in `m_statistics` where `state_str=Get_Statistics_s()`. The value of `m_statistics[state_str]` is increased by `dT`. Finally, `t_Lcs` that is the last time when we calls `when_receive_cs()` is updated by the current time.

Every time we need to print the current statistics (such as when we use the command `print p`), DEVS++ shows performance indices by calling each model's overriding `GetPerformance()`. The default implementation of `Atomic::GetPerformance()` is as follows.

```
/*virtual*/ map<string, double> Atomic::GetPerformance() const {
    map<string, double> statistics;
    if(CollectStatisticsFlag()==true) {
        for(map<string, double>::const_iterator it = m_statistics.begin();
            it != m_statistics.end(); it++)
        {
            double probability = it->second / TimeCurrent();

            if(probability < 0.0 || probability > 1.0) {
                THROW_DEVS_EXCEPTION("Invalid Probability!");
            }
            else
                statistics[it->first] = probability;
        }
    }
    return statistics;
}
```

As we can see, `Atomic::GetPerformance()` returns a `map<string,double>` such that `statistics[key] = m_statistics[key]/TimeCurrent()`.

Thus `statistics[key]` contains the $P(C{=}\texttt{key})$ of Equation (4.6) over the interval from 0 to the current time.

4.2.3 Average Queue Length in DEVS++

The class `Buffer` in `Ex_ClientServer` shows how to collect the average queue length. The default implementation of `Get_Statistics_s()` at `Atomic` is to return `Get_s()`. However, `Buffer` overrides the `Get_Statistics_s()` such that it returns the number of jobs waiting in a buffer as follows.

```
/*virtual*/ string Buffer::Get_Statistics_s() const
{
    char tmp[10]; sprintf(tmp, "%d",(int)m_Clients.size());
    return string(tmp); // length Only
}
```

The class `Buffer` inherits `Atomic::when_receive_cs()` shown in the previous section. But it overrides `GetPerformance()` function as follows.

```
/*virtual*/ map<string, double> Buffer::GetPerformance() const
{
    map<string, double> statistics;
    if(CollectStatisticsFlag()==true) {
        TimeSpan E_i=0;// expectation of queue length
        for(map<string, double>::const_iterator it = m_statistics.begin();
            it != m_statistics.end(); it++)
        {
            double probability=it->second/TimeCurrent(); // P(i)

            if(probability < 0.0 || probability > 1.0) {
                THROW_DEVS_EXCEPTION("Invalid Probability!");
            }
            else{
                int i = atoi(it->first.data());
                E_i += probability * i;// E(i)=\Sum_{i} i * P(i)
            }
        }
        string str = "Average Q length: ";
        statistics.insert(make_pair(str, E_i));
    }
    return statistics;
}
```

It makes $P(C=\texttt{i})$ using `m_statistics[i]`. Then it makes $E(C)$ by summing over `i`$*P(\texttt{i})$ for all `i` as defined in Equation (4.10).

4.3 Client-Server System

The example `Ex_ClientServer` shows all features of performance measurement introduced in this chapter. This example considers a configuration of n servers where n can vary from 1 to 5. Figure 4.7 illustrates the case of $n = 3$.

The entire simulation model consists of the client-server system under test, named `CS`, and the experimental frame, named `EF`, as shown in Figure 4.7. The sub-components of `EF`, `Generator` and `Transducer` were investigated in the previous section, so we will discuss the sub-models of `CS` in the following sections.

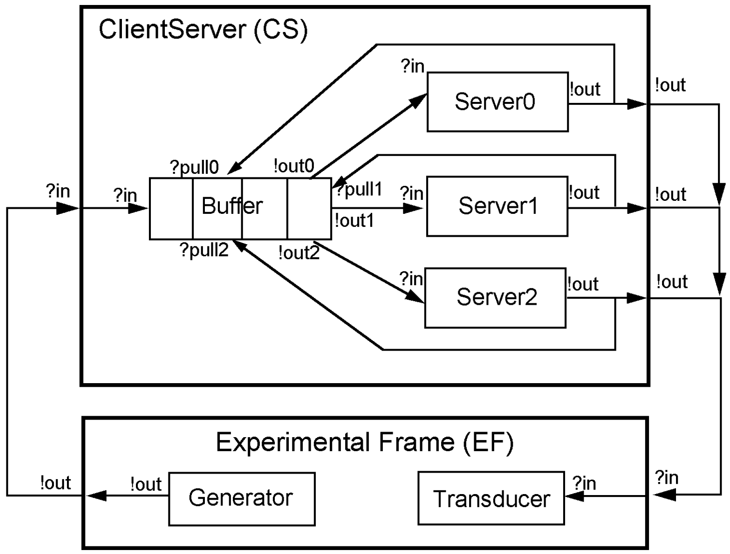

Figure 4.7: Configuration of Client Server System $n = 3$

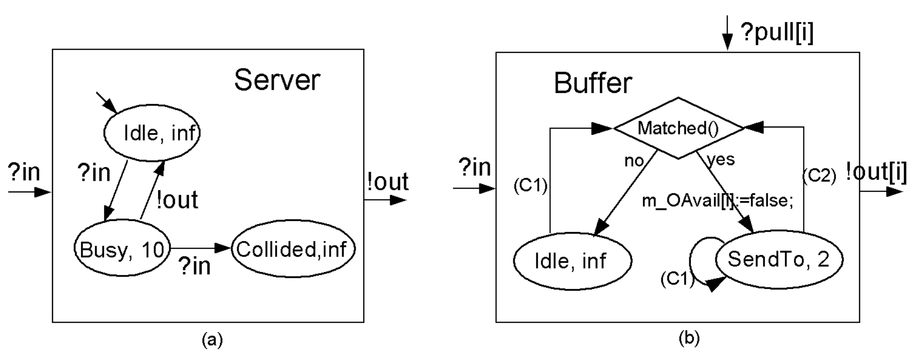

Figure 4.8: Server and Buffer

4.3.1 Server

`Server` is a concrete class derived from `Atomic`. The state transition diagram of `Server` can be drawn as shown in Figure 4.8(a). The C++ codes of `Server` available in `Server.h` and are represented by Figure 4.8(a) there is no need for further explanation here.

4.3.2 Buffer

`Buffer` is a concrete class derived from `Atomic`. This class has a single input port `in`, an n-vector of input ports `pull` and an n- vector of output ports `out` (in this example, n=3). As member data, `phase` is a string; `m_Clients` is a buffer keeping incoming clients whose type is `tmValue`; `m_OAvail` is a vector of boolean values tracking the availability of servers; `m_OSzie` stores the number of connected servers; and `send_index` is an `int` which tracks the server index to which `Buffer` will send output.

```
class Buffer: public Atomic {
public:
    InputPort* in;
    vector<InputPort*> pull;
    vector<OutputPort*> out;

protected:
    string m_phase;
    deque<tmValue*> m_Clients;
    vector<bool> m_OAvail;
    const int m_Osize;
    int send_index;
```

The function `C1` updates member data as a function of an input event `x`. If `x` comes through the input port `in`, `C1` casts the value of `x` to `tmValue` and pushes it back to the buffer `m_Clients`. Otherwise, `x` comes through one of `pull` ports. So `C1` searches the server index `i`, checking the identity of `pull[i]` and the incoming event's port, and updates `m_OAvail[i]=true` which marks the `i`-th server as being available.

```
void C1(const PortValue& x)
{
    if(x.port == in){ //receiving a client
        tmValue* client = dynamic_cast<tmValue*>(x.value);
        if(client) {
            m_Clients.push_back(client);
        } else
```

```
            THROW_DEVS_EXCEPTION("Dynamic Casting Failed!");
    }
    else // receiving a pull signal
    {
        for(int i=0; i<m_Osize; i++) {
            if(x.port == pull[i]) {
                m_OAvail[i]= true; //  server_i is available
                break;
            }
        }
    }
}
```

The function `Matched()` first checks to see if there is a waiting client in `m_Clients` and then checks to see if there exists an available server from 0 to `m_Osize`-1. If a match is found, the function sets `m_OAvail[i]=false`, remembers the index `i` at `send_index`, then returns `true`. Otherwise it returns `false` which means no match.

```
bool Matched()
{
    if(m_Clients.empty() == false){
        for(int i=0; i < m_Osize; i++){// select server
            if(m_OAvail[i] == true){// server i is available
                m_OAvail[i]=false;//Mark server_i non-available
                send_index = i; // remember i in send_index
                return true;
            }
        }
        return false;
    }else
        return false;
}
```

The function `C2` creates an the output event and removes the first client from `m_Clients` when `C2`'s phase is `SENDTO`.

```
void C2(PortValue& y) {
    if(m_phase == SENDTO){
        y.Set(out[send_index], m_Clients[0]);
        m_Clients.pop_front();// remove the first client
    }
}
```

The function init() of Buffer resets phase to IDLE, assigns m_OAvial[i]=true for all indices, and clears all clients in m_Clients.

```
/*virtual*/ void init()
{
    m_phase = IDLE;
    m_OAvail.clear(); // clear first
    for(int i=0; i<m_Osize; i++)
        m_OAvail.push_back(true); // add variable

    while(m_Clients.empty() == false)
    {
        tmValue* cl = m_Clients[0];
        m_Clients.pop_front();
        delete cl;
    }
}
```

Buffer's tau() returns ∞ for IDLE and returns 2.0 for SENDTO.

```
/*virtual*/ Time tau() const {
    if(m_phase == IDLE)
        return DBL_MAX;
    else
        return 2.0;
}
```

The input transition function delta_x of Buffer updates member data by calling C1(x) and then, if the phase of the server is IDLE, checks the returning value of Matched(). If the value is true, the phase of the server changes into SENDTO.

```
/*virtual*/ bool delta_x(const PortValue& x)
{
    C1(x);
    if(m_phase == IDLE){
        if(Matched()){
            m_phase = SENDTO;
            return true; // reschedule as active
        }
    }
    return false;
}
```

When the server is ready to exit the SENDTO state, it gets y by calling C2(y), if Matched() returns true, the phase stays at SENDTO. Otherwise, the phase returns to IDLE.

```
/*virtual*/ void delta_y(PortValue& y) {
    C2(y);
    if(Matched())
        m_phase = SENDTO;
    else
        m_phase = IDLE;
}
```

Recall that `Buffer` class contains the overriding `Get_Statistics_s()` and `GetPerformance()`, which were investigated in Section 4.2.3. For the codes of `Buffer::Get_s()`, the reader should refer to `Buffer.h`.

4.3.3 Performance Analysis

The procedure for constructing the coupled model `EF` and `CS` is omitted here because it is quite straight forward and its schematics were shown in Figure 4.7.

We will analyze change of performance indices by varying the number of servers. The number of servers used in `CS` can be varied by passing different numbers n with the following API, where n is the number of servers desired.

```
Coupled* MakeTotalClientServerSystem(int n);
```

The simulation settings we use here are: the simulation ending time=10000; no display of continuously increasing t_e, the scale factor is maximum, in which the clock jumps to the next event time; and there is no display of discrete event transitions. The following code shows the case where the number of servers is 5.

```
void main( void ) {
    Coupled* Sys = MakeTotalClientServerSystem(5);// n=5
    Sys->PrintCouplings();

    SRTEngine simEngine(*Sys, 10000); //
    simEngine.SetAnimationFlag(false);
    simEngine.SetTimeScale(DBL_MAX); //
    simEngine.Set_dtmode(SRTEngine::P_NONE);
    simEngine.RunConsoleMenu();
    delete Sys;
}
```

Let's change n sequentially from 1 to 5, and build the various system models, and try `mrun 20` for each configuration. After completion of `mrun 20`, DEVS++

Table 4.1: Performance Indices for each $n = i$ of Servers

Performance Indices	n=1	n=2	n=3	n=4	n=5
Queue Length	589.00	173.79	1.65	0.71	0.58
System Time	2,927.33	873.86	18.30	13.54	12.86
Throughput	0.08	0.17	0.20	0.20	0.20
Utilization	0.83	0.83	0.67	0.50	0.40

Utilization is measured by the average utilization of all servers for $2 \leq n$.
For example, Utilization when n=3 means $\sum_{i=1,2,3}$Utilization(i)/3.

summarizes the performance indices to the console. [2] Table 4.1 shows performance indices for each configuration and Figure 4.9s show the trend of performance changes as n changes.

Average Queue Length and Average System Time are drastically reduced until n reaches 3. Average Throughput increase up to 0.2 jobs/time-unit at n=3 and then there is no increase at n=4 and 5. The reason why Throughput doesn't increase after n=3 might be that there is lack of client arrival from outside the system. We can find a similar phenomenon in Utilization which doesn't decrease when n=2 but starts to decrease when n=3.

Another interesting trend is that both utilizations at n=1 and $n = 2$ are equal to about 80%, not 100%, even though Average Queue Length is 589 and 173 and Average System Time is 2,927.33 and 873.86 time-units, respectively. The reason seems to be caused by `Buffer::tau(SENDTO)`=2. `Server`'s $P(C =$`Idle`$)$ is about 0.2, which makes sense when considering `Server::tau(Busy)`=10. In other words, except for the client transmission time from `Buffer` to `Server`, `Server` keeps working all the time.

The following screen shot illustrates the average value and its 95% confidence interval for each statistical item listed where the number of servers is 5. We can find uneven utilizations in this screen shot. For example, $P(C =$`Busy`$)$=0.61 for `SV0` server, while $P(C =$`Busy`$)$=0.17 for `SV4` server. This phenomenon is caused by the searching order in the `Buffer::Matched()` function in which checking for the availability of servers starts from 0 index all the time. We may need to modify the searching order if we want to utilize the servers more evenly.

Note that in order to have a confidence interval for mu, you must have run a large number of simulations.[Zei76, LK91] It would help the analyst to know how many simulations were run to produce these results.

. . .

[2] The log file "devspp_log.txt" collects also the same performance indices. But watch out that the old devspp_log.txt will be over written by the new one every time we execute DEVS++.

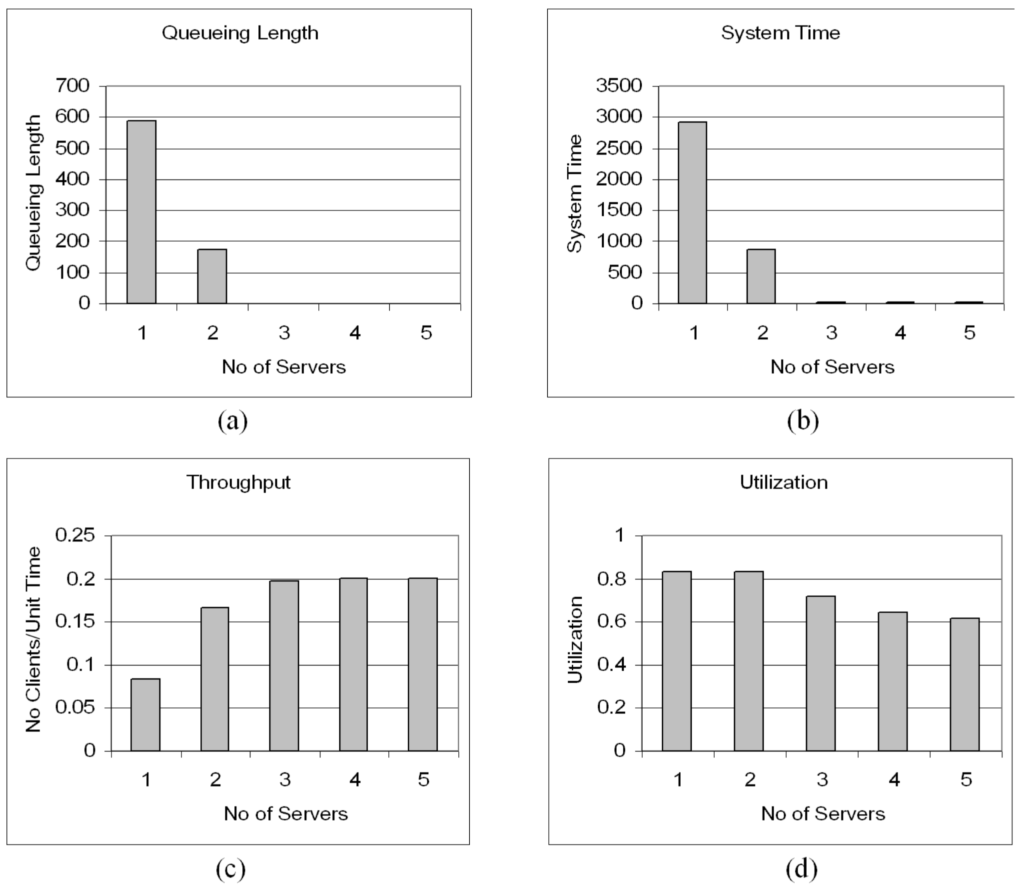

Figure 4.9: Performance Indices

```
============= Total Performance Indices =========
CSsystem.CS.BF
Average Q length: : 0.575612, 95% CI: [0.560994, 0.590231]

CSsystem.CS.SV0
Busy: 0.61411, 95% CI: [0.611001, 0.617219]
Idle: 0.38589, 95% CI: [0.382781, 0.388999]

CSsystem.CS.SV1
Busy: 0.525135, 95% CI: [0.520404, 0.529867]
Idle: 0.474865, 95% CI: [0.470133, 0.479596]

CSsystem.CS.SV2
Busy: 0.413587, 95% CI: [0.408219, 0.418955]
Idle: 0.586413, 95% CI: [0.581045, 0.591781]

CSsystem.CS.SV3
Busy: 0.28686, 95% CI: [0.279344, 0.294376]
Idle: 0.71314, 95% CI: [0.705624, 0.720656]

CSsystem.CS.SV4
Busy: 0.16772, 95% CI: [0.159812, 0.175628]
Idle: 0.83228, 95% CI: [0.824372, 0.840188]

CSsystem.EF.Trans
Average System Time: 12.8643, 95% CI: [12.8222, 12.9064]
Throughput: 0.20068, 95% CI: [0.198181, 0.203179]

========== Simulation Run Completed! ==========
```

Appendix A

Building DEVS++

The directory structure of DEVS++ verion 1.4.2 is as follows.

```
+-DEVSpp
    +- Doc
    +- Examples
        ...
```

This appendix covers the three folders: `DEVSpp`, `DEVSpp/Doc`, and `DEVSpp/Examples`. `DEVSpp` contains header files and cpp source files of DEVS++ as well as the project file and the solution file of Microsoft Visual Studio. `DEVSpp/Doc` contains this document file. `DEVSpp/Examples` includes example folders: `Ex_ClientServer`, `Ex_Monorail`, `Ex_PingPong`, `Ex_Template`, `Ex_Timer`, `Ex_VendingMachine`.

All of examples are addressed in this document except `Ex_Template` which provides a template whose settings can be used as the starting point for the reader's own project (using copying and modifying). The source code used in `Ex_Template` is the same as in `Ex_PingPong`.

As of May 3, 2009, we had tested the compilation of DEVS++ only in Microsoft Visual Studio(MVS) 2005™.

A.1 Using Microsoft Visual Studio 2005™

If you open the solution file of `DEVSpp/DEVSpp.sln`, Visual Studio 2005™ opens the associated project files including `DEVSpp.vcproj` as well as those of the example projects as shown in Figure A.1.

You can open each example solution individually. For example, if you open `DEVSpp/Examples/Ex_PingPong/Ex_PingPong.sln` file, you can see that only `DEVSpp` project and `Ex_PingPong` project are opened in Solution Explorer win-

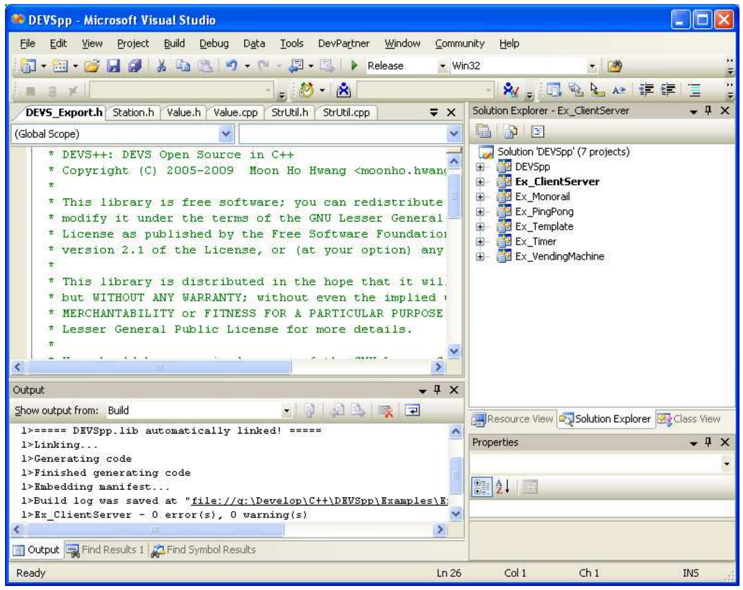

Figure A.1: Screen Capture of Visual Studio 2005™ when opening DEVSpp/DEVSpp.sln

dow of Visual Studio 2005™. To run each example, we should build DEVSpp first and then build the example project.

In order to run the examples provided in `DEVSpp/Examples` folder, we don't have to change the compile and build options at all. But if you want to know the settings inside, the following information will be useful.

There are two different ways to build DEVSpp library in verion 1.4.2: "Debug" & "Release". Each configuration will create its own folder, and there will be `DEVSpp.dll` and `DEVSpp.lib`

The special settings of `Configuration Properties` for DEVS++ are:

1. General/Configuration Type: Dynamic Library (.dll)

2. C/C++

    - Preprocessor/Preprocessor Definitions:
      `WIN32;DEVSpp_EXPORT;` for all configurations
      `_DEBUG;` for Debug configurations.
      `NDEBUG;` for Release configurations.

I believe that the reader will be unlikely to change the `DEVSpp.vcproj` file. But the reader could make her or his own examples. The special settings of `Configuration Properties` for `Ex_*` examples are:

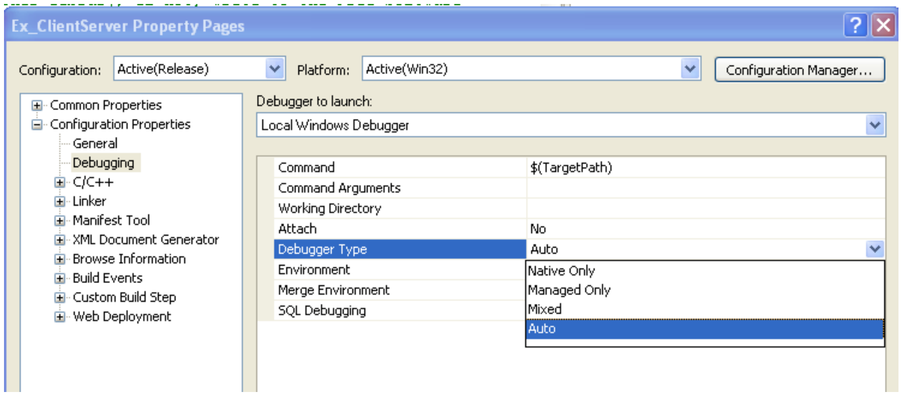

Figure A.2: Change Debugger Type

1. General/Configuration Type: Application (.exe) .
2. C/C++
   - General/Additional Include Directories: `../../../DEVSpp`
   - Preprocessor/Preprocessor Definitions:
     `WIN32;` for all configurations.
     `_DEBUG;` for Debug configurations.
     `NDEBUG;` for Release configurations.
3. Linker/General/Additional Library Directories:
   `$(SolutionDir)$(ConfigurationName)` for all configurations.

A.1.1 When debugging through breakpoints in DEVS++ is failed

When I used MSV 2005, I found that I could not get into breakpoints at source codes DEVS++ sometimes. We may be able to search the internet for how to resolve this situations. What I found so far is that the way to build debugging information from C++ codes for MSV seems little bit unstable.

To make MSV behave correctly, one tip I usually use is to change debugger type between "Auto" and "Managed Only" in the property dialog as shown in Figure A.2.

Another tip I would like to give readers is that when you make your own project (or solution), you should make sure that *project dependency* of your own project has a dependency on DEVSpp. That can be done by "Project-¿Project Dependency..." menu item.

Appendix B

History and Plan

B.1 Revision History

B.1.1 Version 1.4.2

Library

1. Simplified build options as two: Debug (dll) and Release (dll)

2. Assumed that the include path is DEVSpp directory.

3. Changed `SRTEngine`'s `dtmode` command to show `rel` and `abs`.

Manual

1. Added the definition of Deterministic and Nondeterministic DEVSs in Section 1.1.

2. Added Appendix History and Plan.

3. Added indices.

B.1.2 Version 1.4.1

Library

1. Supported four different build options: debug_dll, debug_static, release_dll, release_static

2. Assumed that the include path is the parent of DEVSpp directory.

Manual

1. Released the first manual for DEVS++.
2. Contents: Chapter 1. Introduction to DEVS; Chapter 2. Library Structure; Chapter 3. Simple Examples; Chapter 4. Performance Evaluation; Appendix A. Building DEVS++;

B.2 Plan

B.2.1 Short Term

1. Supporting `gcc++` build.
2. Changing realtime advance mechanism of `SRTEngine`.
3. Supporting a non-thread engine.
4. Supporting multiple output events.

B.2.2 Mid Term

1. Supporting variable structuring DEVS.
2. Supporting distributed simulation.

B.2.3 Long Term

1. Supporting reachability-based verification engine.
2. Supporting animation and visualization.

Bibliography

[DEV08a] *Behavior of Atomic DEVS*. http://en.wikipedia.org/wiki/Behavior_of_DEVS, 2008.

[DEV08b] *Behavior of Coupled DEVS*. http://en.wikipedia.org/wiki/Behavior_of_Coupled_DEVS, 2008.

[Kim94] T.G. Kim. *DEVSim++ User's Manual:C++ Based Simulation with Hierarchical, Modular DEVS Models*. http://smslab.kaist.ac.kr/DES/DES.htm, Systems Modeling Simulation Lab., KAIST, Taejon, Korea, 1994.

[Lew98] Robert W. Lewis. *Programming Industrial Control System Using IEC 1131-3*. The Institution of Electrical Engineers, revised edition, 1998.

[LK91] Averill M. Law and W. David Kelton. *Simulation Modeling and Analysis*. McGraw-Hill, New York, second edition, 1991.

[Nut00] Jim Nutaro. ADEVS: C++ Library of Parallel DEVS. http://www.ornl.gov/~1qn/adevs/index.html, 2000.

[Ska96] K. Skahill. *VHDL for Programmable Logic*. Prentice Hall, 1996.

[Zei76] Bernard P. Zeigler. *Theory of Modelling and Simulation*. Wiley Interscience, New York, first edition, 1976.

[Zei90] Bernard P. Zeigler. *Object-oriented Simulation with Hierarchical, Modular Models:Intelligent Agents and Endomorphic Systems*. Academic Press, Boston, Mass. and San Diego, Calif., second edition, 1990.

[ZPK00] B. P. Zeigler, H. Praehofer, and T. G. Kim. *Theory of Modelling and Simulation: Integrating Discrete Event and Continuous Complex Dynamic Systems*. Academic Press, London, second edition, 2000.

Index