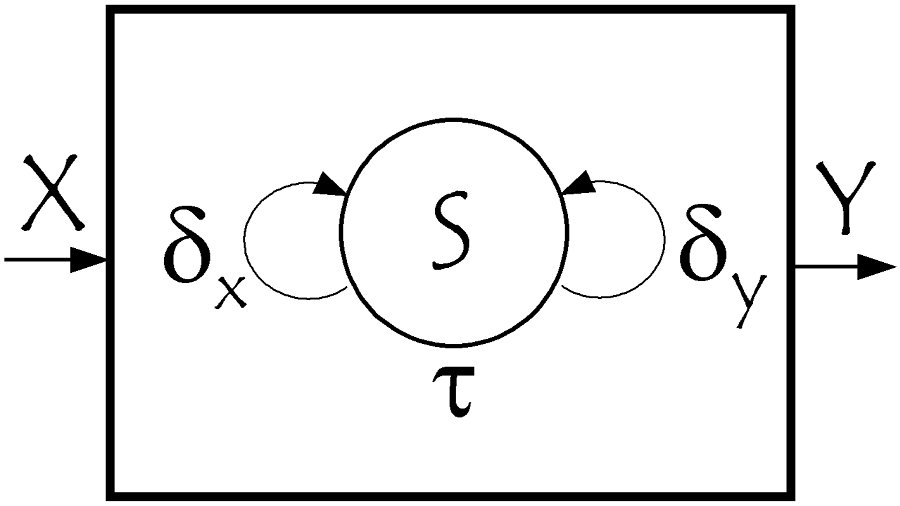

Modeling and Simulation using DEVS#

Moon Ho Hwang

DEVS# version 1.2.1

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

```
@MANUAL{DEVSsharp,
  TITLE =        "{Modeling and Simulation using DEVS\#}",
  author =       "Moon Ho Hwang",
  address =      "http://xsy-csharp.sourceforge.net/DEVSsharp",
  edition =      "first",
  month =        "May",
  year =         "2007",
}
```

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 2005 when I tried to make DEVS# which is an another open source library of DEVS formalism in C++, I had a chance to use C# language in a project. During the project, I realized that C# has some advantages over C++ such as garbage collection, type checking functionality, Web functionality, etc. Then, I compared the execution speeds of these two languages. Surprisingly, C# is not slower than C++ (frankly speaking, C# was little bit faster than C++ in my test case). After the speed testing, I got started to implement a DEVS open library in C# through the sourceforge.net in 2006. Finally, I could open DEVS# library at http://xsy-csharp.sourceforge.net/DEVSsharp.

Although the main objective of developing DEVS# is to provide not only a modeling and simulation environment but also a modeling and verification software based-on DEVS theory, this document would focus on the first functionality: modeling and simulation. However, since this document is not a C# programming book, this book doesn't cover the syntax of C# and how to use Visual Studio developing environment in depth. Thus, I would recommend you to read introductory book of C# first if the reader is not familiar with C# language.

This document consists as follows.

Chapter 1 provides a belief review of DEVS formalism including a verbal description of DEVS behavior. 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 or the abstract 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 a practical example.

As an appendix, Chapter 5 briefly covers the structure of DEVS# library, how to compile examples which are provided in DEVS#, and how to add our own project or solution using Visual Studio 2005. If you want to compile, build, and run the examples first, you'd better read this chapter first.

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 of DEVS++ [Hwa07], corrected some of my not so excellent English expressions, and suggested some interesting systems engineering ideas. Without his devoted help, [Hwa07] that provides the foundation of this book, could never have been completed.

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 I got started to implement DEVS#.

Tucson, Arizona
May, 2007

Moon Ho Hwang

Contents

Preface iii

1 DEVS Formalism and DEVS# code 1
- 1.1 Atomic DEVS . . . 1
- 1.2 Coupled DEVS . . . 4
- 1.3 Building Ping-Pong Game using DEVS# . . . 5

2 Structure of DEVS# 11
- 2.1 Event=PortValue . . . 12
  - 2.1.1 Named . . . 12
  - 2.1.2 Port, InputPort, and OutputPort . . . 12
  - 2.1.3 PortValue . . . 13
- 2.2 DEVS . . . 14
  - 2.2.1 Base DEVS: Devs . . . 14
  - 2.2.2 Atomic DEVS: Atomic . . . 15
  - 2.2.3 Coupled DEVS: Coupled . . . 17
- 2.3 Scalable Real-Time Engine: SRTEngine . . . 18
  - 2.3.1 Constructor . . . 18
  - 2.3.2 Run console menu . . . 18
- 2.4 Random Variables . . . 23
  - 2.4.1 Probability Density Functions . . . 23
  - 2.4.2 Probability Mass Function . . . 24

3 Simple Examples 27
- 3.1 Atomic DEVS Examples . . . 27
  - 3.1.1 Timer . . . 27
  - 3.1.2 Vending Machine . . . 29
- 3.2 Coupled DEVS Examples . . . 34
  - 3.2.1 Monorail System . . . 34

4 Performance Evaluation **43**
4.1 Performance Measures 43
4.1.1 Throughput 43
4.1.2 Cycle Time 44
4.1.3 Utilization 44
4.1.4 Average Queue Length 46
4.1.5 Sample Mean, Sample Variance, and Confidence Interval . 47
4.2 Practice in DEVS# 49
4.2.1 Throughput and System Time in DEVS# 49
4.2.2 Utilization in DEVS# 54
4.2.3 Average Queue Length in DEVS# 56
4.3 Client-Server System 57
4.3.1 Server 57
4.3.2 Buffer 57
4.3.3 Performance Analysis 62

5 Appendix: Building Projects using DEVS# **67**
5.1 Directory Structure of DEVS# 67
5.2 Building Simulation Examples in DEVS# 68
5.3 Adding Our Own Project 68

List of Figures

1.1 Symmetric Structure of Atomic DEVS 2
1.2 State Transition Diagram of Ping-Pong Player 3
1.3 DEVS Model of Ping-Pong Game 5
1.4 References of Ex_PingPong 7

2.1 Classes in DEVS# 11
2.2 Relations of Times 15
2.3 PMF $f(s)$ and its cumulative function $F(x)$ 25

3.1 Timer (a) State Transition Diagram (b) Event Segment (c) t_e Trajectory 28
3.2 State Transition Diagram of Vending Machine 30
3.3 Monorail System 34
3.4 Phase Transition Diagram of Station 35

4.1 A System having a Buffer and a Processor 44
4.2 State Trajectory of a Processor 45
4.3 A State Trajectory of Vending Machine 45
4.4 Trajectory of Queue 47
4.5 IID random variants $X_1 \dots X_n$ from n simulation runs 48
4.6 State Transition Diagrams of Generator: (a) Autonomous Mode (b) Non-Autonomous Mode 50
4.7 State Transition Diagrams of Transducer 53
4.8 Configuration of Client Server System $n = 3$ 58
4.9 Server and Buffer 58
4.10 Performance Indices 64

5.1 Directory Structure of DEVS# 67
5.2 Screen Capture of Visual Studio 2005^{TM} when opening DEVSsharp.sln 69
5.3 New Project Dialog 70

5.4 My Project 71
5.5 Menu Selection of Add Existing Project... 72
5.6 Menu Selection of Add Reference... 72
5.7 Dialog of Add Reference 73

List of Tables

4.1 Performance Indices for each $n = i$ of Servers 63

Chapter 1

DEVS Formalism and DEVS# code

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 *states*.
- $s_0 \in S$ is the initial state.
- $\tau : S \rightarrow \mathbb{T} \cup \{\infty\}$ is the *time advance function* where $\mathbb{T} = [0, \infty)$ is the set of non-negative real numbers. This function is used to determine the lifespan of a state.
- $\delta_x : P \times X \rightarrow S \times \{0, 1\}$ is the *input transition function* where

$$P = \{(s, t_s, t_e)|s \in S, t_s \in \mathbb{T} \cup \{\infty\}, t_e \in \mathbb{T}\}$$

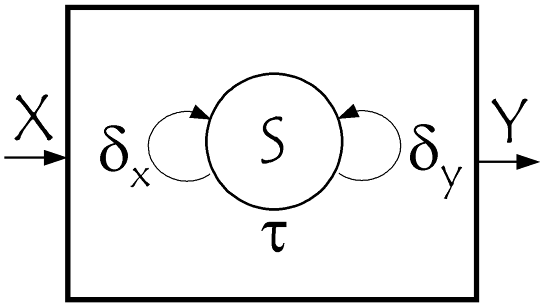

Figure 1.1: Symmetric Structure of Atomic DEVS

is the set of *states with times* such that t_s and t_e are the *lifespan* of the state, s, and the *elapsed time* since the last reset of t_e, respectively. δ_x defines how an input event, x, changes a state as well as the lifespan that the system can be in that state and the elapsed time that the system has been in that state.

- $\delta_y : S \to Y^{\phi} \times S$ [1] is the *output 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. [2] ■

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 input transition function (δ_x) are on the input side; the output event set (Y) and the output 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.

Verbal Description of Dynamics

Suppose that A is an atomic DEVS such that $A =< X, Y, S, s_0, \tau, \delta_x, \delta_y >$, s is the current state of A, and $p = (s, t_s, t_e) \in P$. Then the possible discrete state transitions are:

1. **If an external input** x **comes in**, A executes $\delta_x(p, x) = (s', b)$ where $b \in \{0, 1\}$ and the lifespan and the elapsed time with s' can change or be preserved as follows.

   (a) update $t_s = \tau(s')$ and $t_e = 0$ if $b = 1$;

[1] where $Y^{\phi} = Y \cup \{\phi\}, \phi \notin Y$ is the silent event

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

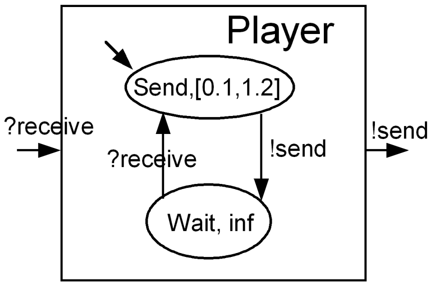

Figure 1.2: State Transition Diagram of Ping-Pong Player

(b) keep t_s and t_e preserved if $b = 0$.

2. **If no external input comes in,** then when the elapsed time reaches the lifetime, A executes $\delta_y(s) = (y, s')$ and update $t_s = \tau(s')$ and $t_e = 0$.

Notice that the elapsed time t_e increases linearly over time so it is a continuous variable whose time derivative is constant 1. However, the lifetime t_s is not changing continuously but it is determined discretely at the time of executing either δ_x or δ_y.

In other word, suppose that there is $p = (s, t_s, t_e) \in P$ at time $t_l \in T$, there is another $p' = (s', t'_s, t'_e) \in P$ at time $t_u \in T$ and $t_l \leq t_u$. If there is no event between t_l and t_u, it implies that the only difference between p and p' is that their elapsed time such that $(s' = s) \wedge (t'_s = t_s) \wedge (t'_e = t_e + t_u - t_l)$.

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 an 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`); □

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*. ■

Verbal Description of Coupled DEVS Behavior

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

1. When N receives an input event, the coupled DEVS transmits the input event to the sub-components through the set of external input couplings.
2. When a sub-component produces its output event, 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.

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 = { }. □

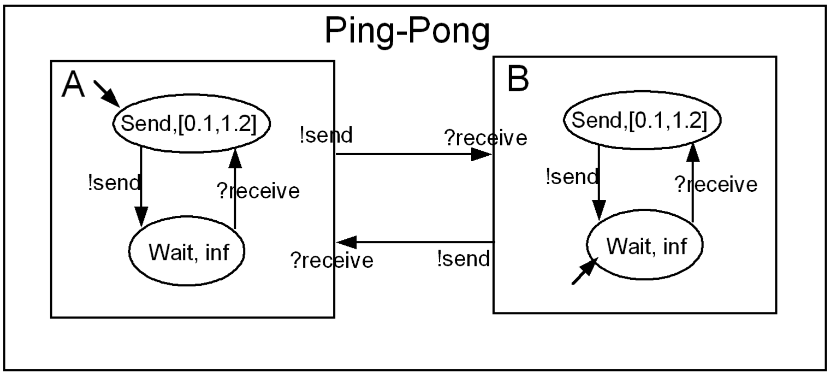

Figure 1.3: DEVS Model of Ping-Pong Game

1.3 Building Ping-Pong Game using DEVS#

This section shows *what DEVS# code looks like* using the ping-pong game introduced in Example 1.2. When you open `DEVSsharp\DEVSsharp.sln`, you can find `Ex_PingPong` project in which there are two source files: `Player.cs` and `Program_PingPong.cs`. Let's take a look at `Player.cs` first.

```
using DEVSsharp; //--- (1)

namespace Ex_PingPong
{
    public class Player : Atomic //-- (2)
    {
        public InputPort receive; //-- (3)
        public OutputPort send;   //-- (3)
        enum PHASE { Wait, Send}
        PHASE m_phase;         //-- (4)
        bool m_width_ball;     //-- (4)
        RVofGeneralPDF rv;     //-- (4)

        public Player(string name, bool with_ball): base(name, TimeUnit.Sec)
        {
            receive = AddIP("receive");  //--(5)
            send = AddOP("send");        //--(5)
            m_width_ball = with_ball;
            rv = new RVofGeneralPDF(); //-- (6)
            init();
        }
```

```
            public override void init() {  //-- (7)
                if (m_width_ball)
                    m_phase = PHASE.Send;
                else
                    m_phase = PHASE.Wait;
            }
            public override double tau()  //-- (8.a)
            {
                if (m_phase == PHASE.Send)
                    return rv.Uniform(0.1, 1.2);
                else
                    return double.MaxValue;
            }
            public override bool delta_x(PortValue x)  //-- (8.b)
            {
                if (m_phase == PHASE.Wait && x.port == receive)
                {
                    m_phase = PHASE.Send;
                    return true;
                }
                else
                {
                    Console.WriteLine("Do we have more than one ball?");
                }
                return false;
            }
            public override void delta_y(ref PortValue y) //-- (8.c)
            {
                if (m_phase == PHASE.Send)
                {
                    y.Set(send);
                    m_phase = PHASE.Wait;
                }
            }

            public override string Get_s() //-- (9)
            {
                return m_phase.ToString();
            }
        }
    }
```

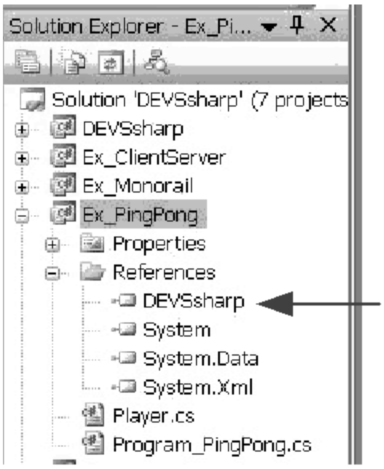

Figure 1.4: References of Ex_PingPong

1. **using DEVSsharp:** First of all, we can find `Ex_PingPong` project uses a reference of `DEVSsharp` project which is indicated in Solution Explorer windows of Visual Studio 2005 as shown in Figure 1.4. [3] By `using DEVSsharp`, we can load information of name space, classes interfaces defined in `DEVSsharp` that is the kernel project name of DEVS#.

2. **Deriving 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.

3. **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

4. **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.

5. **Adding Interfaces:** 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.

6. **Random Variable:** The lifespan of `Send` is a random variable with uniform probability density function (PDF) of [0.1,1.2]. To generate the ran-

[3] For more information of adding a reference, you can refer to Chapter 5.

dom number, `Player` defines a random variable `rv` as a general PDF random variable in (4), and pick the uniform PDF in the range[0.1, 1.2] in `tau()` function in (8.a). The PDFs available in DEVS# are addressed in Section 2.4.

7. **The initial State s_0:** To make the initial state s_0, all concrete classes derived from `Atomic` are supposed to override the function `init()` in which the associated atomic model is reset to the initial state s_0. In this case of `Player`, the initial phase can be determined as `Send` or `Wait` depending on another variable, `m_width_ball` which is indicating to have a ball initially or not.

8. **Characteristic Functions τ, δ_x and δ_y:** all concrete classes derived from `Atomic` should override the characteristic functions: τ, δ_x, and δ_y.

   (a) τ of `Player` returns the random number from [0.1, 1.2] when the state is `Send`, otherwise it returns ∞ that is represented by `double.MaxValue`.

   (b) δ_x of `Player` changes the state `Wait` to `Send` when receiving the input event `receive`.

   (c) δ_y of `Player` generates the output event `send`, at the same time, it changes the state `Send` to `Wait`.

9. **Displaying Status:** To show the current state, we will override the `Get_s()` function which is supposed to return a string representing the current state. `Player` returns the string value of `m_phase` variable.

A ping-pong match we are considering here needs two players that are instances of the previous class `Player`. We use the coupled DEVS in `Program_PingPong.cs` to model the match as shown in the following codes.

```
using DEVSsharp;

namespace Ex_PingPong
{
    class Program_VM
    {
        static Devs MakePingPong(string name)
        {
            Coupled game = new Coupled(name); //-- (1)
            Player A = new Player("A", true); //-- (2)
            Player B = new Player("B", false);//-- (2)

            game.AddModel(A); //-- (3)
```

```
            game.AddModel(B); //-- (3)
            game.AddCP(A.send, B.receive); //-- (4)
            game.AddCP(B.send, A.receive); //-- (4)
            game.PrintCouplings(); //-- (5)
            return game;
        }

        static void Main(string[] args)
        {
            Devs md = MakePingPong("PingPong");
            SRTEngine Engine = new SRTEngine(md, 10000, null);//--(6)
            Engine.RunConsoleMenu();//--(7)
        }
    }
}
```

1. **Making the Ping-Pong Game as coupled DEVS:** We make an instance of `Coupled` in DEVS# for the ping-pong game.
2. **Instancing Two Players** The ping-pong game has two sub-components that are instances of `Player` having different initial states.
3. **Adding Components** We add two players `A` and `B` by calling the function `AddModel` of the class `Coupled`.
4. **Adding Couplings** We add couplings between players `A` and `B` calling the function `AddCP` of the class `Coupled` .
5. **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-----
```

6. **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, `pp`. For more detailed information of SRTEngine, the reader can refer to Section 2.3.

7. **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 Formalism, (C) 2005~2007,
http://xsy-csharp.sourceforge.net/DEVSsharp/

The current date is 5/6/2007.
The current time is 1:05:52 PM.
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 date 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 `PortValue` that 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 any derived class of the basic class `object` of C#. `SRTEngine` is a scalable real-time engine which runs a DEVS instance inside.

In Figure 2.1, a *gray* box indicates a concrete class which can be created as

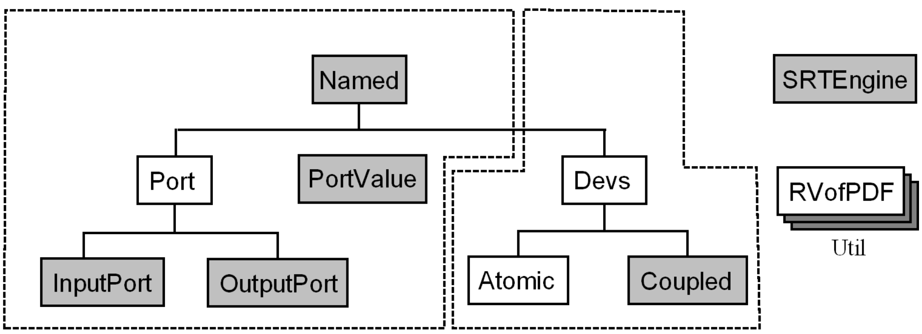

Figure 2.1: Classes in DEVS#

an instance, while a *white* box is an abstract class which can not be created as an instance.

We will first go through `PortValue` 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 classes 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 finally, the `PortValue` class will be seen in the last part of this section.

2.1.1 Named

`Named` is defined in `Named.cs` file as a concrete class. The class provides its constructor whose argument is a string, and has a public `Name` field as a string. The function `ToString()` is the function overrided from `object::ToString()`.

```
public class Named
{
    public String Name;

    public Named(string name)
    {
        m_Name = name;
    }

    public override string ToString()
    {
        return m_name;
    }
}
```

2.1.2 Port, InputPort, and OutputPort

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

```
class Port: public Named
{
```

```
    public Devs Parent;
    protected List<Port> m_FromP,  // From Ports
                         m_ToP; // To Ports
    public List<Port> FromP { get { return m_FromP; } }
    public List<Port> ToP { get { return m_ToP; } }
};

class InputPort: public Port {
    ...
};

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

`Port` is an abstract class derived from `Named`. It has `Parent` field whose type is `Devs`, and which is automatically assigned when we call the `AddIP()` and `AddOP()` functions of `Devs` (see Section 2.2). `Port` has "`List<Port> ToP`" as a set of successors as well as "`List<Port> FromP`" as a set of predecessors which are changed when we call `AddCP()` and `RemoveCP()` of `Coupled` (see Section 2.2.3).

`InputPort` and `OutputPort` are concrete and derived classes from `Port`.

2.1.3 PortValue

As mentioned before, an event in DEVS# is modeled by `PortValue` class that have a pair of a deriving class of `Port` and a deriving class of `object`. The following codes are parts of `PortValue.cs`.

```
class PortValue {
public:
   public Port port; //-- either an output or an input port
   public object value; // deriving class from the Object class

   public PortValue(Port prt){...}
   public PortValue(Port prt, object v) {...}
   public void Set(Port prt){...}
   public void Set(Port prt, object v){...}
   public override string ToString(){...}
 };
```

Two constructors and two `Set` functions are available whose arguments can be `Port p` which means value `v=null`, or a pair of (`Port p` and `object v`). The function ToString() returns the string concatenating of port and value (if value is not null) by using a delimiter ':' character.

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 the source file Devs.cs is an abstract class derived from Named. Devs points its parent through its Parent field which is assigned by Coupled::AddModel() (see Section 2.2.3).

```
public class Devs: 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. In addition, IP and OP get the set of input ports as SortedList<string, InputPort> and the set of output ports as SortedList<string, OutputPort>, respectively.

```
    //-- X port Methods --
    InputPort AddIP(string ipn);
    InputPort GetIP(string ipn) const;
    InputPort RemoveIP(string ipn);
    void PrintAllIPs() ;
    public SortedList<string, InputPort> IP {get ; }

    //-- Y port Methods --
    OutputPort AddOP(string opn);
    OutputPort GetOP(string opn) const;
    OutputPort RemoveOP(string opn);
    void PrintAllOPs() ;
    public SortedList<string, OutputPort> OP {get ; }
```

Devs has time-related properties such as TimeLast for getting (or setting) the last schedule update time; TimeNext for the next schedule time; TimeElapsed

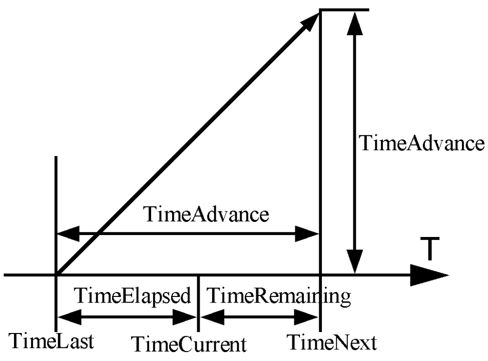

Figure 2.2: Relations of Times

for the elapsed time between the last schedule time and the current time; `TimeRemaining` for the remaining time from the current time to the next schedule time; `TimeAdvance` for the time difference between `TimeNext` and `TimeLast`.

`TimeCurrent` is a static and public data field for the current time;

```
public double TimeLast { get; set; }
public double TimeNext { get; set; }

public double TimeElapsed  { get; }
public double TimeRemaining  { get; }
public double TimeAdvance  { get; }

public static double TimeCurrent ; // data field
```

Figure 2.2 illustrates the relationships among different times. The user doesn't have to set the values of these time variables because such calculation will be performed by the simulation engine class, called `SRTEngine` in DEVS#, according to the user-defined $\tau(s)$ for each state of each model. For more detailed information this algorithm, the user can refer to [Zei87, ZPK00]

2.2.2 Atomic DEVS: Atomic

The atomic DEVS is implemented as `Atomic` in `Atomic.cs` file. `Atomic` is an *abstract* class that is derived from `Devs`.

```
public abstract class Atomic: Devs
{
    public TimeUnit TimeUnit { get; }
```

```
public Atomic(string name, TimeUnit ): base(name) {...}
...
```

An instance of `Atomic` class has its own time unit. `TimeUnit` is defined in `TimeUnit.cs` file as an enumerate type:

```
public enum TimeUnit { MilliSec, Sec, Min, Hour, Day }.
```

Therefore, the lifetime of a state s, $\tau(s)$ is interpreted in $\tau(s)*$ `TimeUnit` inside DEVS#. However, to handle all different schedules in different time units of all atomic models used in a simulation run, time conversions are internally done in DEVS#. As a result, `TimeLast`, `TimeNext`, `TimeElapsed`, `TimeRemaining`, and `TimeAdvance` will be interpreted in *second* internally.

Characteristic Functions

There are four public characteristic functions that are defined as *abstract* in `Atomic` class. 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 to its initial state s_0, such as at the beginning of a simulation run.

```
public abstract void init();
```

The function `tau()` returns the lifespan of the current state when the schedule of the next internal event is reset by the time of an interrupting input event or an generating output event.

```
public abstract double tau();
```

The function `delta_x(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.

```
public abstract bool delta_x(PortValue x) ;
```

The function `delta_y(ref 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()`.

```
public abstract void delta_y(ref PortValue y);
```

Displaying State as a string

There is an other public virtual(but not abstract) function `Get_s()`, that is supposed to return the current status in a string for display purpose.

```
public virtual string Get_s() { return ""; }
```

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()` is supposed to 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.

```
    public void CollectStatistics(bool flag) { m_cs = flag; }
    public bool CollectStatisticsFlag() const { return m_cs; }
    public virtual string Get_Statistics_s() const { return Get_s(); }
    public virtual Dictionary<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.

```
public class Coupled: Devs
{
    public Coupled(string name): base(name) {...}
```

Sub-components Related

There are four functions and one property associated with modeling of sub-components as follows.

```
    public void AddModel(Devs md);
    public Devs GetModel(string name);
    public void RemoveModel(string name);
    public void RemoveAllModels();
    public List<Devs> Models { get; }
```

Couplings Related

Related to couplings, there are three coupling related functions for adding, removing, and printing as follows.

```
    public void AddCP(Port spt, Port dpt);
    public void RemoveCP(Port spt, Port dpt);
    public void PrintCouplings() ;
```

2.3 Scalable Real-Time Engine: SRTEngine

DEVS# provides a simulation engine class, called `SRTEngine`. When we make an instance of `SRTEngine`, its constructor creates an independent simulation thread from the main thread. The reader can find the source codes of `SRTEngine` in `SRTEngine.cs` file.

2.3.1 Constructor

```
SRTEngine(Devs model, double ending_t, Callback call_back);
```

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

The third argument `Callback` is defined as `delegate` which can be seen as a function pointer in C#.

```
public delegate PortValue Callback(Devs model);
```

`Callback` is supposed to return a `PortValue` which represents the user input to the model. Thus, PortValue's `port` should be an input port of `Devs model`. 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);
}
```

Then, we can pass the above function pointer of `InjectMsg` to an instance of `SRTEngine` as follows.

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

We will see another example to use `Callback` in the example `Ex_VendingMachin` in Section 3.1.2.

2.3.2 Run console menu

If we call the function `RunConsoleMenu()` 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
>
```

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;

The corresponding application programming interfaces (APIs) are:

```
public double SRTEngine::GetTimeScale();
public void SRTEngine::SetTimeScale(double ts).
```

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. The corresponding API is

```
public void SRTEngine::Step().
```

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*. The corresponding API is

```
public void SRTEngine::MultiRun(unsigned n)
```

where n=1.

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. The corresponding API is

```
public void SRTEngine::MultiRun(unsigned n).
```

[p]ause

`pause` or `p` pauses a simulation run immediately. The corresponding API is

```
public void SRTEngine::void Pause().
```

pause_at *et*

`pause_at` sets the simulation ending time as *et*. The corresponding APIs are

```
public double SRTEngine::GetEndingTime() const;
public void SRTEngine::SetEndingTime(double 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`. The corresponding API is

```
public void SRTEngine::Continue().
```

reset

`reset` initializes the associated simulation model. The corresponding API is

```
public void SRTEngine::Reset().
```

rerun

`rerun` combines `reset` and `run`. The corresponding API is

```
public void SRTEngine::Rerun().
```

[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)` that is the third argument of the SRTEngine constructor. The corresponding API is

```
public void SRTEngine::Inject(PortValue x).
```

dtmode

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

- `none` displays no discrete state transition.
- `te` displays the elapsed time `TimeElapsed`.
- `tea` displays the elapsed time if associated model's current state s is active, which means the remaining time `TimeRemaining` $< \infty$.

- `tr` displays the remaining time.
- `tra` displays only an active remaining time.
- `nc` no change.

The corresponding APIs are

```
public void SRTEngine::Set_dtmode(PrintStateMode flag, bool ao);
public void SRTEngine::Get_dtmode(ref PrintStateMode flag, ref bool ao).
```

where

```
public enum PrintStateMode {P_NONE, P_remaining, P_elapsed}.
```

animode

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

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

The related APIs are

```
public void SRTEngine::SetAnimationFlag(bool flag),
public bool SRTEngine::GetAnimationFlag(),
public void SRTEngine::SetAnimationInterval(double ai),
public double SRTEngine::GetAnimationInterval() const.
```

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: te [(state, t_s, t_e), active only: off]
current animation mode:  on and interval= 0.25
current log setting: on, p00
```

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

The corresponding APIs are

```
public void SRTEngine::PrintTotalState() const,
public void SRTEngine::PrintCouplings() const,
public void SRTEngine::PrintSettings() const,
public void SRTEngine::PrintPerformanceOfaRun() const.
```

cls

`cls` clears the screen.

log

`log` sets the logging option which generates the log file `DEVS#_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.

`SRTEngine` has a public data field of `logger` which is an instance of `Logger` class. The corresponding APIS of Logger class are

```
public bool Logger::OnFlag {get; set; },
public bool Logger::TransitionFlag {get; set; },
public bool Logger::PerformanceFlag {get; set; },
public bool Logger::TotalStateFlag {get; set; }.
```

[e]xit

`exit` or `e` exits the console menu.

2.4 Random Variables

2.4.1 Probability Density Functions

Several random variables are provided in DEVS#. All random variable classes are defined in `Util\RV.cs`. `RVofGeneralPDF` class generates a random number of the uniform probability density function (PDF), the triangular PDF, the exponential PDF, and the normal PDF as follows.

```
public class RVofGeneralPDF: Random
{
    public RVofGeneralPDF() : base() { }
    public double Uniform(double min, double max);
    public double Triangular(double min, double max, double mode);
    public double Exponential(double mean);
    public double Normal(double mean, double sd);
}
```

We took a look at the use of `RVofGeneralPDF` in the ping-pong example in Section 1.3.

If we want to make an instance of a specific PDF's random variable, we can use `RV_Uniform`, `RV_Triangular`, `RV_Exponential`, and `RV_Normal` which are derived classes from an abstract class, `RVofPDF`. The abstract function `RN()` of `RVofPDF` is overrided in each derived class as follows.

```
public abstract class RVofPDF : Random
{
    protected RVofPDF() : base() { }
    public abstract double RN();
}
public class RV_Uniform : RVofPDF
{
    protected double min, max;
    public RV_Uniform(double _min, double _max): base(){...}
    public override double RN(){...} // return Uniform[min,max]
}
public class RV_Triangular : RVofPDF
{
    protected double min, max, mode;
```

```
    public RV_Triangular(double _min, double _max, double _mode): base(){...}
    public override double RN(){...} // return Triangular(min,max,mode)
}
public class RV_Exponential : RVofPDF
{
    protected double mean;
    public RV_Exponential(double _mean): base(){...}
    public override double RN(){...} // return Exponential(mean);
}

public class RV_Normal : RVofPDF
{
    protected double mean, sd;
    public RV_Normal(double _mean, double sd): base(){...}
    public override double RN() {...}// return Normal(mean, sd);
}
```

We will see the use of these PDFs in example in Section **??**

2.4.2 Probability Mass Function

A template class `RVofPMF<T>` is used for generating a random number of a probability mass function (PMF). This class has the probability table `pmf` which is `Dictionary<T, double>` whose key is T-typed object and value is the probability of occurrence for the key. The function `SampleV()` returns T-typed key whose cumulative pmf value is firstly greater than a random number `r` in uniform[0, 1].

```
public class RVofPMF<T> : Random
{
    //-- pairs of (T, probability)
    public Dictionary<T, double> pmf;
    public RVofPMF() : base() {...}
    public T SampleV() {...}
}
```

Figure 2.3 shows how `RVofPMF::SampleV()` works. Let's assume that there are three kinds of jobs, Job1, Job2, and Job3 coming into a system with their corresponding probabilities 0.5, 0.25, and 0.25, respectively. To pick one of them, we generate a random number r in the uniform PDF [0, 1]. Let's say $r = 0.6$ at this time. Then, since the inverse function of $F(0.6)$ that is the cumulative function of $f(x)$ is Job2, Job2 will be picked at this moment.

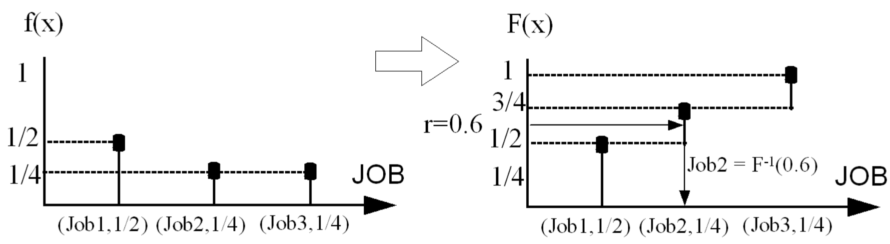

Figure 2.3: PMF $f(s)$ and its cumulative function $F(x)$

For this example, the user should fill out the table `pmf` as pmf={(Job1, 0.5), (Job2, 0.25), (Job3, 0.25) } before calling `SampleV()` function. Of course, summation of values `pmf[key]` for all `key`s should be 1 for being a correct PMF function.

We will see an application of `RVofPDF` and `RVofPMF` when we take a look at `Generator` class in Section 4.2.1

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 class from `Atomic`. In `Timer.cs` file, `Timer` is defined to generat an output `op` every 3.3 seconds as illustrated in Figure 3.1.

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

```
public class Timer : Atomic
{
    private OutputPort op;

    public Timer(string name): base(name, TimeUnit.Sec)
    {   op = AddOP("op"); init(); }
    public override void init(){}
    public override double tau() { return 3.3; }
```

Since there is no input transition defined, `delta_x` has the null body except returns `false` [1]. However, `delta_y` returns the output `op` by making the output event `y` set to `op`.

[1] Actually, there is no difference between return false or true in this example.

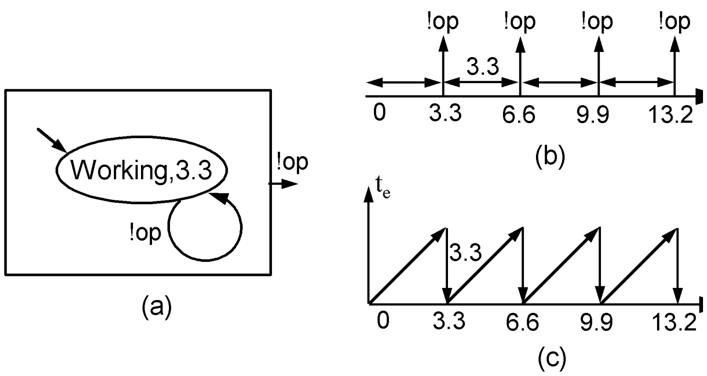

Figure 3.1: Timer (a) State Transition Diagram (b) Event Segment (c) t_e Trajectory

```
        public override bool delta_x(PortValue x) { return false; }
        public override void delta_y(ref PortValue y) { y.Set(op);  }
```

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

```
        public override string Get_s() { return "Working"; }
    }
```

The file `Program_Timer.cs` has the main function for a console application as follows. As we can see, we make an instance `timer` from `Timer` class. And then we make an instance of `SRTEngine`. Here, we don't have to pass the third `Callback` function argument because this `Timer` example doesn't need the user input. Finally, this codes run the console menu of `SRTEngine` class.

```
class Program
{
    static void Main(string[] args)
    {
        Timer timer = new Timer("STimer");
        SRTEngine Engine = new SRTEngine(timer, 10000, null);
        Engine.RunConsoleMenu();
    }
}
```

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.

Exercise 3.1 Consider the example `Ex_Timer`.

a. Let's change the display mode from `te` to `tr` 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_r`, respectively? Guess the value of `t_e` at this moment.

b. Add one more state variable `int n` in `Timer` class. `n` should be set = zero in `init()`, and it should increase by one in `delta_y()`. `Get_s()` shows `n` in the string format

```
string.Format("Working, n={0}", n);
```

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'. [2] 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. [3]

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

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

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

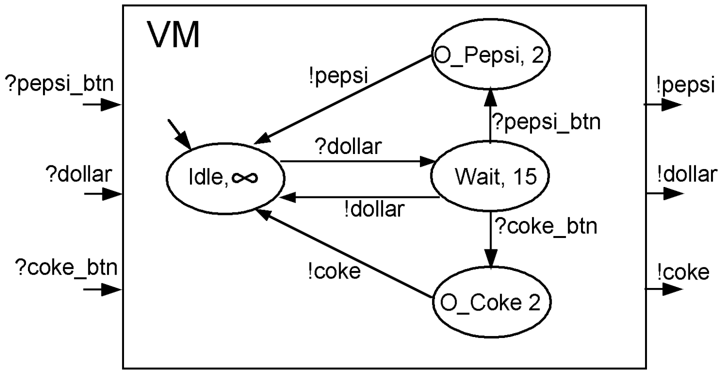

Figure 3.2: State Transition Diagram of Vending Machine

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 which is defined in `VendingMachine.cs` file. The class `VM` has three input port `idollar`, `pepsi_btn` and `coke_btn`; three output port `odollar`, `pepsi`, `coke`, all assigned by returning values of the `AddIP` and `AddOP` functions in the constructor.

```
public class VM : Atomic
{
    public InputPort idollar, pepsi_btn, coke_btn;
    public OutputPort odollar, pepsi, coke;
    enum PHASE { Idle, Wait, O_Pepsi, O_Coke }
    PHASE m_phase;

    public VM(string name) : base(name, TimeUnit.Sec)
    {
        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 ∞ for `Wait`, `O_Pepsi`, `O_Coke`, and `Idle`, respectively.

```
public override void init() { m_phase = PHASE.Idle; }
public override double tau()
{
    if (m_phase == PHASE.Wait)
        return 15;
    else if (m_phase == PHASE.O_Pepsi)
        return 2;
    else if (m_phase == PHASE.O_Coke)
        return 2;
    else
        return double.MaxValue;
}
```

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.

```
public override bool delta_x(PortValue x)
{
    if (m_phase == PHASE.Idle && x.port == idollar)
    {
        m_phase = PHASE.Wait;
        return true; // Reschedule Me
    }
    else if (m_phase == PHASE.Wait && x.port == pepsi_btn)
    {
        m_phase = PHASE.O_Pepsi;
        return true; // Reschedule Me
    }
    else if (m_phase == PHASE.Wait && x.port == coke_btn)
    {
        m_phase = PHASE.O_Coke;
        return true; // Reschedule Me
```

```
        }
        return false; // Ignore the input
    }
```

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

```
    public override void delta_y(ref PortValue ys)
    {
        if (m_phase == PHASE.Wait)
            ys.Set(odollar);
        else if (m_phase == PHASE.O_Pepsi)
            ys.Set(pepsi);
        else if (m_phase == PHASE.O_Coke)
            ys.Set(coke);
        m_phase = PHASE.Idle;
    }
```

The virtual function `Get_s()` is also overridden and returns an `m_phase.ToString()`.

```
    public override string Get_s()
    {
        return m_phase.ToString();
    }
}
```

Since this vending machine example needs the user input during a simulation run, we need to define a callback function for the user input. In `Program_VM.cs` file, we can see the following static function.

```
    static PortValue InjectMsg(Devs model)
    {
        if (model is VM)
        {
            VM vm = (VM)model;
            Console.Write("[d]ollar [p]epsi_botton [c]oca_botton > ");
            string input = Console.ReadLine();
            if (input == "d")
                return new PortValue(vm.idollar);
            else if (input == "p")
                return new PortValue(vm.pepsi_btn);
            else if (input == "c")
                return new PortValue(vm.coke_btn);
            else
```

```
            {
                Console.WriteLine("Invalid input! Try again!");
                return new PortValue(null,null);
            }
        }
        else
            throw new Exception("Invalid Model!");
    }
```

The callback function `InjectMsg` casts the type of md from `Devs` to `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 `Program_VM.cs` 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.

```
    static void Main(string[] args)
    {
        VM vm = new VM("VM");
        SRTEngine Engine = new SRTEngine(vm, 10000, InjectMsg);
        Engine.RunConsoleMenu();
    }
```

Let's try the command `step`. Observe that since the initial state s_0 of `VM` is `Idle` and its lifespan `tau(Idle)`=∞, and the initial schedule is also `t_s`=∞. In this case, the elapsed time `t_e` cannot ever reach `t_s`. Thus this command `step` doesn't stop until 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` command, 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)
```

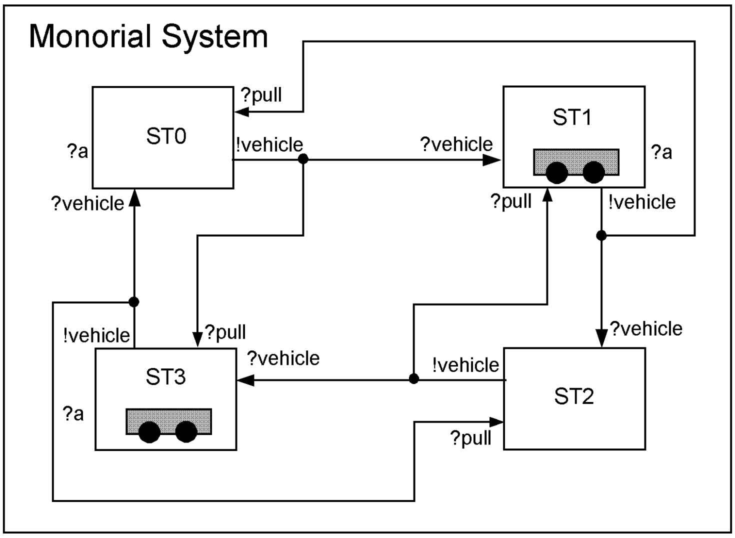

Figure 3.3: Monorail System

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. Return `t_s` and `t_e` to the values they had right before the input of the additional dollar.

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

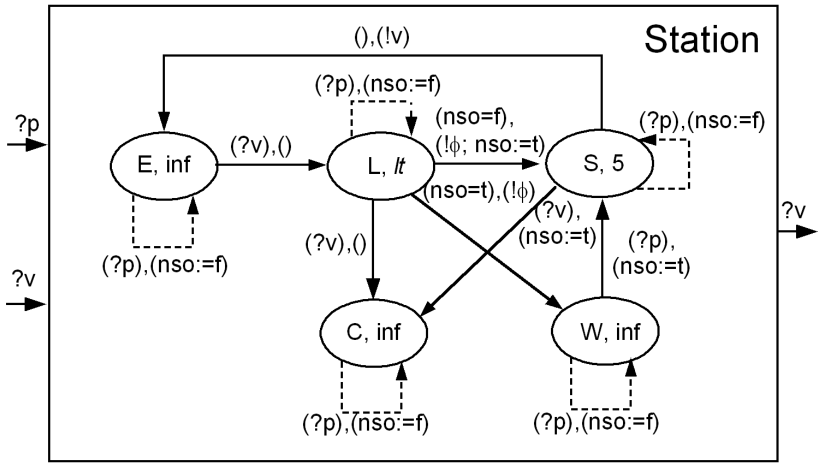

Figure 3.4: Phase Transition Diagram of Station
(A dashed line indicates $\delta_x(s, t_s, t_e, x) = (s', 0)$.)

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

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

Station

First of all, a macro `REMEMBERING` in the first line in `Station.cs` file is defined for testing the effect of monitoring the next station's status using `nso`.

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

The class `Station` has several state variables: `m_phase` being one of `enum PHASE {Sending, Empty, Loading, Waiting, Collided}`; `bool init_occupied` indicating the initial occupation state of the station, `bool nso` indicating if the next station is occupied or not; and the constant variable `double loading_t` indicating the lifespan of a state when its phase is `Loading`.

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

```
public class Station: Atomic
{
    enum PHASE {Sending, Empty, Loading, Waiting, Collided}
    PHASE m_phase;

    readonly bool   init_occupied;
    bool   nso; //next_state_occpied
    readonly double loading_t;

    public InputPort ipull, ivehicle;
    public OutputPort ovehicle;

    public Station(string name, bool occupied, double lt):
        base(name, TimeUnit.Sec)
    {
        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`.

```
public override void init()
{
    if(init_occupied == true)
        m_phase = PHASE.Sending;
    else
        m_phase = PHASE.Empty;
}
```

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

```
public override double tau()
{
    if (m_phase == PHASE.Sending)
        return 10;
    else if (m_phase == PHASE.Loading)
        return loading_t;
    else
        return double.MaxValue;
}
```

`Station::delta_x` defines the input transition such that if it receives an input through `ipull`, it marks `nso = false` which means that "the next station is not occupied any more". At that time, if the station's phase is `Waiting`, `delta_x` then changes the phase to `Sending`. To remember the next station be occupied by this Sending action, `Station::delta_x` sets `nso=tru` 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`.

```
    public override bool delta_x(PortValue x)
    {
        if( x.port == ipull) {
            nso = false;
            if (m_phase == PHASE.Waiting)
            {
#if REMEMBERING
            nso = true;
#endif
                m_phase = PHASE.Sending;
                return true;
            }
        }
        else if(x.port == ivehicle) {
            if(m_phase == PHASE.Empty)
                m_phase = PHASE.Loading;
            else // rest cases lead to Colided!
                m_phase = 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 `Empty`.

```
    public override void delta_y(ref PortValue y)
    {
        if (m_phase == PHASE.Loading)
        {
            if(nso == true)
                m_phase = PHASE.Waiting;
            else {
#if REMEMBERING
            nso = true;
#endif
                m_phase = PHASE.Sending;
            }
        }
        else if (m_phase == PHASE.Sending)
        {
            y.Set(ovehicle);
            m_phase = PHASE.Empty;
        }
    }
```

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

```
    public override string Get_s()
    {
        return string.Format("phase= {0}, nso= {1}", m_phase, nso);
    }
```

Monorail System

To construct the monorail system, we will make four instances from `Station` as shown in `Program_Monorail.cs` file. `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. The ending time of simulation is 1000, and there is no callback function used here.

```
static Coupled MakeMonorail(string name)
{
    Coupled monorail = new Coupled(name);
    Station ST0 = new Station("ST0", false, 10);
    ST0.CollectStatistics(true);
    monorail.AddModel(ST0);

    Station ST1 = new Station("ST1", true, 30);
    ST1.CollectStatistics(true);
    monorail.AddModel(ST1);

    Station ST2 = new Station("ST2", false, 10);
    ST2.CollectStatistics(true);
    monorail.AddModel(ST2);

    Station ST3 = new Station("ST3", true, 10);
    ST3.CollectStatistics(true);
    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;
}

static void Main(string[] args)
{
    Coupled ms = MakeMonorail("mr");

    SRTEngine Engine = new SRTEngine(ms, 1000, null);
    Engine.RunConsoleMenu();
}
```

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.

When the simulation stops, DEVS# makes the beep sounds every 1 second. To stop the beep sounds, input RETURN key (two times it is kind of bugs but I could not fix it yet). The following screen is the results of the command `print p`.

```
mr.ST0
phase= Empty, nso= True: 0.590
phase= Empty, nso= False: 0.000
phase= Loading, nso= False: 0.010
phase= Sending, nso= True: 0.200
phase= Loading, nso= True: 0.190
phase= Waiting, nso= True: 0.010

mr.ST1
phase= Sending, nso= False: 0.010
phase= Empty, nso= False: 0.020
phase= Loading, nso= False: 0.400
phase= Sending, nso= True: 0.190
phase= Empty, nso= True: 0.190
phase= Loading, nso= True: 0.190

mr.ST2
phase= Empty, nso= True: 0.400
phase= Loading, nso= True: 0.000
phase= Loading, nso= False: 0.200
phase= Sending, nso= True: 0.200
phase= Empty, nso= False: 0.200

mr.ST3
phase= Sending, nso= False: 0.010
phase= Empty, nso= False: 0.210
phase= Loading, nso= False: 0.200
phase= Sending, nso= True: 0.190
```

```
phase= Empty, nso= True: 0.390
```

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= Loading, nso= False: 0.200` 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% (`phase= Loading, nso= False: 0.400` and `phase= Loading, nso= True: 0.190`). 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.cs` 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. □

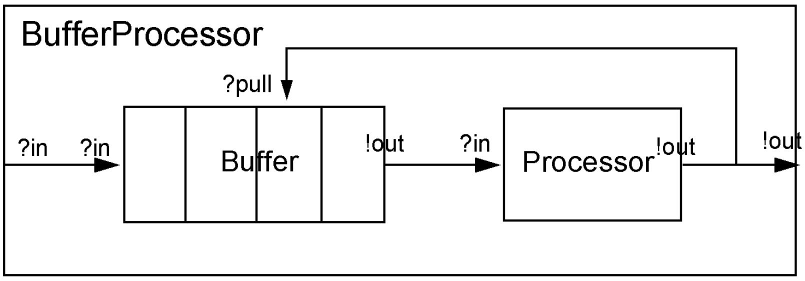

Figure 4.1: A System having a Buffer and a Processor

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.

Suppose that an *activity* consists of two events such that one begins at t_l and the other ends at t_u. Then the *activity duration* is $t_u - t_l$. If we have activity data as a set of time pairs $A = \{(t_{li}, t_{ui}) | t_{li} \leq t_{ui}\}$ where i is in some index set, $N = \{1, 2, \ldots, n\}$, then the (average) *cycle time* is

$$t_{cyc}(A) = \frac{\sum_{i \in N} (t_{ui} - t_{li})}{n}. \tag{4.2}$$

Cycle time can be interpreted in different contexts. 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`.

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

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`

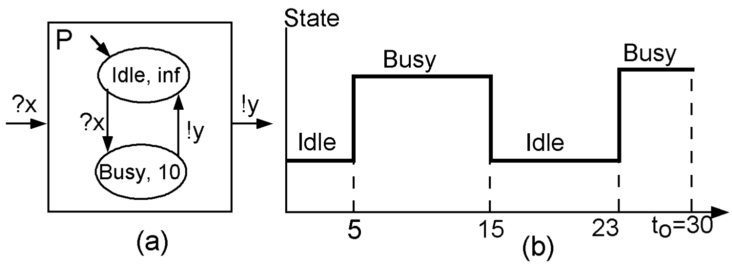

Figure 4.2: State Trajectory of a Processor

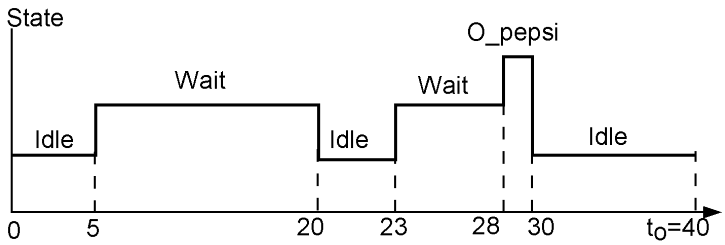

Figure 4.3: A State Trajectory of Vending Machine

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 of a piece-wise constant segment is defined by $td : S \times \mathbb{N} \rightarrow T$ where $\mathbb{N}$ is a set of natural numbers. This function maps from state s and the order $i \in \mathbb{N}$ of a segment piece to a time span value if the segment piece in the state s, otherwise the value is 0. For example, in the state trajectory of Figure 4.3, $td(\texttt{Idle}, 1) = 5 - 0 = 5$, while $td(\texttt{Idle}, 2) = 0$ because the state of the second segment is `Wait`.

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

$$P(C = s) = \frac{\sum\limits_{i \in N} td(s, i)}{t_o}. \tag{4.3}$$

It is true that

$$\sum_{s\in S}\sum_{i\in N} td(s,i) = t_o. \tag{4.4}$$

So it is also true that

$$\sum_{s\in S} P(C=s) = \sum_{s\in S}\left(\frac{\sum_{i\in N} td(s,i)}{t_o}\right) = \frac{t_o}{t_o} = 1. \tag{4.5}$$

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 $t \in T$ 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.

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

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

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

$$E(C) = \sum_{x\in\mathbb{N}} xP(C=x) \tag{4.7}$$

The *Average Queue Length* is defined as Equation (4.7).

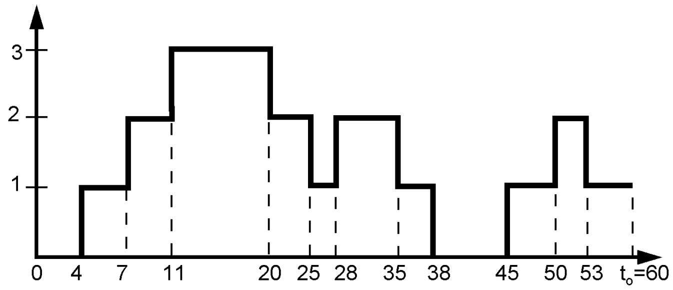

Figure 4.4: Trajectory of Queue

Example 4.4 Suppose that we have a state trajectory of a queue as shown in Figure 4.4. By Equation (4.6), 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 (4.7), the Average Queue Length is $E(C = x){=}0{*}0.183{+}1{*}0.35{+}2{*}0.317{+}3{*}0.15{=}1.434$. □

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 $X_1, X_2, \ldots X_n$ from n simulation runs as illustrated in Figure 4.5. Then the *sample mean*

$$\hat{\mu} = \frac{\sum_{i=1}^{n} X_i}{n} \tag{4.8}$$

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

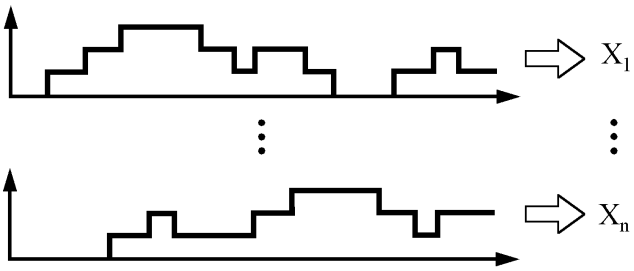

Figure 4.5: IID random variants $X_1 \ldots X_n$ from n simulation runs

variance

$$\hat{\sigma}^2(n) = \frac{\sum_{i=1}^{n}[X_i - \hat{\mu}]^2}{n-1} \tag{4.9}$$

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

$$\hat{\mu} \pm t_{n-1,1-\alpha/2}\sqrt{\frac{\hat{\sigma}^2(n)}{n}} \tag{4.10}$$

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

$$P\left[\hat{\mu} - t_{n-1,1-\alpha/2}\sqrt{\frac{\hat{\sigma}^2(n)}{n}} \leq \mu \leq \hat{\mu} + t_{n-1,1-\alpha/2}\sqrt{\frac{\hat{\sigma}^2(n)}{n}}\right] = 1-\alpha \tag{4.11}$$

and we say that we are 100(1-a) percent confident that the real μ lies in the interval given by Equation (4.10).

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 $\hat{\sigma}^2$=1.7 and the 90% confidence interval for μ is $\hat{\mu} \pm t_{9,0.95}\sqrt{\frac{\hat{\sigma}^2(n)}{n}} = 13.4 \pm 1.83\sqrt{\frac{1.7}{10}} = 13.4 \pm 0.75$ □

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 version 1.2.1.

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 `DEVSsharp/ModelBase/` 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] `ModelBase` library provides a basic class for flow entities, called the class `Job` in `Job.cs` file.

Job

`Job` class has public data fields: `int type`, `int id` and `Dictionary<string, double> TimeMap`. `TimeMap` will be used for stamping a pair of an event string and its occurrence time (we will see examples in `Generator` class and `Transducer` class later).

There are tree constructors, a string conversion function `ToString()`. The virtual function, `Clone()` is supposed to return a clone of this class instance.

```
public class Job
{
    public int type;
    public int id;
    public Dictionary<string, double> TimeMap;

    public Job(int Type) {...}
    public Job(int Type, int Id){...}
    public Job(Job ob) { ... }

    public override string ToString();

    public virtual Job Clone() { return new Job(this); }
}
```

To generate and to collect instances of `Job` class, we will use two atomic models: `Generator` in `Generator.cs` and `Transducer` in `Transducer.cs`, which are key models in the experimental frame. 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.

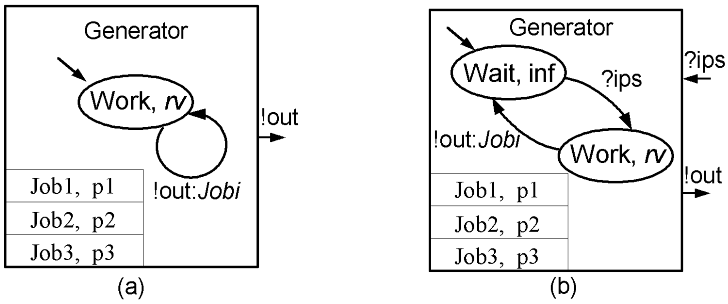

Figure 4.6: State Transition Diagrams of Generator: (a) Autonomous Mode (b) Non-Autonomous Mode

Generator

The class `Generator` in `DEVSsharp/ModelBase/Generator.cs` produces jobs depending on (1) autonomous mode, (2) generating delay pdf, and (3) producing job spectrum.

If `Generator`'s `bool m_bAutonomous` filed is `true` then `Generator` keeps producing at `Work` phase, while it is `false` then `Generator` will be waiting for an external pull signal through the input port, `ips` as shown in Figure 4.6(b).

The generating time delay at `Work` phase of `Generator` can be determined by an random variable `rv` of a given pdf, `m_PDFofInterGenerating`.

The job spectrum can be determined by a PMF in which the probability of each job class is described. For more detail information for the class of `PVofPMF`, revisit to Section 2.4. The following codes show the interface ports, and internal data fields. Here `int m_no_gen` is used for tracking the job id which will be unique during a simulation run.

```
public class Generator: Atomic
{
    public OutputPort oout;//-- output port for job
    public InputPort ips;//-- pull signal port
    bool m_bAutonomous; // automatic or manual
    RVofPMF<Job> m_JobSpectrum; //-- Job Spectrum
    RVofPDF m_PDFofInterGenerating; // pdf of inter-generating time
    int m_no_gen; //-- to no of generating: used for job.id

    enum PHASE { Wait, Work }
    PHASE m_phase;
```

The constructor of `Generator` needs five arguments: (1) `name` as a string, (2) `TimeUnit tu`, (3) `bool auto` indicating autonomous or non-autonomous which is assign to the internal variable `m_bAutonomous`, (4) `InterGenerating` PDF as `RVofPDF`, and (5) Job spectrum as `RVofPMF<Job>` as follows.

```
public Generator(string name, TimeUnit tu, bool auto, RVofPDF InterGenerating,
                 RVofPMF<Job> JobSelection):base(name,tu){...}
```

`Generator::init()` resets `m_no_gen` to zero. `m_phase` is set to `Work` if `m_bAutonomous` is `true`, otherwise, it is set to `Wait`. If `m_JobSpectrum` is not null `m_statistics` is initialized by each type of job available in `m_JobSpectrum`. `m_statistics` of `Generator` will collect statistics *how may jobs have been produced*.

```
public override void init()
{
    m_no_gen = 0; //
    if (m_bAutonomous)
        m_phase = PHASE.Work;
    else
        m_phase = PHASE.Wait;

    if (m_JobSpectrum != null)
    {
        foreach (Job job in m_JobSpectrum.pmf.Keys)
            m_statistics.Add(job.type.ToString(), 0.0);
    }
}
```

When the phase of `Generator` is `Work`, `Generator::tau()` returns a random value from the pdf `m_PDFofInterGenerating`(which is set through the constructor function of `Generator`). If `Generator`'s phase is `Wait`, it returns the infinity as the lifespan of `Wait`.

```
public override double tau()
{
    if (m_phase == PHASE.Work)
    {
        double t = m_PDFofInterGenerating.RN();
        return t;
    }else // PHASE.Wait
        return double.MaxValue;
}
```

`Generator::detla_x()` treats the situation that `Generator` is non-autonomous, and it receives the pull signal through `ips` port when it waits for the signal. Otherwise, `Generator` ignores any input signal.

```
public override bool delta_x(PortValue x)
{
    if (m_bAutonomous == false && x.port == ips &&
        m_phase == PHASE.Wait)
    {
        m_phase = PHASE.Work;
        return true;
    }
    return false;
}
```

In `Generator::detla_y()`, `Generator` has a non-trivial `m_JobSpectrum` , `Generator` picks one job by calling `m_JobSpectrum.SampleV()` and clones the picked job to `clnt`. The unique job id for `clnt` is assigned. At this time, `Generator` stamps the current time into `clnt`'s `TimeMap` with its key value as the string "`SysIn`". This event time will be used when collecting System Time Performance Index by `Transducer` that we will look through later.

To collect of how many different jobs are generated, `Generator` accumulates the number of jobs generated with respect to their job types.

After generating a job (or null job) through `oout` port, if `Generator` is non-autonomous, it goes back to the phase `Wait`, otherwise it keep generating the next job by staying its phase `Work`.

```
public override void delta_y(ref PortValue y)
{
    if (m_JobSpectrum != null && m_JobSpectrum.pmf.Count > 0)
    {
        Job clnt = (m_JobSpectrum.SampleV()).Clone();
        clnt.id = ++m_no_gen;
        //-- (event, time) stamping
        clnt.TimeMap.Add("SysIn", Devs.TimeCurrent);
        y.Set(oout, clnt);

        m_statistics[clnt.type.ToString()] += 1;
    }
    else // no job value sent
        y.Set(oout);

    if (m_bActive == false)
        m_phase = PHASE.Wait;
}
```

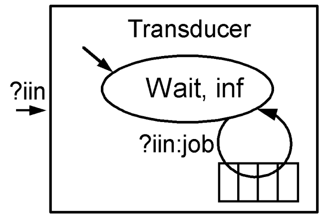

Figure 4.7: State Transition Diagrams of Transducer

Transducer

Transducer's behavior is pretty much opposite to that of Generator. Figure 4.7 shows its state transition diagram. It has an input port iin and a buffer Collector as Dictionary<int, List<Job>> to collect jobs coming in with respect to their types.

```
public class Transducer: Atomic
    {
        public InputPort iin;
        Dictionary<int, List<Job>> Collector;

        public Transducer(string name, TimeUnit tu): base(name, tu)
        {
            CollectStatistics(true); // default collecting statistics
            iin = AddIP("in");
            Collector = new Dictionary<int, List<Job>>();
        }
```

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

```
    public override void init() { Collector.Clear(); }
    public override double tau() { return double.MaxValue; }
```

Transducer::delta_x() castes the input value x.value to pv of Job 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;

```
    public override bool delta_x(PortValue x)
    {
        Job pv = (Job) x.value;
        if(pv != null)
        {
```

```
            //-- (event, time) stamping
            pv.TimeMap.Add("SysOut", Devs.TimeCurrent);
            if (Collector.ContainsKey(pv.type) == false)
                Collector.Add(pv.type, new List<Job>());
            Collector[pv.type].Add(pv);
        }
        //else
        //    throw new Exception("Type casting Failed!");
        return false;
    }
```

Recall that Transducer collects incoming Jobs 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.

- Average Throughput value defined in Equation (4.1) is the number of Jobs in Collector divided by the current time that is the observation time-length t_o of a simulation run.
- Average System Time defined in Equation (4.2) is the average value of all time durations (arrival-time, departure-time) for each Job in Collector.
- In addition, even though the number of jobs went through the system can be recalculated by Average Throughput, it will be displayed by counting the jobs collected into Transducer in terms of their job types.

The function Transducer::GetPerformance() returns these three indices. The source code of GetPerformance() is available in Transducer.cs.

4.2.2 Utilization in DEVS#

Recall that to get Utilization, we need to accumulate the time intervals of piecewise 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).

```
public override void when_receive_cs()
{
    double dT = TimeCurrent - TimeLastcs;// dT: accumulating time span
    if (CollectStatisticsFlag())
    {
        string state_str = Get_Statistics_s();
        if (m_statistics.ContainsKey(state_str)==false)
            m_statistics.Add(state_str, 0.0); // add new entry
        m_statistics[state_str] += dT;
    }
    TimeLastcs = TimeCurrent;
}
```

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 Dictionary<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.

```
public virtual Dictionary<string, double> GetPerformance()
{
    Dictionary<string, double> statistics = new Dictionary<string,double>();
    if(CollectStatisticsFlag()==true) {
        foreach(string str in m_statistics.Keys)
        {
            double probability = m_statistics[str] / Devs.TimeCurrent;

            if(probability < 0.0 || probability > 1.0)
                throw new Exception("Invalid Probability!");
            else
```

```
                statistics[str] = probability;
            }
        }
        return statistics;
    }
```

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

Thus `statistics[key]` contains the $P(C{=}\texttt{key})$ of Equation (4.3) over the interval from 0 to the current time.

4.2.3 Average Queue Length in DEVS#

The class `Buffer` in `DEVSsharp/ModelBase/Buffer.cs` 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 `m_Jobs` as follows.

```
public override string Get_Statistics_s()
{   // length Only
    return string.Format("{0}", m_Jobs.Count);
}
```

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

```
public override Dictionary<string, double> GetPerformance()
{
    Dictionary<string, double> statistics = new Dictionary<string,double>();
    if(CollectStatisticsFlag()==true) {
        double E_i=0; // expectation of queueing line length
        foreach(string key in m_statistics.Keys)
        {
            double probability = m_statistics[key]/TimeCurrent;// P(i)

            if(probability < 0.0 || probability > 1.0) {
                throw new Exception("Invalid Probability!");
            }
            else{
                int i = System.Convert.ToInt32(key);
                E_i += probability * i;// E(i)=\Sum_{i} i*P(i)
            }
        }
        statistics.Add("Average Q length: ", E_i);
```

```
        }
        return statistics;
    }
```

It makes $P(C{=}\texttt{i})$ using `m_statistics[i]`. Then it makes $E(C)$ by summing over `i`$*P($`i`$)$ for all `i` as defined in Equation (4.7).

The rest other functions defined in `Buffer` class will be examined in the next section.

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 4.8 illustrates the case of $n = 3$. However, all classes used in this example are defined in `ModelBase` library. Thus this example show how to use the classes defined in other project too. Observed that References of `Ex_ClinetServer` displayed in Solution Explore includes `DEVSsharp` and `ModelBase`.

The entire simulation model consists of the client-server system under test, named `CS`, and the experimental frame, named `EF`, as shown in Figure 4.8. 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.

4.3.1 Server

`Server` defined in `DEVSsharp/ModelBase/Server.cs` is a concrete class derived from `Atomic`. The state transition diagram of `Server` can be drawn as shown in Figure 4.9(a). The C# codes of `Server` available in `DEVSsharp/ModelBase/Server.cs` and are represented by Figure 4.9(a) there is no need for further explanation here.

4.3.2 Buffer

`Buffer` defined in `DEVSsharp/ModelBase/Buffer.cs` 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_Jobs` is a buffer keeping incoming clients whose type is `Job` class; `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.

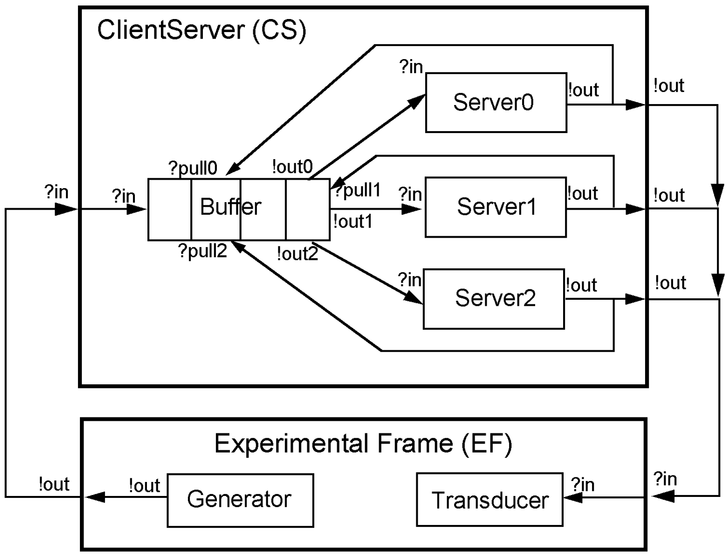

Figure 4.8: Configuration of Client Server System $n = 3$

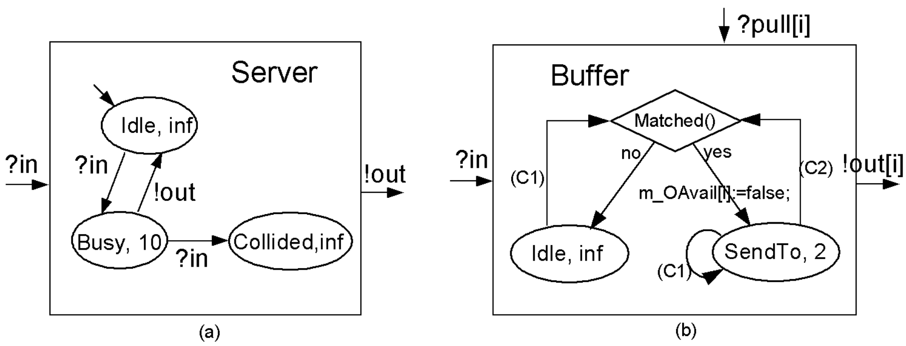

Figure 4.9: Server and Buffer

```
public class Buffer: Atomic
{
    public InputPort iin;
    public List<InputPort> pull;
    public List<OutputPort> oout;

    private List<Job> m_Jobs; // FIFO Buffer
    protected enum PHASE { Idle, SendTo, Ask };
    protected PHASE m_phase;
    protected List<bool> m_OAvail;//check availability of next resource
    protected int m_Osize; // size of output
    protected int m_send_index; // index to which it will send
    protected double m_send_time;
```

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 `Job` and pushes it back to the buffer `m_Jobs`. 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.

```
virtual protected void C1(PortValue x)
{
    if(x.port == iin){ //receiving a client
        if(x.value is Job) {
            Job client = (Job)x.value;
            if(client != null)
                m_Jobs.Add(client);
        }
        else
            throw new Exception("Invalid Input!");
    }
    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 job in `m_Jobs` 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.

```
private bool Matched()
{
    if(m_Jobs.Count > 0){
        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
                m_send_index = i; // remember i in m_send_index
                return true;
            }
        }
        return false;
    }else
        return false;
}
```

The function `C2` creates an the output event and removes the first job from `m_Jobs` when `C2`'s phase is `SendTo`.

```
virtual protected void C2(ref PortValue y)
{
    if(m_phase == PHASE.SendTo){
        Job job = m_Jobs[0];
        y.Set(oout[m_send_index], job);
        m_Jobs.RemoveAt(0);// remove the first client
    }
}
```

The function `init()` of `Buffer` resets phase to `Idle`, assigns `m_OAvial[i]=true` for all indices, and `clears` all jobs in `m_Jobs`.

```
public override void init()
{
    m_phase = PHASE.Idle;
    m_Jobs.Clear();
    m_send_index = 0;
    m_OAvail.Clear();
    for (int i = 0; i < m_Osize; i++)
        m_OAvail.Add(true); // add variable
}
```

Buffer's tau() returns ∞ for Idle and returns 2.0 for SendTo.

```
public override double tau()
{
    if (m_phase == PHASE.Idle)
        return double.MaxValue;
    else
        return m_send_time;
}
```

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.

```
public override bool delta_x(PortValue x)
{
    C1(x);
    if (m_phase == PHASE.Idle)
    {
        if (Matched())
        {
            m_phase = PHASE.SendTo;//
            return true; // reschedule as active
        }
    }
    return false;
}
```

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

```
public override void delta_y(ref PortValue y)
{
    C2(ref y);
    if (Matched())
        m_phase = PHASE.SendTo;
    else
        m_phase = 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.cs.

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 4.8.

All atomic models' time units were set as `TimeUnit.Sec`. And `Generator` used here was autonomous and it's inter-generating job time was a random number of the exponential pdf with mean 5; the time period at `Busy` of `Server` was constant 10; `Buffer`'s sending delay time was 2.

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 static function defined in `Ex_ClientServer/Program_CS.cs`.

```
static Coupled MakeTotalClientServerSystem(int n)
```

The simulation settings we use here are: the simulation ending time=10,000 second; 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.

```
static void Main(string[] args)
{
    Coupled Sys = MakeTotalClientServerSystem(5) ;
    Sys.PrintCouplings();

    SRTEngine simEngine = new SRTEngine(Sys, 10000, null); //
    simEngine.SetAnimationFlag(false);
    simEngine.SetTimeScale(double.MaxValue); //
    simEngine.Set_dtmode(SRTEngine.PrintStateMode.P_NONE, false);
    simEngine.RunConsoleMenu();
}
```

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# summarizes the performance indices to the console. [2] Table 4.1 shows performance indices for each configuration and Figure 4.10s 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/sec at n=3 and then there is no increase at n=4 and 5. The reason why Throughput doesn't

[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# again.

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 (sec.)	2,927.33	873.86	18.30	13.54	12.86
Throughput (jobs/sec.)	0.08	0.17	0.20	0.20	0.20
Utilization	0.83	0.83	0.67	0.50	0.40

The simulation run time was t_o = 10,000 seconds; 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.

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 sec, 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 function `Buffer::Matched()` 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.

```
...
============= Total Performance Indices =========
CSsystem.CS.BF
Average Q length: : 0.567, 95% CI: [0.562, 0.573]

CSsystem.CS.SV0
Idle: 0.385, 95% CI: [0.382, 0.388]
Busy: 0.615, 95% CI: [0.612, 0.618]
```

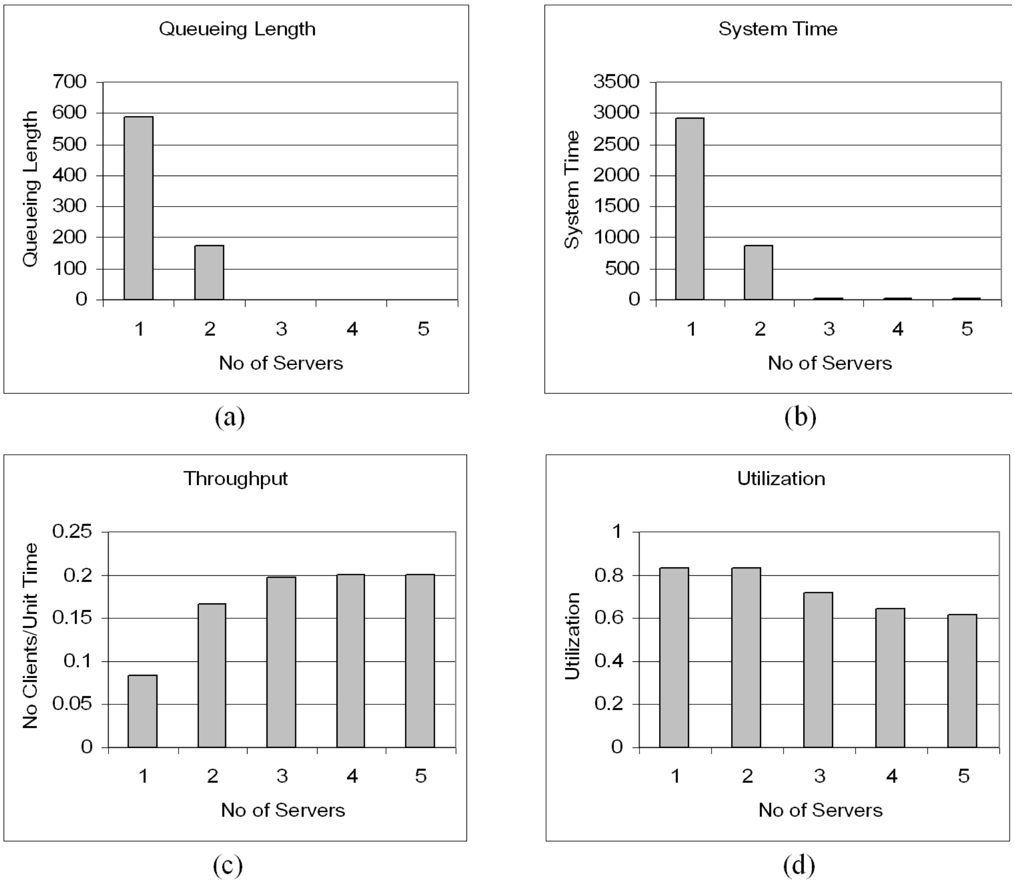

Figure 4.10: Performance Indices

```
CSsystem.CS.SV1
Idle: 0.471, 95% CI: [0.468, 0.474]
Busy: 0.529, 95% CI: [0.526, 0.532]

CSsystem.CS.SV2
Idle: 0.584, 95% CI: [0.581, 0.587]
Busy: 0.416, 95% CI: [0.413, 0.419]

CSsystem.CS.SV3
Idle: 0.713, 95% CI: [0.707, 0.718]
Busy: 0.287, 95% CI: [0.282, 0.293]

CSsystem.CS.SV4
Idle: 0.835, 95% CI: [0.830, 0.839]
Busy: 0.165, 95% CI: [0.161, 0.170]

CSsystem.EF.Trans
Average System Time of Job1 (Sec): 12.817, 95% CI: [12.791, 12.843]
# Job1 went through the system during 10000.00 Secs: 2011.500, 95% CI: [2000.585
, 2022.415]
Average Throughput of Job1 per Sec: 0.201, 95% CI: [0.200, 0.202]

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

Chapter 5

Appendix: Building Projects using DEVS#

This appendix covers the directory structure of DEVS#, how to build the example projects provided in DEVS# library, and how to add new project.

5.1 Directory Structure of DEVS#

In DEVS# version 1.2.1, the root director of DEVS# is `DEVSsharp` in which many classes covered in Chapter 2 such as `Named`, `Port`, `PortValue`, `Devs`, `Atomic`, `Coupled`. However, the classes of random variables are contained in the sub-directory `DEVSsharp\Util`. `DEVSsharp\Doc` contais this document in a PDF file and HTML files. `DEVSsharp\Examples` has example projects such as `Ex_Timer`, `Ex_PingPong`, `Ex_VendingMaching`, and `Ex_ClientServer`; `DEVSsharp\ModelBase` contains classes which can be used in several other projects

```
DEVSsharp
   +-- Util
   +-- Doc
   +-- Examples
          ...
   +-- ModelBase
          ...
   +-- Verifiable
          ...
```

Figure 5.1: Directory Structure of DEVS#

such as `Generater`, `Transducer`, `Buffer`, `Server`, and so on; `DEVSsharp\Verifiable` containing all classes which are used for verification that was not covered in this document at all. For verification research, the author has a plan to write another document soon.

5.2 Building Simulation Examples in DEVS#

Modeling and simulation examples projects covered in Chapters 2, 3 and 4 will be available `DEVSsharp/Examples/Ex_`*Projects*. We can open each solution file whose file extension is `sln` in each project folder. For example, we can open `DEVSsharp/Examples/Ex_ClientServer/Ex_ClientServer.sln`. Each solution file will opens reference project(s) such as `DEVSsharp` (and `ModelBase` for `Ex_ClientServer`) as well as the example project itself.

In addition, if we open `DEVSsharp/DEVSsharp.sln`, Visual Studio 2005 will open all examples of `Timer`, `PingPong`, `ClientServer` and `VendingMachine` as well as `DEVSsharp` and `ModelBase` as shown in Solution Explorer window in Figure 5.2.

5.3 Adding Our Own Project

This section shows how to add new project which uses DEVS# library in a step-by-step procedure.

1. **Create a new project.** Select `File->New->Project...` menu. Then we can see a dialog box as shown in Figure 5.3. We assume to create Console Application at `DEVSsharp\Examples`. The location we want to create new project can be selected by [`Browser...`] button. The new project is titled as `MyProject`.

   Then we can see the window screen of Visual Studio 2005 after creating `MyProject` as shown in Figure 5.4.

2. **Add DEVSshapr project.** Since we need to use `DEVSsharp` library, we should add it by clicking right mouse button to 'MyProject' at Solution Explorer and then select context menu through `Add->Existing Project...` as shown in Figure 5.5. A file selection dialog will pop up, so we will select `DEVSsharp\DEVSsharp.csproj` that is the project file of `DEVSsharp`.

3. **Add DEVSsharp as a MyProject's reference**. Now we need to add `DEVSshap` project to `MyProject` as a reference. To do this, first left-button click MyProject at Solution Explorer and then select title menu

Figure 5.2: Screen Capture of Visual Studio 2005^{TM} when opening DEVSsharp.sln

of `Project->Add Reference...` or right-button click for context menu of `Add Reference....`

We can see a dialog box titled as Add Reference as shown in Figure 5.7. We select Projects tab and select `DEVSsharp` and press OK button. Then `MyPoroject`'s References at Solution Explorer becomes to contain `DEVSsharp`.

The following codes shows a simple example which are mainly copied from `Ex_Timer` project. Don't forget to add the statement of `using DEVSsharp`.

```
using System;
using System.Collections.Generic;
using System.Text;

using DEVSsharp; //<-- Don't forget add this statement
```

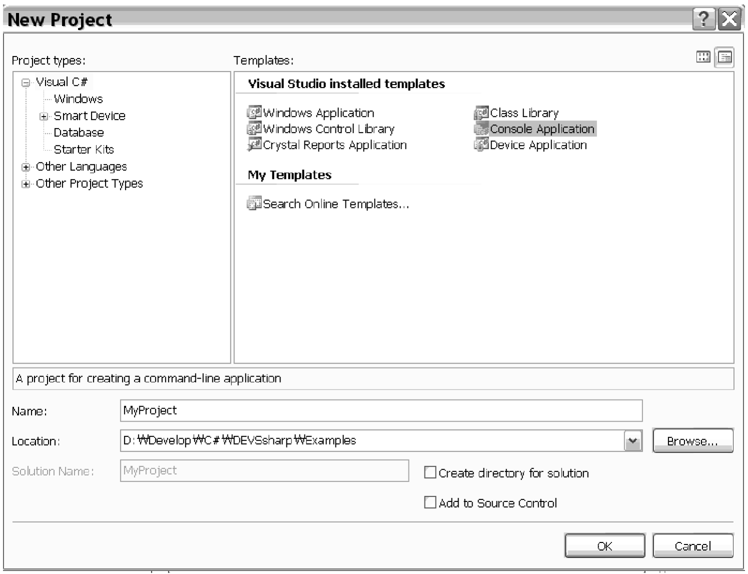

Figure 5.3: New Project Dialog

```
namespace MyProject
{
    public class Timer : Atomic
    {
        private OutputPort op;

        public Timer(string name)
            : base(name, TimeUnit.Sec)
        { op = AddOP("op"); init(); }
        public override void init() { }
        public override double tau() { return 3.3; }
        public override bool delta_x(PortValue x) { return false; }
        public override void delta_y(ref PortValue y) { y.Set(op); }
        public override string Get_s() { return "Working"; }
    }

    class Program
    {
        static void Main(string[] args)
        {
            Timer timer = new Timer("STimer");
            SRTEngine Engine = new SRTEngine(timer, 10000, null);
```

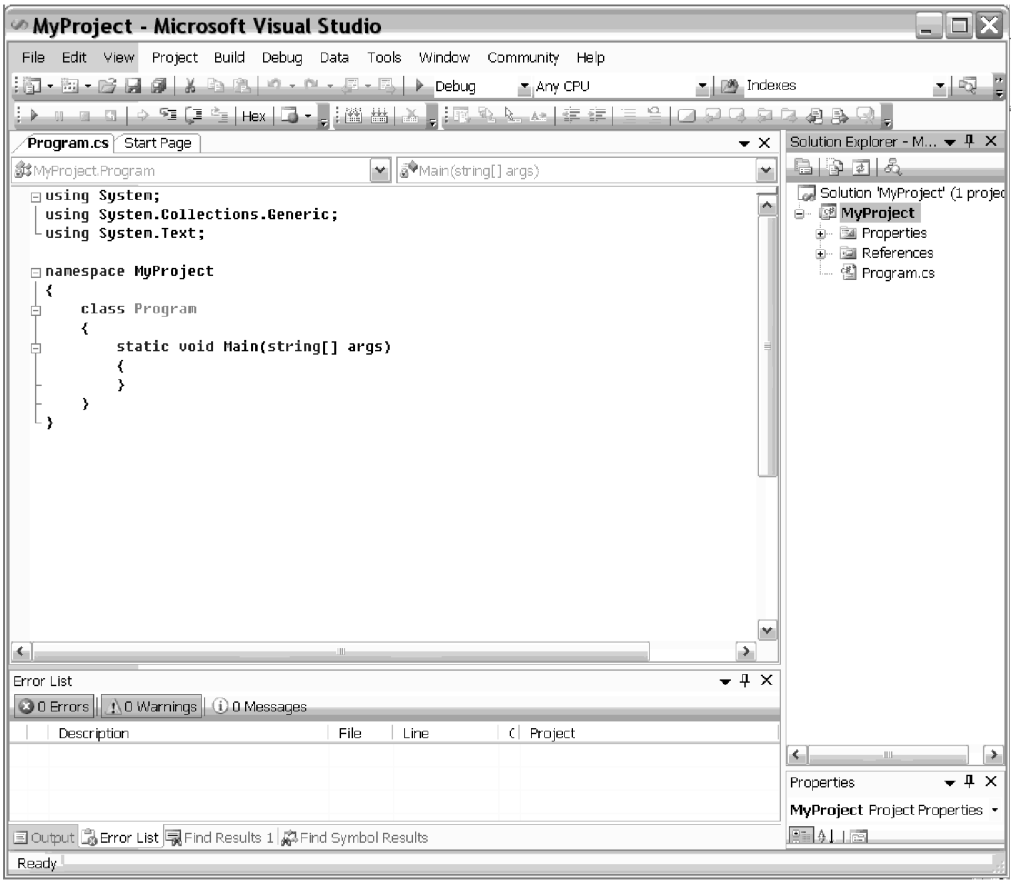

Figure 5.4: My Project

```
                Engine.RunConsoleMenu();
            }
        }
    }
```

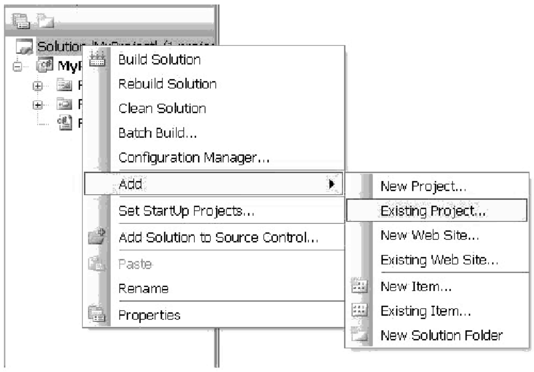

Figure 5.5: Menu Selection of Add Existing Project...

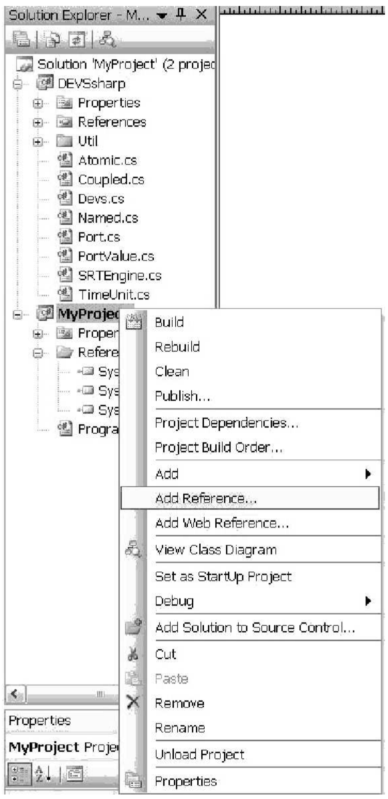

Figure 5.6: Menu Selection of Add Reference...

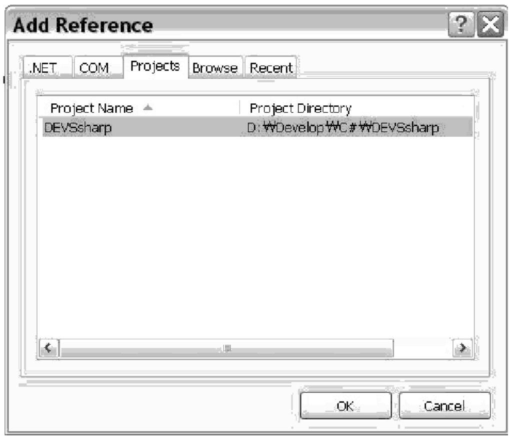

Figure 5.7: Dialog of Add Reference

Bibliography

[Hwa07] Moon Ho Hwang. *DEVS++: C++ Open Source Library of DEVS Formalism.* http://odevspp.sourceforge.net/, first edition, Apr 2007.

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

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

[Zei87] Bernard P. Zeigler. Hierarchical, modular discrete-event modelling in an object-oriented environment. *SIMULATION*, 49(5):219–230, 1987.

[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.