A Discrete EVent system Simulator

Jim Nutaro

Contents

Chapter 1

Building and Installing

The Adevs package is organized into the following directory structure:

```
adevs-x.y.z
           +->docs
           +->examples
           +->include
           +->src
           +->test
           +->util
```

Everything except the random number generators are implemented as template classes, and so are contained entirely in the header files that are located in the *include* directory. If you do not want to use the random number generators, its sufficient to include *adevs.h* in your source code, and to make sure that your compiler can find the *include* directory.

If you do want to use the random number generators, then enter the the src directory and run 'make'. This will build the library *libadevs.a* that can be linked with your executable.

If you want to run the test suite, then first you need to build the library file and install Tcl (the test scripts need Tcl to run; if you can run 'tclsh' then you already have a working copy of Tcl). After that, go the the *test* directory and run 'make check'. This will automatically build and execute all of the test cases. If the test suite is run to completion, then everything works fine. If something goes wrong, then make will exit and report an error. Use 'make clean' to cleanup afterward.

Chapter 2

Modeling and simulation with Adevs

Adevs is a simulator for models described in terms of the Discrete Event System Specification (DEVS)[1] The key feature of models described in DEVS (and implemented in Adevs) is that their dynamic behavior is defined by events. An event can be any kind of change that is significant within the contex of the model being developed.

Modeling of discrete event system modeling can be most easily introduced with an example. Suppose that we want to model the checkout line at a convenience store. There is a single clerk who serves customers in a first come-first serve fashion. Each customer has a different number of items, and so they require more or less time for the clerk to ring up their bill. We are interested in determining the average and maximum amount of time that customers spend waiting in line.

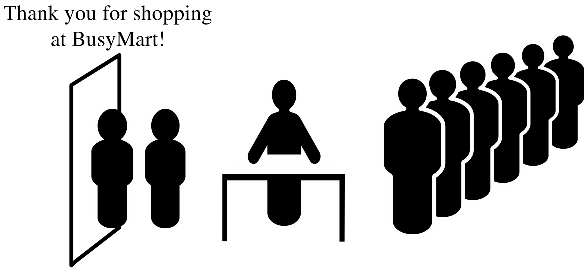

Figure 2.1: Customers waiting in line at BusyMart.

To simulate this system, we will need an object to represent each customer in the line. A **Customer** class is created for this purpose. Customer objects have three attributes. One attribute is the time needed to ring up the customer's bill. Since we want to be able to determine how long a customer has been waiting in line, we will also include two attributes that record the time at which the customer entered the queue and the time that the customer left the queue. The difference of these times is the amount of time that the customer spent waiting in line. Here is the customer class, from which we will create customer objects as needed. The class is coded in a single header file, Customer.h.

```
#include "adevs.h"
/// A Busy-Mart customer.
struct Customer
{
    /// Time needed for the clerk to process the customer
    double twait;
```

---

[1] A comprehensive introduction to the Discrete Event System Specification can be found in "Theory of Modeling and Simulation, 2nd Edition" by Bernard Zeigler *et. al.*, published by Academic Press in 2000.

```
    /// Time that the customer entered and left the queue
    double tenter, tleave;
};
/// Create an abbreviation for the Clerk's input/output type.
typedef adevs::PortValue<Customer*> IO_Type;
```

The model of the clerk is our first example of an atomic model. Fortunately, the clerk's behavior is very simple. The clerk has a line of people waiting at her counter. When a customer arrives at the clerk's counter, that person is placed at the end of the line. If the clerk is not busy and somebody is waiting in line then the clerk rings up that customer's bill and sends the customer on his way. The clerk then looks for another customer waiting in line. If there is a customer, the clerk proceeds as before. Otherwise, the clerk sits idly at her counter waiting for more customers.

The DEVS model of the clerk is described in a particular way. First, we need to specify the type of object that the clerk can consume and produce. For this model, we use **PortValue** objects. The **PortValue** class is part of the **Digraph** model class, which will be introduced later. The **PortValue** class describes a port/value pair. Suppose that customers arrive in line via an "arrive" port and leave via a "depart" port and the value objects are instances of the **Customer** class.

The second thing that we need to determine are the state variables that describe the clerk. In this case, we need to know which customers are in line. This can be described with a list of customers; we can use a **list** from the C++ Standard Template Library.

To complete the model of the clerk, we must implement four methods that model the clerk's behavior. First, let's construct the header file for the clerk. Then we can proceed to fill in the details.

```
#include "adevs.h"
#include "Customer.h"
#include <list>
/**
 * The Clerk class is derived from the adevs Atomic class.
 * The Clerk's input/output type is specified using the template
 * parameter of the base class.
 */
class Clerk: public adevs::Atomic<IO_Type>
{
    public:
        /// Constructor.
        Clerk();
        /// Internal transition function.
        void delta_int();
        /// External transition function.
        void delta_ext(double e, const adevs::Bag<IO_Type>& xb);
        /// Confluent transition function.
        void delta_conf(const adevs::Bag<IO_Type>& xb);
        /// Output function.
        void output_func(adevs::Bag<IO_Type>& yb);
        /// Time advance function.
        double ta();
        /// Output value garbage collection.
        void gc_output(adevs::Bag<IO_Type>& g);
        /// Destructor.
        ~Clerk();
        /// Model input port.
        static const int arrive;
        /// Model output port.
        static const int depart;
    private:
        /// The clerk's clock
        double t;
        /// List of waiting customers.
        std::list<Customer*> line;
        /// Time spent so far on the customer at the front of the line
        double t_spent;
};
```

This header file is a template for almost any atomic model that we want to create. The **Clerk** class is derived from the Adevs **Atomic** class, and it defines the virtual state transition, output, time advance, and garbage collection methods that are required by the **Atomic** base class. The **Clerk** also includes a set of static, constant port variables that correspond to the **Clerk**'s input (customer arrival) and output (customer departure) ports.

The constructor for the **Clerk** class invokes the constructor of the **Atomic** base class. The template argument of the base class is used to define the **Clerk**'s input/output type. The **Clerk** state variables are defined as private class attributes. The variables arrive and depart are assigned integer values that are unique within the scope of the **Clerk** class. Typically, the ports for a model are numbered in a way that corresponds with the order in which they are listed; for example,

```
// Assign locally unique identifiers to the ports
const int Clerk::arrive = 0;
const int Clerk::depart = 1;
```

The **Clerk** constructor places the **Clerk** into its initial state. For our experiment, this state is an empty line and the **Clerk**'s clock is initialized to zero.

```
Clerk::Clerk():
Atomic<IO_Type>(), // Initialize the parent Atomic model
t(0.0), // Set the clock to zero
t_spent(0.0) // No time spent on a customer so far
{
}
```

Because the clerk has an empty line at first, the only interesting thing that can happen is for a customer arrive. Customer arrivals are events that appear on the clerk's "arrive" input port. The arrival of a customer will cause the clerk's external transition method to be activated. The arguments to the method are the time that has elapsed since the clerk last changed state and a bag of **PortValue** objects.

The clerk's local clock is updated by adding the elapsed time to the current value of the clock. Next, the time spent working on the current customer's order is updated by adding the elapsed time to the time spent so far. After doing this, the input events are processed. Each **PortValue** object has two fields. The first is the port field; it contains the number of the port that the event arrived on. The second is the **Customer** that arrived. The arrival time of every arriving customer is recorded and then the customer is placed at the back of the line.

```
void Clerk::delta_ext(double e, const Bag<IO_Type>& xb)
{
    // Print a notice of the external transition
    cout << "Clerk: Computed the external transition function at t = " << t+e << endl;
    // Update the clock
    t += e;
    // Update the time spent on the customer at the front of the line
    if (!line.empty())
    {
        t_spent += e;
    }
    // Add the new customers to the back of the line.
    Bag<IO_Type>::const_iterator i = xb.begin();
    for (; i != xb.end(); i++)
    {
        // Copy the incoming Customer and place it at the back of the line.
        line.push_back(new Customer(*((*i).value)));
        // Record the time at which the customer entered the line.
        line.back()->tenter = t;
    }
    // Summarize the model state
    cout << "Clerk: There are " << line.size() << " customers waiting." << endl;
    cout << "Clerk: The next customer will leave at t = " << t+ta() << "." << endl;
}
```

The time advance function gives the time at which the clerk will finish processing its current customer. The time advance function describes the amount of time that should elapsed before the clerk's next internal (or self) event, barring an input that arrives in the interim. The clerk's time advance function is very simple. If there are no customers in line, then the clerk will not do anything in the abscence of input, and so the time advance function returns infinity (here, represented by DBL_MAX). Otherwise, the clerk will wait until the first customer has been rung up; i.e., until the difference of the **Customer**'s twait and t_spent has elapsed.

```
double Clerk::ta()
{
    // If the list is empty, then next event is at inf
    if (line.empty()) return DBL_MAX;
    // Otherwise, return the time remaining to process the current customer
    return line.front()->twait-t_spent;
}
```

Eventually, the clerk will be done ringing up the customer. At this time, the clerk sends the customer on his way and looks for a new customer in the line. If there is another customer waiting in line, the clerk will begin ringing that customer up in the same fashion as before. This will all occur when the simulation clock reaches the clerk's time of next event; i.e., the time of the clerk's last event plus the time advance.

Two things happen when the time advance expires. First, the **Clerk**'s output method is called. When this happens, the **Clerk** places the departing customer on its "depart" output port. Next, the **Clerk**'s internal transition method is activated. The **Clerk**'s internal transition method changes the state of the **Clerk** by removing the departed customer from the line. The output function and internal transition function are shown below.

```
void Clerk::delta_int()
{
    // Print a notice of the internal transition
    cout << "Clerk: Computed the internal transition function at t = " << t+ta() << endl;
    // Update the clock
    t += ta();
    // Reset the spent time
    t_spent = 0.0;
    // Remove the departing customer from the front of the line.
    line.pop_front();
    // Summarize the model state
    cout << "Clerk: There are " << line.size() << " customers waiting." << endl;
```

Enter checkout line	Time to process order
1	1
2	4
3	4
5	2
7	10
8	20
10	2
11	1

Table 2.1: Customer arrival times and time to process customer orders.

```
    cout << "Clerk: The next customer will leave at t = " << t+ta() << "." << endl;
}

void Clerk::output_func(Bag<IO_Type>& yb)
{
    // Get the departing customer
    Customer* leaving = line.front();
    // Set the departure time
    leaving->tleave = t + ta();
    // Eject the customer
    IO_Type y(depart,leaving);
    yb.insert(y);
    // Print a notice of the departure
    cout << "Clerk: Computed the output function at t = " << t+ta() << endl;
    cout << "Clerk: A customer just departed!" << endl;
}
```

At this point we have almost completely defined the behavior of the model clerk; only one thing remains to be done. Suppose that a customer arrives at the clerk's line at the same time that the clerk has finished ringing up a customer. In this case we have a conflict because the internal transition function and external transition function must both be activated to handle these two events (i.e., the simultaneous arriving customer and departing customer). These types of conflicts are resolved by the confluent transition function. For the clerk model, the confluent transition function is computed using the internal transition function first (to remove the newly departed customer from the list) followed by the external transition function (to add new customers to the end of the list and begin ringing up the first customer). Below is the implementation of the clerk's confluent transition function.

```
void Clerk::delta_conf(const Bag<IO_Type>& xb)
{
    delta_int();
    delta_ext(0.0,xb);
}
```

To see how this model behaves, suppose we had customers arrive according to the schedule shown in the table below. In this example, the first customer appears on the clerk's "arrive" port at time 1, the next customer appears on the "arrive" port at time 2, and so on. The print statements in the **Clerk**'s internal, external, and output functions let us watch the evolution of the clerk's line. Here is the output trace produced by the above sequence of inputs.

```
Clerk: Computed the external transition function at t = 1
Clerk: There are 1 customers waiting.
```

```
Clerk: The next customer will leave at t = 2.
Clerk: Computed the output function at t = 2
Clerk: A customer just departed!
Clerk: Computed the internal transition function at t = 2
Clerk: There are 0 customers waiting.
Clerk: The next customer will leave at t = 1.79769e+308.
Clerk: Computed the external transition function at t = 2
Clerk: There are 1 customers waiting.
Clerk: The next customer will leave at t = 6.
Clerk: Computed the external transition function at t = 3
Clerk: There are 2 customers waiting.
Clerk: The next customer will leave at t = 6.
Clerk: Computed the external transition function at t = 5
Clerk: There are 3 customers waiting.
Clerk: The next customer will leave at t = 6.
Clerk: Computed the output function at t = 6
Clerk: A customer just departed!
Clerk: Computed the internal transition function at t = 6
Clerk: There are 2 customers waiting.
Clerk: The next customer will leave at t = 10.
Clerk: Computed the external transition function at t = 7
Clerk: There are 3 customers waiting.
Clerk: The next customer will leave at t = 10.
Clerk: Computed the external transition function at t = 8
Clerk: There are 4 customers waiting.
Clerk: The next customer will leave at t = 10.
Clerk: Computed the output function at t = 10
Clerk: A customer just departed!
Clerk: Computed the internal transition function at t = 10
Clerk: There are 3 customers waiting.
Clerk: The next customer will leave at t = 12.
Clerk: Computed the external transition function at t = 10
Clerk: There are 4 customers waiting.
Clerk: The next customer will leave at t = 12.
Clerk: Computed the external transition function at t = 11
Clerk: There are 5 customers waiting.
Clerk: The next customer will leave at t = 12.
Clerk: Computed the output function at t = 12
Clerk: A customer just departed!
Clerk: Computed the internal transition function at t = 12
Clerk: There are 4 customers waiting.
Clerk: The next customer will leave at t = 22.
Clerk: Computed the output function at t = 22
Clerk: A customer just departed!
Clerk: Computed the internal transition function at t = 22
Clerk: There are 3 customers waiting.
Clerk: The next customer will leave at t = 42.
Clerk: Computed the output function at t = 42
Clerk: A customer just departed!
Clerk: Computed the internal transition function at t = 42
Clerk: There are 2 customers waiting.
Clerk: The next customer will leave at t = 44.
```

```
Clerk: Computed the output function at t = 44
Clerk: A customer just departed!
Clerk: Computed the internal transition function at t = 44
Clerk: There are 1 customers waiting.
Clerk: The next customer will leave at t = 45.
Clerk: Computed the output function at t = 45
Clerk: A customer just departed!
Clerk: Computed the internal transition function at t = 45
Clerk: There are 0 customers waiting.
Clerk: The next customer will leave at t = 1.79769e+308.
```

The basic simulation algorithm is illustrated by this example. Notice that the external transition function is always activated when an input (in this case, a customer) arrives on an input port. This is because the external transition function describes the response of the model to input events.

The internal transition function is always activated when the simulation clock has reached the model's time of next event. The internal transition function describes the autonomous behavior of the system (i.e., how the system responds to events that it has scheduled for itself). Internal transitions are scheduled with the time advance function.

The internal transition function is always immediately preceded by the output function. Consequently, a model can only produce outputs by scheduling an event for itself. The value of the output is computed using the current state of the model.

To complete our simulation of the convenience store, we need two other **Atomic** models. The first model produce customers for the **Clerk** to serve. The customer arrival rate could be modeled using a random variable with appropriate statistics, or it could be driven by a table of values such as the one used in the previous example. In either case, we hope that the customer arrival process accurately reflects what happens in a typical day at the convenience store. For this example, we will use a table to drive the customer arrival process. Data for this table could come directly from observing customers at the store, or it might be produced by a statistical model in another tool (e.g., a spreadsheet program).

We will use an **Atomic** model called a **Generator** to create customer arrival events. The input file format is identical to that used in the previous example. The input file contains a line for each customer that arrives. Each line has the customer arrival time first, followed by the customer service time. The **Generator** is an input free **Atomic** model since all of its events are scripted in the input file. The **Generator** will need a single output port, which we will call "arrive", through which is can export arriving customers. The model state is the list of **Customer**s that will arrive at the store. Here is the header file for the **Generator**.

```
#include "adevs.h"
#include "Customer.h"
#include <list>
/**
 * This class produces Customers according to the provided schedule.
 */
class Generator: public adevs::Atomic<IO_Type>
{
    public:
        /// Constructor.
        Generator(const char* data_file);
        /// Internal transition function.
        void delta_int();
        /// External transition function.
        void delta_ext(double e, const adevs::Bag<IO_Type>& xb);
        /// Confluent transition function.
        void delta_conf(const adevs::Bag<IO_Type>& xb);
        /// Output function.
        void output_func(adevs::Bag<IO_Type>& yb);
        /// Time advance function.
        double ta();
        /// Output value garbage collection.
        void gc_output(adevs::Bag<IO_Type>& g);
        /// Destructor.
        ~Generator();
        /// Model output port.
        static const int arrive;

    private:
        /// List of arriving customers.
        std::list<Customer*> arrivals;
};
```

The dynamic behavior of this model is very simple. The constructor opens the file containing the customer data and uses it to create a list of **Customer** objects. The inter-arrival times of the customers are stored in their tenter fields. Here is the constuctor that initializes the model.

```
// Assign a locally unique number to the arrival port
const int Generator::arrive = 0;

Generator::Generator(const char* sched_file):
Atomic<IO_Type>()
{
    // Open the file containing the schedule
    fstream input_strm(sched_file);
    // Store the arrivals in a list
    double next_arrival_time = 0.0;
    double last_arrival_time = 0.0;
    while (true)
    {
        Customer* customer = new Customer;
        input_strm >> next_arrival_time >> customer->twait;
        // Check for end of file
        if (input_strm.eof())
        {
            delete customer;
            break;
        }
        // The entry time holds the inter arrival times, not the
        // absolute entry time.
        customer->tenter = next_arrival_time-last_arrival_time;
        // Put the customer at the back of the line
        arrivals.push_back(customer);
        last_arrival_time = next_arrival_time;
    }
}
```

Because the generator is input free, the external transition function is empty. Similarly, the confluent transition function merely calls the internal transition function.

```
void Generator::delta_ext(double e, const Bag<IO_Type>& xb)
{
    /// The generator is input free, and so it ignores external events.
}

void Generator::delta_conf(const Bag<IO_Type>& xb)
{
    /// The generator is input free, and so it ignores input.
    delta_int();
}
```

The effect of an internal event (i.e., an event scheduled for the generator by itself) is to first place the arriving **Customer** onto the **Generator**'s "arrive" output port. This is done by the output function.

```
void Generator::output_func(Bag<IO_Type>& yb)
{
    // First customer in the list is produced as output
    IO_Type output(arrive,arrivals.front());
    yb.insert(output);
}
```

After the generator has produced this output event, its internal transition function removes the newly arrived customer from the arrival list.

```
void Generator::delta_int()
{
    // Remove the first customer.  Because it was used as the
    // output object, it will be deleted during the gc_output()
    // method call at the end of the simulation cycle.
    arrivals.pop_front();
}
```

Internal events are scheduled with the time advance function. The **Generator**'s time advance function returns the time remaining until the next **Customer** arrives at the store. Remember that the tarrival field contains **Customer**'s the inter-arrival times, not the absolute arrival times.

```
double Generator::ta()
{
    // If there are not more customers, next event time is infinity
    if (arrivals.empty()) return DBL_MAX;
    // Otherwise, wait until the next arrival
    return arrivals.front()->tenter;
}
```

To conduct the simulation experiment, the **Generator** output is coupled to the **Clerk** input. By doing this, **Customer** objects appearing on the **Generator**'s "arrive" output port cause a corresponding appearance of a **Customer** on the **Clerk**'s "arrive" input port. This input event, in turn, causes the **Clerk**'s external transition function to be activated. The relationship between input and output events can be best understood by viewing the whole model as two distinct components, the **Generator** and the **Clerk**, that are connected via their input and output ports. This view of the model is depicted in Figure 2.2.

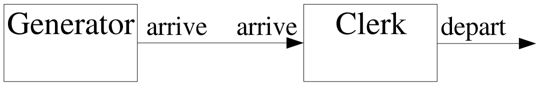

Figure 2.2: The combined **Generator** and **Clerk** model.

As Figure 2.2 suggests, output events produced by the **Generator** on its "arrive" port, via the output function, will appear as input events on the **Clerk**'s "arrive" port when the **Clerk**'s external transition function is evaluated. The component models and their interconnections constitute a coupled (or network) model. To create the coupled model depicted above, we need to create an instance of a **Digraph** model that has the **Generator** and **Clerk** as component models. Shown below is the code snippet that creates this two component model.

```
int main(int argc, char** argv)
{
    ...
    // Create a digraph model whose components use PortValue<Customer*>
    // objects as input and output objects.
    adevs::Digraph<Customer*> store;
    // Create and add the component models
    Clerk* clrk = new Clerk();
    Generator* genr = new Generator(argv[1]);
    store.add(clrk);
    store.add(genr);
    // Couple the components
    store.couple(genr,genr->arrive,clrk,clrk->arrive);
    ...
```

This code snippet first creates the components models and then adds them to the **Digraph**. Next, the components are interconnected by coupling the "arrive" output port of the **Generator** to the "arrive" input port of the **Clerk**.

Having created a coupled model which represents the store, all that remains is to perform the simulation. Here is the code necessary to simulate our model.

```
adevs::Simulator<IO_Type> sim(&store);
while (sim.nextEventTime() < DBL_MAX)
{
    sim.execNextEvent();
}
```

Putting this all of this together gives the main routine for the simulation program that will generate the execution traces that are shown in the above examples.

```
#include "Clerk.h"
#include "Generator.h"
#include "Observer.h"
#include <iostream>
using namespace std;

int main(int argc, char** argv)
{
    if (argc != 3)
    {
        cout << "Need input and output files!" << endl;
        return 1;
    }
    // Create a digraph model whose components use PortValue<Customer*>
    // objects as input and output objects.
    adevs::Digraph<Customer*> store;
    // Create and add the component models
    Clerk* clrk = new Clerk();
    Generator* genr = new Generator(argv[1]);
    Observer* obsrv = new Observer(argv[2]);
    store.add(clrk);
    store.add(genr);
    store.add(obsrv);
    // Couple the components
    store.couple(genr,genr->arrive,clrk,clrk->arrive);
    store.couple(clrk,clrk->depart,obsrv,obsrv->departed);
    // Create a simulator and run until its done
    adevs::Simulator<IO_Type> sim(&store);
    while (sim.nextEventTime() < DBL_MAX)
    {
        sim.execNextEvent();
    }
    // Done, component models are deleted when the Digraph is
    // deleted.
    return 0;
}
```

We have completed our first Adevs simulation program! However, a few details have been glossed over. The first question, an essential one for a programming language without garbage collection, is what happens to the objects that we created in the **Generator** and **Clerk** output functions?

The answer is that each model has a garbage collection method that is called at the end of every simulation cycle. The argument to the garbage collection method is the bag of objects created as output in the current simulation cycle. In our store example, the **Atomic** models simply delete the customer pointed to by each **PortValue** object in the garbage list. The implementation of the garbage collection method is shown below. This listing is for the **Generator** model; the **Clerk**'s ***gc_output()*** method is identical.

```
void Generator::gc_output(Bag<IO_Type>& g)
{
    // Delete the customer that was produced as output
    Bag<IO_Type>::iterator i;
    for (i = g.begin(); i != g.end(); i++)
    {
        delete (*i).value;
    }
}
```

A second issue that has been overlooked is how to collect the statistics that were our original objective. One approach is to modify the **Clerk** so that it writes waiting times to a file as **Customer**s are processed. While this could work, it has the unfortunate effect of cluttering up the **Clerk** with experiment specific code.

A better approach is to have an **Observer** that is coupled to the **Clerk**'s "depart" output port. The **Observer** can record the desired statistics as it receives **Customer**s on its "depart" input port. The advantage of this approach is that we can modify the **Clerk** model to perform the same experiment on different queueing strategies (e.g., we could add a priority to each customer and have the clerk process customers with a high priority first) without changing the experimental setup (i.e., customer generation and data collection). We can also change the experiment (i.e., customer generation and data collection) without changing the clerk.

Below is a listing of the **Observer** class. The model is driven solely by external events. The effect of an external event is simply to have the model record the time that the **Customer** departed the **Clerk**'s queue (i.e., the current simulation time) and the amount of time that the **Customer** waited in line. Here is the **Observer** header file.

```
#include "adevs.h"
#include "Customer.h"
#include <fstream>
/**
 * The Observer records performance statistics for a Clerk model
 * based on its observable output.
 */
class Observer: public adevs::Atomic<IO_Type>
{
    public:
        /// Input port for receiving customers that leave the store.
        static const int departed;
        /// Constructor. Results are written to the specified file.
        Observer(const char* results_file);
        /// Internal transition function.
        void delta_int();
        /// External transition function.
        void delta_ext(double e, const adevs::Bag<IO_Type>& xb);
        /// Confluent transition function.
        void delta_conf(const adevs::Bag<IO_Type>& xb);
        /// Time advance function.
        double ta();
        /// Output function.
        void output_func(adevs::Bag<IO_Type>& yb);
        /// Output value garbage collection.
        void gc_output(adevs::Bag<IO_Type>& g);
        /// Destructor.
        ~Observer();
    private:
        /// File for storing information about departing customers.
        std::ofstream output_strm;
};
```

Below is the **Observer** source file.

```
#include "Observer.h"
using namespace std;
using namespace adevs;

// Assign a locally unique number to the input port
const int Observer::departed = 0;

Observer::Observer(const char* output_file):
Atomic<IO_Type>(),
output_strm(output_file)
{
    // Write a header describing the data fields
    output_strm << "# Col 1: Time customer enters the line" << endl;
    output_strm << "# Col 2: Time required for customer checkout" << endl;
    output_strm << "# Col 3: Time customer leaves the store" << endl;
    output_strm << "# Col 4: Time spent waiting in line" << endl;
}
```

```
double Observer::ta()
{
    // The Observer has no autonomous behavior, so its next event
    // time is always infinity.
    return DBL_MAX;
}

void Observer::delta_int()
{
    // The Observer has no autonomous behavior, so do nothing
}

void Observer::delta_ext(double e, const Bag<IO_Type>& xb)
{
    // Record the times at which the customer left the line and the
    // time spent in it.
    Bag<IO_Type>::const_iterator i;
    for (i = xb.begin(); i != xb.end(); i++)
    {
        const Customer* c = (*i).value;
        // Compute the time spent waiting in line
        double waiting_time = (c->tleave-c->tenter)-c->twait;
        // Dump stats to a file
        output_strm << c->tenter << " " << c->twait << " " << c->tleave << " " << waiting_time << endl;
    }
}

void Observer::delta_conf(const Bag<IO_Type>& xb)
{
    // The Observer has no autonomous behavior, so do nothing
}

void Observer::output_func(Bag<IO_Type>& yb)
{
    // The Observer produces no output, so do nothing
}

void Observer::gc_output(Bag<IO_Type>& g)
{
    // The Observer produces no output, so do nothing
}

Observer::~Observer()
{
    // Close the statistics file
    output_strm.close();
}
```

This model is coupled to the **Clerk**'s "depart" output port in the same manner as before. The resulting coupled model is illustrated in Figure 2.3; now we have three components instead of just two.

Given the customer arrival data in Table 2.1, the corresponding customer depature and waiting times are shown in Table 2.2. Given this output, we could use a spreadsheet or some other suitable software to

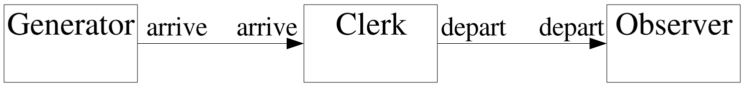

Figure 2.3: The **Generator**, **Clerk**, and **Observer** model.

Time that the customer leaves the store	Time spent waiting in line
2	0
6	0
10	3
12	5
22	5
42	14
44	32
45	33

Table 2.2: Customer departure times and wait times.

find the maximum and average customer wait times.

Again, notice that the customer depature times correspond exactly with the production of customer depature events by the **Clerk** model. These customer depature events are delivered to the **Observer** via the **Clerk** to **Observer** coupling shown in Figure 2.3. Each entry in Table 2.2 is the result of executing the **Observer**'s external transition function. Also notice that the **Observer**'s internal and confluent transition functions will never be executed. This is because the **Observer**'s ***ta()*** method always returns infinity.

This section has demonstrated the most common parts of a simulation program that is built with Adevs. The remainder of the manual covers **Atomic** and **Network** models in greater detail, demonstrates the construction of variable structure models, and shows how continuous models can be added to your discrete event simulation.

Chapter 3

Atomic Models

Atomic models are the basic building blocks of a DEVS model. The behavior of an atomic model is described by its state transition functions (internal, external, and confluent), its output function, and its time advance function. Within Adevs, these aspects of an atomic model are implemented by sub-classing the **Atomic** class and implementing the pure virtual methods that correspond to the internal, external, confluent, output, and time advance functions.

The state of an atomic model is represented by the attributes of the class that implements the model. The internal transition function describes how the state evolves in the absence of input. The time advance function is used to schedule internal changes of state, and the output function gives the model output when these internal events occur. The external transition function describes how the system state changes in response to input. The confluent transition function handles the simultaneous occurrence of an internal and external event. The types of objects that can be accepted as input and output are specified with a template argument to the **Atomic** base class.

The **Clerk** described in Section 2 demonstrates all of the aspects of an **Atomic** model. We'll use it to demonstrate how an every **Atomic** model generates output, processes input events, and schedules self-events. Here is the **Clerk**'s class definition:

```
include "adevs.h"
#include "Customer.h"
#include <list>

class Clerk: public adevs::Atomic<IO_Type>
{
    public:
        /// Constructor.
        Clerk();
        /// Internal transition function.
        void delta_int();
        /// External transition function.
        void delta_ext(double e, const adevs::Bag<IO_Type>& xb);
        /// Confluent transition function.
        void delta_conf(const adevs::Bag<IO_Type>& xb);
        /// Output function.
        void output_func(adevs::Bag<IO_Type>& yb);
        /// Time advance function.
        double ta();
        /// Output value garbage collection.
        void gc_output(adevs::Bag<IO_Type>& g);
        /// Destructor.
        ~Clerk();
        /// Model input port.
        static const int arrive;
        /// Model output port.
        static const int depart;

    private:
        /// The clerk's clock
        double t;
        /// List of waiting customers.
        std::list<Customer*> line;
        /// Time spent so far on the customer at the front of the line
        double t_spent;
};
```

and here its implementation

```
#include "Clerk.h"
#include <iostream>
using namespace std;
using namespace adevs;

// Assign locally unique identifiers to the ports
const int Clerk::arrive = 0;
const int Clerk::depart = 1;

Clerk::Clerk():
Atomic<IO_Type>(), // Initialize the parent Atomic model
t(0.0), // Set the clock to zero
t_spent(0.0) // No time spent on a customer so far
{
}

void Clerk::delta_ext(double e, const Bag<IO_Type>& xb)
{
    // Print a notice of the external transition
    cout << "Clerk: Computed the external transition function at t = " << t+e << endl;
    // Update the clock
    t += e;
    // Update the time spent on the customer at the front of the line
    if (!line.empty())
    {
        t_spent += e;
    }
    // Add the new customers to the back of the line.
    Bag<IO_Type>::const_iterator i = xb.begin();
    for (; i != xb.end(); i++)
    {
        // Copy the incoming Customer and place it at the back of the line.
        line.push_back(new Customer(*((*i).value)));
        // Record the time at which the customer entered the line.
        line.back()->tenter = t;
    }
    // Summarize the model state
    cout << "Clerk: There are " << line.size() << " customers waiting." << endl;
    cout << "Clerk: The next customer will leave at t = " << t+ta() << "." << endl;
}

void Clerk::delta_int()
{
    // Print a notice of the internal transition
    cout << "Clerk: Computed the internal transition function at t = " << t+ta() << endl;
    // Update the clock
    t += ta();
    // Reset the spent time
    t_spent = 0.0;
    // Remove the departing customer from the front of the line.
    line.pop_front();
    // Summarize the model state
    cout << "Clerk: There are " << line.size() << " customers waiting." << endl;
    cout << "Clerk: The next customer will leave at t = " << t+ta() << "." << endl;
}

void Clerk::delta_conf(const Bag<IO_Type>& xb)
{
    delta_int();
    delta_ext(0.0,xb);
}

void Clerk::output_func(Bag<IO_Type>& yb)
{
    // Get the departing customer
    Customer* leaving = line.front();
    // Set the departure time
    leaving->tleave = t + ta();
    // Eject the customer
    IO_Type y(depart,leaving);
    yb.insert(y);
    // Print a notice of the departure
    cout << "Clerk: Computed the output function at t = " << t+ta() << endl;
    cout << "Clerk: A customer just departed!" << endl;
}

double Clerk::ta()
{
    // If the list is empty, then next event is at inf
    if (line.empty()) return DBL_MAX;
    // Otherwise, return the time remaining to process the current customer
    return line.front()->twait-t_spent;
}

void Clerk::gc_output(Bag<IO_Type>& g)
{
    // Delete the outgoing customer objects
    Bag<IO_Type>::iterator i;
```

Enter checkout line	Time to process order
1	1
2	4
3	4
5	2
7	10
8	20
10	2
11	1

Table 3.1: Customer arrival times and time to process customer orders.

```
    for (i = g.begin(); i != g.end(); i++)
    {
        delete (*i).value;
    }
}

Clerk::~Clerk()
{
    // Delete anything remaining in the customer queue
    list<Customer*>::iterator i;
    for (i = line.begin(); i != line.end(); i++)
    {
        delete *i;
    }
}
```

Consider a simulation of the store with the same sequence of customer arrivals that were used in Section 2 (i.e., listed in Table 2.1); I've listed the data again here:

Table 3.1 describes an input sequence that is fed to the **Clerk** model. The algorithm for processing this, or any other, input sequence is listed below. The **Atomic** model that is being simulated is called 'model', t is the current simulation time (i.e., the last event time), and t_input is the time stamp of the smallest unprocessed event in the input sequence.

1. Set the next event time tN to the smaller of the next internal event time t_self = t + model.ta() and the next input event time t_input.
2. If t_self = tN and t_input ¡ tN then produce an output event at time t_self by calling model.output_func() and then compute the next state by calling model.delta_int().
3. If t_self = t_input = tN then produce an output event at time t_self by calling model.output_func() and compute the next state by calling model.delta_conf(x) where x contains the input events scheduled at time t_input.
4. If t_self ¡ tN and t_input = tN then compute the next state by calling model.delta_ext(t_input-t,x), where x contains the input events schedule at time t_input.
5. Set t equal to tN.
6. Repeat if there are more input or self events to process.

The simulation runs until there are no internal or external events to process. The first step of the algorithm computes the next event time by taking the smaller of the next input event time and the next self event time. If the next self event happens first, then the model produces an output and its next state is computed with the internal transition function. If the next input event happens first, then the next state of the model is computed with the external transition function; no output event is produces in this case. The elapsed time argument to the external transition function is the amount of time that has passed since the previous event at that model. If the next input and self event happen at the same time, then the model produces an output and the next model state is computed with the confluent transition function. The simulation clock is then advanced to the event time and these steps are repeated.

The execution trace resulting from the customer arrival sequence in Table 3.1 is shown below. It has been broken up to show where each simulation cycle begins and ends and the type of event occurring in each cycle.

```
-External event----------------------------------------------
Clerk: Computed the external transition function at t = 1
Clerk: There are 1 customers waiting.
Clerk: The next customer will leave at t = 2.
-Confluent event----------------------------------------------
Clerk: Computed the output function at t = 2
Clerk: A customer just departed!
Clerk: Computed the internal transition function at t = 2
Clerk: There are 0 customers waiting.
Clerk: The next customer will leave at t = 1.79769e+308.
Clerk: Computed the external transition function at t = 2
Clerk: There are 1 customers waiting.
Clerk: The next customer will leave at t = 6.
-External event----------------------------------------------
Clerk: Computed the external transition function at t = 3
Clerk: There are 2 customers waiting.
Clerk: The next customer will leave at t = 6.
-External event----------------------------------------------
Clerk: Computed the external transition function at t = 5
Clerk: There are 3 customers waiting.
Clerk: The next customer will leave at t = 6.
-Internal event----------------------------------------------
Clerk: Computed the output function at t = 6
Clerk: A customer just departed!
Clerk: Computed the internal transition function at t = 6
Clerk: There are 2 customers waiting.
Clerk: The next customer will leave at t = 10.
-External event----------------------------------------------
Clerk: Computed the external transition function at t = 7
Clerk: There are 3 customers waiting.
Clerk: The next customer will leave at t = 10.
-External event----------------------------------------------
Clerk: Computed the external transition function at t = 8
Clerk: There are 4 customers waiting.
Clerk: The next customer will leave at t = 10.
-Confluent event----------------------------------------------
Clerk: Computed the output function at t = 10
Clerk: A customer just departed!
Clerk: Computed the internal transition function at t = 10
Clerk: There are 3 customers waiting.
Clerk: The next customer will leave at t = 12.
```

```
Clerk: Computed the external transition function at t = 10
Clerk: There are 4 customers waiting.
Clerk: The next customer will leave at t = 12.
-External event----------------------------------------------
Clerk: Computed the external transition function at t = 11
Clerk: There are 5 customers waiting.
Clerk: The next customer will leave at t = 12.
-Internal event----------------------------------------------
Clerk: Computed the output function at t = 12
Clerk: A customer just departed!
Clerk: Computed the internal transition function at t = 12
Clerk: There are 4 customers waiting.
Clerk: The next customer will leave at t = 22.
-Internal event----------------------------------------------
Clerk: Computed the output function at t = 22
Clerk: A customer just departed!
Clerk: Computed the internal transition function at t = 22
Clerk: There are 3 customers waiting.
Clerk: The next customer will leave at t = 42.
-Internal Event----------------------------------------------
Clerk: Computed the output function at t = 42
Clerk: A customer just departed!
Clerk: Computed the internal transition function at t = 42
Clerk: There are 2 customers waiting.
Clerk: The next customer will leave at t = 44.
-Internal event----------------------------------------------
Clerk: Computed the output function at t = 44
Clerk: A customer just departed!
Clerk: Computed the internal transition function at t = 44
Clerk: There are 1 customers waiting.
Clerk: The next customer will leave at t = 45.
-Internal event----------------------------------------------
Clerk: Computed the output function at t = 45
Clerk: A customer just departed!
Clerk: Computed the internal transition function at t = 45
Clerk: There are 0 customers waiting.
Clerk: The next customer will leave at t = 1.79769e+308.
```

Now let's define a more complicated clerk model that will interrupt the checkout of one customer in order to more quickly serve customers with very small orders. This new clerk operates as follows. If a customer is being served and another customer arrives whose order can be processed very quickly, then the clerk stops serving the current customer and begins serving the new customer. The clerk, however, will only do this occasionally. To be precise, let's say that a small order is one that requires no more than a single unit of time to process. Moreover, the clerk will not interrupt the processing of an order more often than every 10 units of time.

The new clerk model has two state variables. The first state variable records the amount of time that must elapsed before the clerk is willing to preempt the processing of one customer to serve a customer with a small order. The second is the list of customers waiting to be served. Here is the header file for the new clerk model, which we will call **Clerk2**.

```
#include "adevs.h"
#include "Customer.h"
#include <list>
```

```
class Clerk2: public adevs::Atomic<IO_Type>
{
    public:
        /// Constructor.
        Clerk2();
        /// Internal transition function.
        void delta_int();
        /// External transition function.
        void delta_ext(double e, const adevs::Bag<IO_Type>& xb);
        /// Confluent transition function.
        void delta_conf(const adevs::Bag<IO_Type>& xb);
        /// Time advance function.
        double ta();
        /// Output function.
        void output_func(adevs::Bag<IO_Type>& yb);
        /// Output value garbage collection.
        void gc_output(adevs::Bag<IO_Type>& g);
        /// Destructor.
        ~Clerk2();
        /// Model input port.
        static const int arrive;
        /// Model output port.
        static const int depart;
    private:
        /// Structure for storing information about customers in the line
        struct customer_info_t
        {
            // The customer
            Customer* customer;
            // Time remaining to process the customer order
            double t_left;
        };
        /// List of waiting customers.
        std::list<customer_info_t> line;
        //// Time before we can preempt another customer
        double preempt;
        /// The clerk's clock
        double t;
        /// Threshold correspond to a 'small' order processing time
        static const double SMALL_ORDER;
        /// Minimum time between preemptions.
        static const double PREEMPT_TIME;
};
```

The **Clerk2** constructor sets the clerk's clock and preemption timer to zero.

```
Clerk2::Clerk2():
Atomic<IO_Type>(),
preempt(0.0),
t(0.0)
{
}
```

The output function of the **Clerk2** model sets the exit time of the departing customer and then ejects the customer via its "depart" port.

```
void Clerk2::output_func(Bag<IO_Type>& yb)
{
    /// Set the exit time of the departing customer
    line.front().customer->tleave = t+ta();
    /// Place the customer at the front of the line onto the depart port.
    IO_Type y(depart,line.front().customer);
    yb.insert(y);
    // Report the departure
    cout << "Clerk: A customer departed at t = " << t+ta() << endl;
}
```

The **Clerk2**'s external transition function is significantly different its predecessor. When a new customer arrives, the first thing that the clerk does is reduce the checkout time of the customer that is currently being processed. This reduction reflects the amount of time that has already been spent on the customer's order, which is the time elapsed since the **Clerk2**'s last state transition. Next, the preemption wait time is reduced and the clock is incremented by the same amount. The **Clerk2** records the time at which each arriving customer enters the line; this time is the value of the clock. If any of the arriving customers has a small checkout time and the preemption wait time has expired, then that customer goes to the front of the line. Notice that this preempts the current customer, who now has the second place in line, and causes the preempt wait time to be reset. Otherwise, the new customer simply goes to the back of the line.

```
void Clerk2::delta_ext(double e, const Bag<IO_Type>& xb)
{
    /// Update the clock
    t += e;
    /// Update the time spent working on the current order
    if (!line.empty())
    {
        line.front().t_left -= e;
    }
    /// Reduce the preempt time
    preempt -= e;
    /// Place new customers into the line
    Bag<IO_Type>::const_iterator iter = xb.begin();
    for (; iter != xb.end(); iter++)
    {
        cout << "Clerk: A new customer arrived at t = " << t << endl;
        /// Create a copy of the incoming customer and set the entry time
        customer_info_t c;
        c.customer = new Customer(*((*iter).value));
        c.t_left = c.customer->twait;
        /// Record the time at which the customer enters the line
        c.customer->tenter = t;
        /// If the customer has a small order
        if (preempt <= 0.0 && c.t_left <= SMALL_ORDER)
        {
            cout << "Clerk: The new customer has preempted the current one!" << endl;
            /// We won't preempt another customer for at least this long
            preempt = PREEMPT_TIME;
            /// Put the new customer at the front of the line
            line.push_front(c);
        }
        /// otherwise just put the customer at the end of the line
        else
        {
            cout << "Clerk: The new customer is at the back of the line" << endl;
            line.push_back(c);
        }
    }
}
```

The internal transition function is similar, in many respects, to the external transition function. It begins by decrementing the preempt wait time and incrementing the clock by the amount of time that has elapsed since the last state transition. The customer that just departed the store via the output function is then removed from the front of the queue. If the line is empty then there is nothing else to do and the clerk sits idly behind her counter. If the preemption wait time has expired then the clerk scans the line for the first customer with a small order. If such a customer can be found, that customer moves to the front of the line. Finally, the clerk starts ringing up the first customer in her line. Here is the internal transition function for the **Clerk2** model.

```
void Clerk2::delta_int()
{
    // Update the clerk's clock
    t += ta();
    // Update the preemption timer
    preempt -= ta();
    // Remove the departing customer from the front of the line.
    // The departing customer will be deleted later by our garbage
    // collection method.
    line.pop_front();
    // Check to see if any customers are waiting.
    if (line.empty())
    {
        cout << "Clerk: The line is empty at t = " << t << endl;
        return;
    }
    // If the preemption time has passed, then look for a small
    // order that can be promoted to the front of the line.
    list<customer_info_t>::iterator i;
    for (i = line.begin(); i != line.end() && preempt <= 0.0; i++)
    {
        if ((*i).t_left <= SMALL_ORDER)
        {
            cout << "Clerk: A queued customer has a small order at time " << t << endl;
            customer_info_t small_order = *i;
            line.erase(i);
            line.push_front(small_order);
            preempt = PREEMPT_TIME;
            break;
        }
    }
}
```

The time advance function returns the time remaining to process the customer that is at the front of the line, or infinity (DBL_MAX) if there are no customers to process.

```
double Clerk2::ta()
{
    // If the line is empty, then there is nothing to do
    if (line.empty()) return DBL_MAX;
    // Otherwise, wait until the first customer will leave
    else return line.front().t_left;
}
```

The last function to implement is the confluent transition function. The **Clerk2** model has the same confluent transition as the **Clerk** that is described in section 2:

```
void Clerk2::delta_conf(const Bag<IO_Type>& xb)
{
    delta_int();
    delta_ext(0.0,xb);
}
```

The behavior of the **Clerk2** model is significantly more complex than that of the **Clerk** model. To exercise the **Clerk2**, we replace the **Clerk** model in the example from section 2 the **Clerk2** model and perform the same experiment. Here is the execution output trace for the **Clerk2** model in response to the input sequence shown in Table 3.1. This trace was generated by the print statements shown in the source code listings for the **Clerk2** model.

```
Clerk: A new customer arrived at t = 1
Clerk: The new customer has preempted the current one!
Clerk: A customer departed at t = 2
Clerk: The line is empty at t = 2
Clerk: A new customer arrived at t = 2
Clerk: The new customer is at the back of the line
Clerk: A new customer arrived at t = 3
Clerk: The new customer is at the back of the line
Clerk: A new customer arrived at t = 5
Clerk: The new customer is at the back of the line
Clerk: A customer departed at t = 6
Clerk: A new customer arrived at t = 7
Clerk: The new customer is at the back of the line
Clerk: A new customer arrived at t = 8
Clerk: The new customer is at the back of the line
Clerk: A customer departed at t = 10
Clerk: A new customer arrived at t = 10
Clerk: The new customer is at the back of the line
Clerk: A new customer arrived at t = 11
Clerk: The new customer has preempted the current one!
Clerk: A customer departed at t = 12
Clerk: A customer departed at t = 13
Clerk: A customer departed at t = 23
Clerk: A customer departed at t = 43
Clerk: A customer departed at t = 45
Clerk: The line is empty at t = 45
```

The evolution of the **Clerk2** line is depicted in Fig. 3.1. Until time 11, the line evolves just as it did with the **Clerk** model. At time 11, the **Clerk2** changes the course of the simulation by moving a customer with a small order to the front of the line.

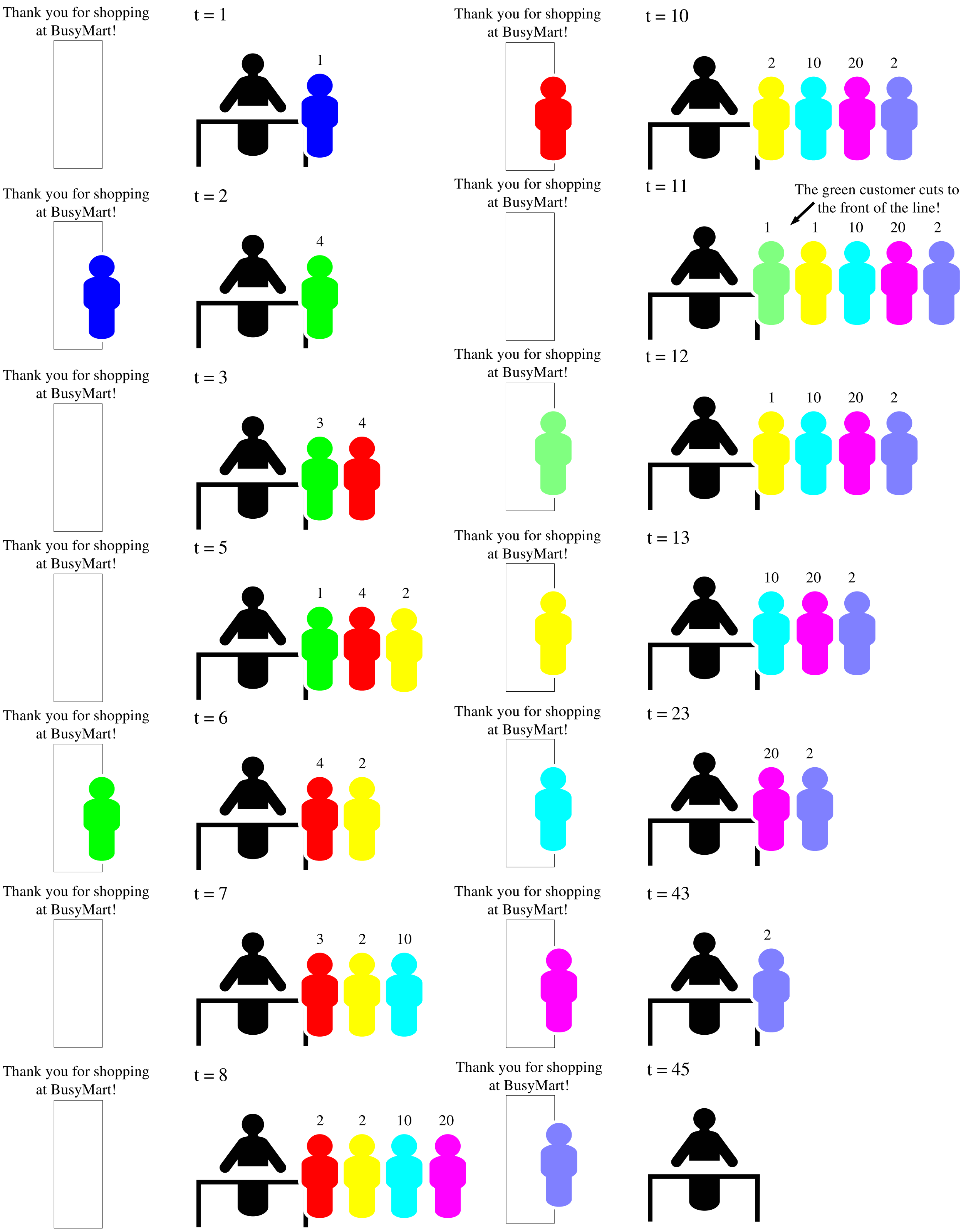
Thank you for shopping at BusyMart!
t = 1
1
Thank you for shopping at BusyMart!
t = 2
4
Thank you for shopping at BusyMart!
t = 3
3 4
Thank you for shopping at BusyMart!
t = 5
1 4 2
Thank you for shopping at BusyMart!
t = 6
4 2
Thank you for shopping at BusyMart!
t = 7
3 2 10
Thank you for shopping at BusyMart!
t = 8
2 2 10 20
Thank you for shopping at BusyMart!
t = 10
2 10 20 2
Thank you for shopping at BusyMart!
t = 11
The green customer cuts to the front of the line!
1 1 10 20 2
Thank you for shopping at BusyMart!
t = 12
1 10 20 2
Thank you for shopping at BusyMart!
t = 13
10 20 2
Thank you for shopping at BusyMart!
t = 23
20 2
Thank you for shopping at BusyMart!
t = 43
2
Thank you for shopping at BusyMart!
t = 45

Figure 3.1: The evolution of the **Clerk2** line in response to the customer arrival sequence listed in Table 3.1.

Chapter 4

Network Models

A network model is a set of atomic models and other network models that are interconnected. Network models are used to model large systems that have many interacting parts. Because network models can be components of other network models, it is possible to build models of very large multi-level systems in an organized fashion.

Unlike atomic models, network models do not directly define new dynamic behavior. The dynamics of a network model are determined by the dynamics of its component parts and their interactions. Atomic models define basic dynamic behavior and network models define structure. Separating a model into dynamic behavior and structure greatly aids the task of simulating large systems with many kinds of interacting parts.

4.1 Parts of a Network Model

Adevs network models are derived from the abstract **Network** class. This class has two abstract methods: ***route*** and ***getComponents***. The ***route*** method implements connections between the components of the **Network** model and between the input/output interface of the **Network** model and its components. The ***getComponents*** method provides a list of components that constitute the **Network** model.

The ***route*** method is the real workhorse of any **Network** model. It describes three things. The first is how components of the **Network** model are connected to each other. The second is how input to the **Network** model is converted into input for the component models. The third is how output from the component models become output from the **Network** model.

The signature of the ***route*** method is

```
void route(const X& value, Devs<X>* model, Bag<Event<X> >& r)
```

where the value argument is the event being routed, the model argument is the **Network** or **Atomic** model that originated the event, and the r argument is a bag to be filled with the event targets. Each target is described by an **Event** object that has the target model and the value to be delivered to it. The simulator uses the ***route*** method to convert output events produced by **Atomic** models to, ultimately, input events for other **Atomic** models. This conversion is done by a somewhat indirect process in which the ***route*** method plays a central role.

An example is the easiest way to understand how the simulator uses the ***route*** method. The simplest example is converting the output from one **Atomic** component of the **Network** into an input for another **Atomic** component in the same **Network**. Figure 4.1 illustrates this case.

The simulator begins by invoking the **output_func** method of **Atomic** model A. Next, the simulator iterates through the elements of A's output bag and calls the **Network**'s ***route*** method for each one. The arguments passed to ***route*** at each call are

1. the output object itself, which becomes the value argument,

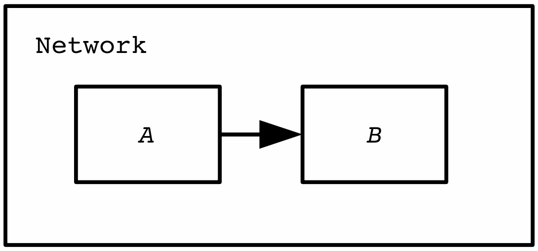

Figure 4.1: Two connected **Atomic** components in a single **Network**.

2. a pointer to A, which is the model argument, and
3. an empty **Bag**.

Two things must be done by the ***route*** method for **Atomic** model B to receive the output object. An **Event** object must be created that contains the output object and a pointer to B and then the **Event** object must be inserted into the **Bag** r. If we suppose, for the sake of illustration, that input and output objects have type int, then the ***route*** method is

```
void route(const int& value, Devs<int>* model, Bag<Event<int> >& r) {
   if (model == A) {
      Event<int> e(B,value);
      r.insert(e);
   }
}
```

where A and B are pointers to the respective components. This ***route*** method implements the network shown in Fig. 4.1.

It is also possible for the **Network** model itself to receive input. This can happen when the network is a component in another **Network** model. Suppose that input to our example **Network** model becomes input to **Atomic** model A. Figure 4.2 extends Fig. 4.1 to include this connection.

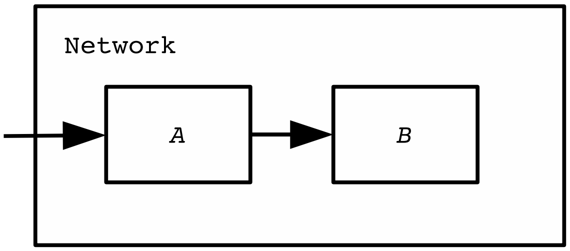

Figure 4.2: Two connected **Atomic** components with external input coupling to component A.

When an event appears at the input of the network, the simulator calls the **Network**'s ***route*** method with the following arguments:

1. the input object itself, which becomes the value argument,
2. a pointer to the **Network** that is receiving the event, and
3. an empty **Bag**.

As before, the ***route*** method must create an **Event** object that indicates the receiving model and the event value. This **Event** is put into the **Bag** r. The code below implements the network shown in Fig. 4.2; the C++ this pointer points to the **Network** itself.

```
void route(const int& value, Devs<int>* model, Bag<Event<int> >& r) {
   if (model == A) {
       Event<int> e(B,value);
       r.insert(e);
   }
   else if (model == this) {
       Event<int> e(A,value);
       r.insert(e);
   }
}
```

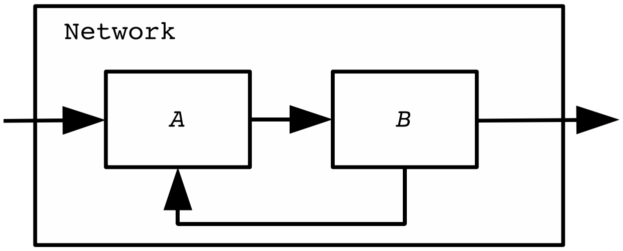

Figure 4.3: A two component network model with external input, external output, and internal coupling.

To complete the example, let's extend the network shown in Fig. 4.2 to include two more connections: a connection from the output of model B to the output of the network and a feedback connection from B to A. This configuration is shown in Fig. 4.3. The only new part of the ***route*** method is that output from model B requires creating an **Event** whose target is the **Network** itself. This event will become output from the **Network** itself. Here is the implementation.

```
void route(const int& value, Devs<int>* model, Bag<Event<int> >& r) {
   if (model == A) {
        Event<int> e(B,value);
        r.insert(e);
    }
    else if (model == this) {
        Event<int> e(A,value);
        r.insert(e);
    }
    else if (model == B) {
        Event<int> e1(this,value);
        Event<int> e2(A,value);
        r.insert(e1);
        r.insert(e2);
    }
}
```

The ***getComponents*** method is the only other method that must be implemented by a **Network** subclass. The simulator passes to this method an empty **Set** of model pointers which must be filled with pointers to the network's components. The ***getComponents*** method signature is

```
void getComponents(Set<Devs<X>*>& c)
```

where c is the set to be filled. There isn't much else to say about this method. The code below shows how it is implemented for the two component network shown in Fig. 4.3; this code, of course, also works for the networks shown in Figs. 4.2 and 4.1.

```
void getComponents(Set<Devs<int>*>& c) {
   c.insert(A);
   c.insert(B);
}
```

There are just three other items to mention in relation to **Network** models. First, components should not be connected to themselves. This means that direct feedback loops and direct throughs in a network model must be avoided. These two cases are illustrated in Fig. 4.4. Second, direct coupling can only occur between components belonging to the same network, and every component must belong to, at most, one network. Third, you'll notice that it is possible for the ***route*** method to modify the value of an output before sending it along. This is permitted and can be useful in some cases.

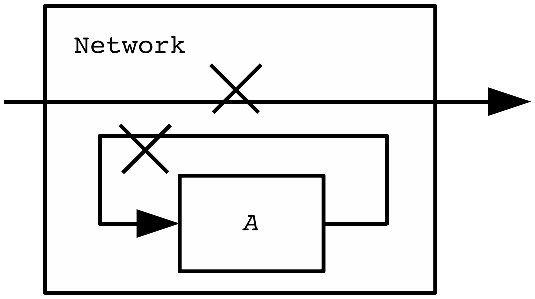

Figure 4.4: Illegal coupling in a **Network** model.

4.2 Simulating a Network Model

Each iteration of a network model simulation has four phases: advance the simulation clock to the next event time, compute model output events and convert the output events to input events, calculate the next state of each model with events to process, and cleanup garbage events. These four phase are repeated until the next event time is at infinity (i.e., DBL_MAX) or you decide to stop the simulation.

Conveniently, there are no special rules for simulating networks of network models. The simulator considers the entire collection of atomic models when determining the next event time, output events from atomic models are recursively routed to atomic destinations, and state transitions and garbage collection are performed over the complete set of active atomic components. Hierarchies of network models are a convenient organizing tool for the modeler, but the simulator ultimately treats a multi-level network as a flat structure.

Algorithm 1 is a sketch of the network model simulation procedure. The atomic model simulation algorithm from section 3 is embedded in the network simulation algorithm. The rules for atomic models do not change in any way; each atomic model sees a sequence of input events and produces a sequence of output events just as before. The only difference here is that the input events are created by other atomic models, and so the input sequence for each atomic model is constructed as the simulation progresses.

4.3 Building a Network Model

Network models are derived from the abstract **Network** class. Every network model must implement the two methods described above: ***getComponents*** and ***route***. Usually, member variables for storing the network structure and methods for initializing the structure are also needed.

I'll use the Adevs **SimpleDigraph** class to illustrate the construction process. The **SimpleDigraph** models a network of components whose connections are represented with a directed graph. If, for example, component A is connected to component B, then all output events generated by A become input events to

Algorithm 1 The simulation procedure for a network model.

```
Initialize the state of every Atomic model
Set the last event time $t_{l,i}$ of every Atomic model $i$ to 0
Set the simulation time $t$ to 0
while The smallest Atomic model next event time < DBL_MAX do
  Set $t$ to the smallest Atomic model next event time
  Find the set of Atomic models whose next event time is equal to $t$. These are the imminent models.
  Get the output of each imminent model by calling its output_func
  Convert imminent model output to input using the Network model route method (do this recursively if the model has more than one level)
  for each Atomic model $i$ that is imminent or has input do
    if $i$ is an imminent model and it does not have input then
      Compute the next model state with delta_int()
    else if $i$ is an imminent and it has input then
      Compute the next model state with delta_conf(xb), where xb is the model input
    else if $i$ is not an imminent model and it has input then
      Compute the next model state with delta_ext($t - t_{l,i}$,xb), where xb is the model input
    end if
    Set $t_{l,i}$ to $t$
  end for
end while
```

B. The **SimpleDigraph** has two methods for building a network. The ***add*** method takes an **Atomic** or **Network** model and adds it to the component set. The ***couple*** method accepts a pair of component models and connects the output of the first to the input of the second. Below is the class definition for the model; note that is has a template parameter for setting the input/output type. The **Network**, **Devs**, **Bag**, and **Set** are in the adevs namespace, and adevs:: must precede them unless the **SimpleDigraph** is in the adevs namespace (which it is).

```
template <class VALUE> class SimpleDigraph: public Network<VALUE> {
   public:
      /// A component of the SimpleDigraph model
      typedef Devs<VALUE> Component;

      /// Construct a network with no components
      SimpleDigraph():Network<VALUE>(){}
      /// Add a model to the network.
      void add(Component* model);
      /// Couple the source model to the destination model
      void couple(Component* src, Component* dst);
      /// Assigns the model component set to c
      void getComponents(Set<Component*>& c);
      /// Use the coupling information to route an event
      void route(const VALUE& x, Component* model, Bag<Event<VALUE> >& r);
      /// The destructor destroys all of the component models
      ~SimpleDigraph();

   private:
      // Component model set
      Set<Component*> models;
      // Coupling information
      std::map<Component*,Bag<Component*> > graph;
};
```

The **SimpleDigraph** has two member variables. Pointers to components of the network are stored in the **Set** models. The components can be **Atomic** objects, **Network** objects, or both. The **SimpleDigraph** components are the nodes of the directed graph. The links, or edges, are stored in the **map** graph.

The ***add*** method does three things. First, it checks that the network is not being added to itself; this is illegal and would cause no end of trouble for the simulator. Next, it adds the component to its component set. Last, the **SimpleNetwork** sets the component's parent. The last step is needed so that the simulator can climb up and down the model tree. If it is omitted then event routing is likely fail. Here is the implementation of the ***add*** method.

```
template <class VALUE>
void SimpleDigraph<VALUE>::add(Component* model) {
   assert(model != this);
   models.insert(model);
   model->setParent(this);
}
```

The ***couple*** method does two things, but one of them is somewhat superfluous. First, it adds the source (src) and destination (dst) models to the component set. We could simply have required that the user call the **add** method before using the ***couple*** method, but adding the components here doesn't hurt and might prevent a few headaches. The second step is essential; the method adds the src → dst link to the graph. Notice that the ***SimpleDigraph*** itself is a node in the network (but it is not in the component set!). Components that are connected to the network create network outputs. A network connection to a component means that the component will receive network inputs. Here is the ***couple*** method implementation.

```
template <class VALUE>
void SimpleDigraph<VALUE>::couple(Component* src, Component* dst) {
   if (src != this) add(src);
   if (dst != this) add(dst);
   graph[src].insert(dst);
}
```

Of the two required methods, ***route*** is the more complicated. The arguments to the method are an input event, the network element (i.e., either the **SimpleDigraph** or one of its components) that is the event source, and the **Bag** that must be filled with **Event** objects that indicate the event receivers. The method begins by finding the collection of components that are connected to the event source. Next we iterate through this collection and for each receiver add an **Event** to the event receiver **Bag**. When this is done the method returns. The implementation is below.

```
template <class VALUE>
void SimpleDigraph<VALUE>::route(const VALUE& x, Component* model,Bag<Event<VALUE> >& r) {
   // Find the list of target models and ports
   typename std::map<Component*,Bag<Component*> >::iterator graph_iter;
   graph_iter = graph.find(model);
   // If no target, just return
   if (graph_iter == graph.end()) return;
   // Otherwise, add the targets to the event bag
   Event<VALUE> event;
   typename Bag<Component*>::iterator node_iter;
   for (node_iter = (*graph_iter).second.begin();
      node_iter != (*graph_iter).second.end(); node_iter++) {
      event.model = *node_iter;
      event.value = x;
      r.insert(event);
   }
}
```

The second required method, ***getComponents***, is trivial. If we had used some collection other than an Adevs **Set** to store the components, then the method would have needed to explicitly insert every component model into the **Set** c. But because models and c are both **Set** objects, and the **Set** has an assignment operator, a simple call to that operator is sufficient.

```
template <class VALUE>
void SimpleDigraph<VALUE>::getComponents(Set<Component*>& c) {
   c = models;
}
```

The constructor and the destructor complete the class. The constructor implementation appears in the class definition; it only calls the superclass constructor. The destructor deletes the component models. Its implementation is shown below.

```
template <class VALUE>
SimpleDigraph<VALUE>::~SimpleDigraph() {
   typename Set<Component*>::iterator i;
   for (i = models.begin(); i != models.end(); i++) {
      delete *i;
   }
}
```

4.4 Digraph Models

This section introduces **Digraph** model as a tool for building block diagram, or directed graph, multi-component models. The model of the convenience store, developed in section 2 , is our first example of a **Digraph** model. The code used to construct the convenience store model (without the **Observer**) is shown below. The block diagram that corresponds to this code snippet is shown in Fig. 4.5.

```
// Create a digraph model whose components use PortValue<Customer*>
// objects as input and output objects.
adevs::Digraph<Customer*> store;
// Create and add the component models
Clerk* clrk = new Clerk();
Generator* genr = new Generator(argv[1]);
store.add(clrk);
store.add(genr);
// Couple the components
store.couple(genr,genr->arrive,clrk,clrk->arrive);
```

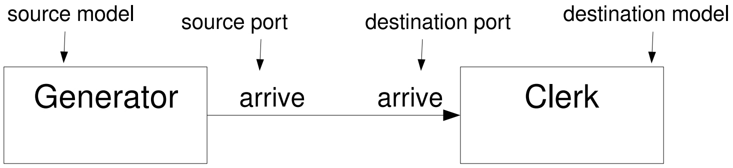

Figure 4.5: A **Digraph** model with two components.

The **Digraph** model is part of the Adevs simulation library. Models that are part of a **Digraph** must use the **adevs::PortValue** objects as their input and output type. The **Digraph** class is a template class with two template parameters. The first is the type of object that will be used as a value in a **PortValue** object. The second parameter is the type of object that will be used as a port in the **PortValue** object. The port parameter is of type 'int' by default.

The **Digraph** class has two primary methods. The ***add()*** method is used to add component models to the block diagram model. The ***couple()*** method is used to connect components of the **Digraph** model. The first two arguments to the ***couple*** method are the source model and source port. The second two arguments are the destination model and the destination port.

The effect of coupling a source model to a destination model is that output produced by the source model on the source port appear as input to the destination model on the destination port. To illustrate this, consider the output function of the **Generator** model shown in Fig. 4.5.

```
void Generator::output_func(Bag<IO_Type>& yb)
{
    // First customer in the list is produced as output
    IO_Type output(arrive,arrivals.front());
    yb.insert(output);
}
```

This places an output value of type ¡Customer*¿ on the “arrive” output port of the **Generator**. A corresponding **PortValue** object appears in the input bag of the **Clerk**. The value of this **PortValue** object points to the **Customer*** object created by the **Generator** and the port is the **Clerk**'s “arrive” port.

In addition to coupling **Atomic** models, the **Digraph** class can also have other **Network** models as its components. Suppose that we want to model a convenience store that has two checkout clerks. When customers are ready to pay their bill, they look for the line with the smallest number of people and enter that line. We can reuse the **Clerk**, **Generator**, and **Observer** models that were introduced in section 2 to build this new model.

The header and source code for the model of the customer's line-selection process is shown below. The model has two output ports, one for each line. There are three input ports. One of these accepts new customers. The others are used to keep track of the number of customers in the each line. The state transition and output functions are self explanatory. Here is the class definition

```
#include "adevs.h"
#include "Customer.h"
#include <list>

// Number of lines to consider.
#define NUM_LINES 2

class Decision: public adevs::Atomic<IO_Type>
{
    public:
        /// Constructor.
        Decision();
        /// Internal transition function.
        void delta_int();
        /// External transition function.
        void delta_ext(double e, const adevs::Bag<IO_Type>& x);
        /// Confluent transition function.
        void delta_conf(const adevs::Bag<IO_Type>& x);
        /// Output function.
        void output_func(adevs::Bag<IO_Type>& y);
        /// Time advance function.
        double ta();
        /// Output value garbage collection.
        void gc_output(adevs::Bag<IO_Type>& g);
        /// Destructor.
        ~Decision();
        /// Input port that receives new customers
        static const int decide;
        /// Input ports that receive customers leaving the two lines
        static const int departures[NUM_LINES];
        /// Output ports that produce customers for the two lines
        static const int arrive[NUM_LINES];

    private:
        /// Lengths of the two lines
        int line_length[NUM_LINES];
        /// List of deciding customers and their decision.
        std::list<std::pair<int,Customer*> > deciding;
        /// Delete all waiting customers and clear the list.
        void clear_deciders();
        /// Returns the arrive port associated with the shortest line
        int find_shortest_line();
};
```

and here is the implementation

```
#include "Decision.h"
#include <iostream>
using namespace std;
using namespace adevs;

// Assign identifiers to ports.  Assumes NUM_LINES = 2.
// The numbers are selected to allow indexing into the
// line length and port number arrays.
const int Decision::departures[NUM_LINES] = { 0, 1 };
const int Decision::arrive[NUM_LINES] = { 0, 1 };
// Inport port for arriving customer that need to make a decision
const int Decision::decide = NUM_LINES;

Decision::Decision():
Atomic<IO_Type>()
{
    // Set the initial line lengths to zero
    for (int i = 0; i < NUM_LINES; i++)
    {
        line_length[i] = 0;
    }
}

void Decision::delta_int()
{
    // Move out all of the deciders
    deciding.clear();
}

void Decision::delta_ext(double e, const Bag<IO_Type>& x)
{
    // Assign new arrivals to a line and update the line length
    Bag<IO_Type>::const_iterator iter = x.begin();
    for (; iter != x.end(); iter++)
    {
        if ((*iter).port == decide)
        {
            int line_choice = find_shortest_line();
            Customer* customer = new Customer(*((*iter).value));
            pair<int,Customer*> p(line_choice,customer);
            deciding.push_back(p);
            line_length[p.first]++;
        }
    }
    // Decrement the length of lines that had customers leave
    for (int i = 0; i < NUM_LINES; i++)
    {
        iter = x.begin();
        for (; iter != x.end(); iter++)
        {
            if ((*iter).port < NUM_LINES)
            {
                line_length[(*iter).port]--;
            }
        }
    }
}

void Decision::delta_conf(const Bag<IO_Type>& x)
{
    delta_int();
    delta_ext(0.0,x);
}

double Decision::ta()
{
    // If there are customers getting into line, then produce output
    // immediately.
    if (!deciding.empty())
    {
        return 0.0;
    }
    // Otherwise, wait for another customer
    else
    {
        return DBL_MAX;
    }
}
```

```
void Decision::output_func(Bag<IO_Type>& y)
{
    // Send all customers to their lines
    list<pair<int,Customer*> >::iterator i = deciding.begin();
    for (; i != deciding.end(); i++)
    {
        IO_Type event((*i).first,(*i).second);
        y.insert(event);
    }
}

void Decision::gc_output(Bag<IO_Type>& g)
{
    Bag<IO_Type>::iterator iter = g.begin();
    for (; iter != g.end(); iter++)
    {
        delete (*iter).value;
    }
}

Decision::~Decision()
{
    clear_deciders();
}

void Decision::clear_deciders()
{
    list<pair<int,Customer*> >::iterator i = deciding.begin();
    for (; i != deciding.end(); i++)
    {
        delete (*i).second;
    }
    deciding.clear();
}

int Decision::find_shortest_line()
{
    int shortest = 0;
    for (int i = 0; i < NUM_LINES; i++)
    {
        if (line_length[shortest] > line_length[i])
        {
            shortest = i;
        }
    }
    return shortest;
}
```

The block diagram model of the store with multiple clerks is shown in Fig. 4.6. The external interface for this block diagram model is identical to the previous clerk models (i.e., the **Clerk** and **Clerk2** models), and we can use the generator and observer models to conduct the same experiments as before. The external "arrive" input of the multi-clerk model is connected to the "decide" input of the **Decision** model. The "depart" output ports of each of the **Clerk** models is connected to the external "arrive" output port of the multi-clerk model. The **Decision** model has two output ports, each one producing customers for a distinct clerk. These output ports are coupled to the "arrive" port of the appropriate clerk model. The **Clerk**'s "depart" output ports are then coupled to the appropriate "departure" port of the decision model.

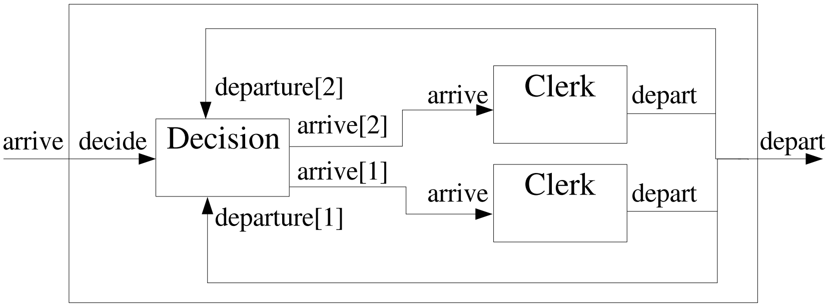

Figure 4.6: Component models and their interconnections in the multi-clerk convenience store model.

The multi-clerk model is implemented by deriving a new class from the **Digraph** class. The constructor of the new class creates and adds the component models and establishes their interconnections. Here is the header file for the new multi-clerk model.

```
#include "adevs.h"
#include "Clerk.h"
#include "Decision.h"

/**
A model of a store with multiple clerks and a "shortest line"
decision process for customers.
*/
class MultiClerk: public adevs::Digraph<Customer*>
{
    public:
        // Model input port
        static const int arrive;
        // Model output port
        static const int depart;
        // Constructor.
        MultiClerk();
        // Destructor.
        ~MultiClerk();
};
```

And here is the source file

```
#include "MultiClerk.h"
using namespace std;
using namespace adevs;

// Assign identifiers to I/O ports
const int MultiClerk::arrive = 0;
const int MultiClerk::depart = 1;
```

```
MultiClerk::MultiClerk():
Digraph<Customer*>()
{
    // Create and add component models
    Decision* d = new Decision();
    add(d);
    Clerk* c[NUM_LINES];
    for (int i = 0; i < NUM_LINES; i++)
    {
        c[i] = new Clerk();
        add(c[i]);
    }
    // Create model connections
    couple(this,this->arrive,d,d->decide);
    for (int i = 0; i < NUM_LINES; i++)
    {
        couple(d,d->arrive[i],c[i],c[i]->arrive);
        couple(c[i],c[i]->depart,d,d->departures[i]);
        couple(c[i],c[i]->depart,this,this->depart);
    }
}

MultiClerk::~MultiClerk()
{
}
```

Notice that the **MultiClerk** destructor does not delete the component models. This is because the component models are adopted by the base class when they are added to the **Digraph**. Consequently, the component models are deleted by the base class destructor, rather than the destructor of the derived class.

4.5 Cell Space Models

A cell space model is a collection of atomic and network models arrange in a regular grid and each model communicates with some arrangement of its neighboring models. Conway's Game of Life is a classic example of a cell space model, and that model can be described very nicely as a discrete event system. The game is played on a flat board that is divided into regular cells. Each cell has a neighborhood that consists of the eight adjacent cells: above, below, left, right, and the four corners. A cell can be dead or alive, and the switch from dead to alive and vice versa occurs according to two rules:

1. If a cell is alive and it has less than two or more than three living neighbors then the cell dies.
2. If a cell is dead and it has three three living neighbors then the cell is reborn.

Our implementation of the Game of Life has two parts: the atomic models that implement the individual cells and the **CellSpace** model that contains the cells and routes their output events. The **CellSpace** is a type of **Network**. The components of a **CellSpace** exchange **CellEvent** objects that have four fields: the x, y, and z coordinates of the target cell and a value to deliver. The **CellEvent** class is a template class whose template argument sets the value type. The size of the **CellSpace** is determined when the **CellSpace** object is created, and it has methods for adding and retrieving cells by their location.

The **Atomic** components in our Game of Life implementation have two state variables: the dead or alive status of the cell and the number of living neighbors. Two methods are implemented to test the death and rebirth rules, and the cell sets its time advance to 1 whenever a rule is satisfied. The cell output is its new dead or alive state. External events update the cell's living neighbor count. In order to produce properly targeted **CellEvent**s, each cell also keeps track of its own location in the cell space. In the example code, the cell space is rendered graphically using OpenGL, but I'll omit that part. Here is the header file for our Game of Life cell.

```
/// Possible cell phases
typedef enum { Dead, Alive } Phase;
/// IO type for a cell
typedef adevs::CellEvent<Phase> CellEvent;

/// A cell in the Game of Life.
class Cell: public adevs::Atomic<CellEvent> {
   public:
      /**
      Create a cell and set the initial state.
      The width and height fields are used to determine if a
      cell is an edge cell.  The last phase pointer is used to
      visualize the cell space.
      */
      Cell(long int x, long int y, long int width, long int height,
      Phase phase, short int nalive, Phase* vis_phase = NULL);

      ... Required Adevs methods and destructor ...

   private:
      // location of the cell in the 2D space
      long int x, y;
      // dimensions of the 2D space
      static long int w, h;
      // Current cell phase
      Phase phase;
      // number of living neighbors.
      short int nalive;
      // Output variable for visualization
      Phase* vis_phase;

      // Returns true if the cell will be born
      bool check_born_rule() const {
         return (phase == Dead && nalive == 3);
      }
      // Return true if the cell will die
      bool check_death_rule() const {
         return (phase == Alive && (nalive < 2 || nalive > 3));
      }
};
```

The template argument supplied to the base **Atomic** class is a **CellEvent** whose value field has the type **Phase**. The ***check_born_rule*** method tests the rebirth condition and ***check_death_rule*** method tests the death condition. The appropriate rule, as determined by the cell's dead or alive status, is used in the time advance, output, and internal transition methods. The number of living cells is updated by the cell's ***delta_ext*** method whenever neighboring cells report a change in their health. Here are the ***Cell***'s method implementations.

```
Cell::Cell(long int x, long int y, long int w, long int h,
Phase phase, short int nalive, Phase* vis_phase):
adevs::Atomic<CellEvent>(),x(x),y(y),phase(phase),nalive(nalive),vis_phase(vis_phase) {
   // Set the global cellspace dimensions
   Cell::w = w; Cell::h = h;
   // Set the initial visualization value
   if (vis_phase != NULL) *vis_phase = phase;
}

double Cell::ta() {
   // If a phase change should occur then change state
   if (check_death_rule() || check_born_rule()) return 1.0;
   // Otherwise, do nothing
   return DBL_MAX;
}

void Cell::delta_int() {
   // Change the cell state if necessary
   if (check_death_rule()) phase = Dead;
   else if (check_born_rule()) phase = Alive;
}

void Cell::delta_ext(double e, const adevs::Bag<CellEvent>& xb) {
   // Update the living neighbor count
   adevs::Bag<CellEvent>::const_iterator iter;
   for (iter = xb.begin(); iter != xb.end(); iter++) {
      if ((*iter).value == Dead) nalive--;
      else nalive++;
   }
}

void Cell::delta_conf(const adevs::Bag<CellEvent>& xb) {
   delta_int();
   delta_ext(0.0,xb);
}

void Cell::output_func(adevs::Bag<CellEvent>& yb) {
   CellEvent e;
   // Assume we are dying
   e.value = Dead;
   // Check in case this in not true
   if (check_born_rule()) e.value = Alive;
   // Set the visualization value
   if (vis_phase != NULL) *vis_phase = e.value;
   // Generate an event for each neighbor
   for (long int dx = -1; dx <= 1; dx++) {
      for (long int dy = -1; dy <= 1; dy++) {
         e.x = (x+dx)%w;
         e.y = (y+dy)%h;
         if (e.x < 0) e.x = w-1;
         if (e.y < 0) e.y = h-1;
         // Don’t send to self
         if (e.x != x || e.y != y)
```

```
            yb.insert(e);
        }
    }
}
```

The ***output_func*** method shows how a cell sends messages to its neighbors. The double for loop creates a **CellEvent** targeted at each adjacent cell. The location of the target cell is written to the x, y, and z fields of the **CellEvent** object. Just like arrays, the location values can range from zero to the cell space size minus one. The **CellSpace** will do the actual routing of the **CellEvent**s to their targets. Note however that if the target of the **CellEvent** is outside of the cell space, then the **CellSpace** itself will produce the **CellEvent** as an output.

The remainder of the simulation program looks very much like the other simulation programs that we've seen so far (except for some OpenGL specific code, omitted here, that is used to display the cell space). A **CellSpace** object is created and we add each cell to it. Then a **Simulator** object is create and a pointer to the **CellSpace** is passed to the **Simulator**'s constructor. Last, we execute events until our stopping criteria is met. The execution part is already familiar, so let's just focus on creating the **CellSpace**. Here is the code snippet that performs the construction.

```
// Create the cellspace model
cell_space = new adevs::CellSpace<Phase>(WIDTH,HEIGHT);
for (int x = 0; x < WIDTH; x++) {
   for (int y = 0; y < HEIGHT; y++) {
      // Count the living neighbors
      short int nalive = count_living_cells(x,y);
      // The 2D phase array contains the initial Dead/Alive state of each cell
      cell_space->add(
         new Cell(x,y,WIDTH,HEIGHT,phase[x][y],nalive,&(phase[x][y])),x,y);
   }
}
```

Just as with the **Digraph** class, the **CellSpace** template argument determines the value type for the **CellEvent**s that are used as input and output by the **CellSpace** components. The **CellSpace** constructor sets the dimensions of the space. Every **CellSpace** is three dimensional, and the constructor accepts three arguments that set the x, y, and z dimensions; omitted arguments default to 1. The constructor signature is

```
CellSpace(long int width, long int height = 1, long int depth = 1)
```

Components are added to the cellspace with the ***add*** method. This method places a component at a specific (x,y,z) location. Its signature is

```
void add(Cell* model, long int x, long int y = 0, long int z = 0)
```

where **Cell** is a **Devs** (atomic or network) by the type definition

```
typedef Devs<CellEvent<X> > Cell;
```

The **CellSpace** deletes its components when it is deleted. The **CellSpace** class has five other methods for retrieving cells and getting the dimensionality of the cell space. These are more or less self-explanatory; the signatures are shown below.

```
const Cell* getModel(long int x, long int y = 0, long int z = 0) const;
Cell* getModel(long int x, long int y = 0, long int z = 0);
long int getWidth() const;
long int getHeight() const;
long int getDepth() const;
```

The Game of Life produces a surprising number of clearly recognizable patterns. Some of these patterns are fixed and unchanging; others oscillate, cycling through a set of patterns that always repeats itself; others seem to crawl or fly. One familiar static pattern is the Block shown in Fig. 4.7. Our discrete event implementation of the Game of Life doesn't do any work when simulating a Block. None of the cells in a Block change in any way; their phases are constant and so are their neighbor counts. The Blinker shown in Fig. 4.8 is more interesting. This oscillating pattern has just two stages: a vertical and a horizontal. Table 4.1 shows the input, output, and state transitions that are computed for the cell marked with * in Fig. 4.8. Just like the pattern it is a part of, the cell oscillates between two different states.

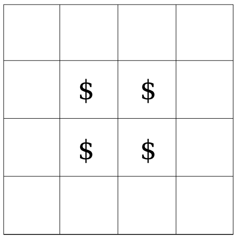

Figure 4.7: The Block.

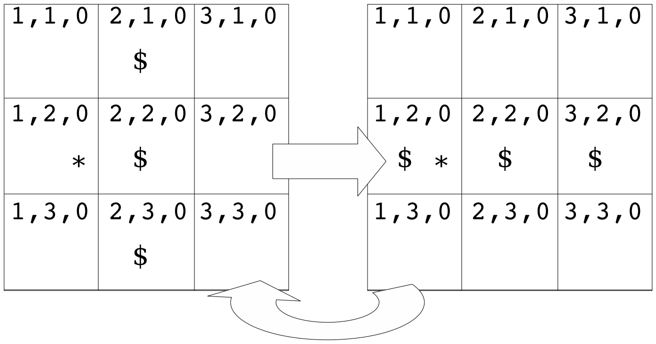

Figure 4.8: The Blinker. The input, output, and state transitions for the cell marked with a * are shown in Table 4.1. The address of each cell is shown in its upper left corner. Living cells are indicated with a $.

Time	State	Input	Output to all neighbors
0	(dead,3)	No input	No Output
1	(alive,1)	(dead,2,1,0) (dead,2,3,0)	alive
2	(dead,1)	(alive,2,1,0) (alive,2,3,0)	dead

Table 4.1: State, input, and output trajectory for the cell marked with * in Fig. 4.8.

The confluent transition function plays a major role in the Blinker simulation. Most of the rows in Table 4.1 (all but the first row, in fact) have both an input and an output, which means that an internal and external event coincide and so the next state is determined by the ***delta_conf*** method. It is also important that the input and output bags carry multiple values. The external transition function (which is used in defining the confluent transition function) must be able to compute the number of living neighbors before determining its next state. If input events were provided one at a time (e.g., if the input bag were replaced by a single input event), then our discrete event Game of Life would be much more difficult to implement.

Chapter 5

Variable Structure Models

The composition of a variable structure model changes through time. New components are added as machinery is installed in a factory, organisms reproduce, or shells are fired from a cannon. Existing components are removed as machines break, organisms dies, or shells in flight find their targets. Components are rearranged as parts move through a manufacturing process, organisms migrate, or a command and control network loses communication lines. But structure change can not occur willy nilly if we want our simulation to produce well defined, repeatable outcomes. For this reason, Adevs provides a simple but effective mechanism for coordinating structure changes with model state transitions[1].

5.1 Building and Simulating Variable Structure Models

Every Adevs model, **Network** and **Atomic**, has an abstract method called ***model_transition***. This method is inherited from the **Devs** class that is at the top of the Adevs class hierarchy. The signature of the ***model_transition*** method is

```
bool model_transition()
```

and its default implementation simply returns false.

At the end of every simulation cycle the simulator invokes the ***model_transition*** method of every **Atomic** model that changed in that cycle. When the ***model_transition*** method is invoked the **Atomic** model can do almost anything it likes except alter the component set of a **Network** model. If the ***model_transition*** method returns true, then the simulator will also call the model's parent. The parent is, of course, a **Network** model; its ***model_transition*** method may add, remove, and rearrange components. But it must not delete components! The simulator will automatically delete removed components when the structure change calculations are finished. As before, if the **Network**'s ***model_transition*** method returns true then the simulator will invoke the ***model_transition*** method of the **Network**'s parent.

After invoking every eligible model's ***model_transition*** method, the simulator performs a somewhat complicated cleanup process. This process requires that simulator construct two sets. The first set contains all of the components that belonged to all of the **Network** models whose ***model_transition*** method was invoked and all of the components belonging to components that are in this set. The second set is defined in the same way, but it is computed using component sets as they exist after the ***model_transition*** methods have been invoked. The simulator deletes every model that has actually been removed; these are the models in the first set but not in the second. The simulator initializes every model that is genuinely new by computing its next event time (i.e., its creation time plus its time advance) and putting it into the event schedule; these are the models in the second second set but not in the first. The simulator leaves all other models alone. This confusing procedure is illustrated in Fig. 5.1.

---

[1] The dynamic structure features in Adevs are based on the Dynamic DEVS modeling formalism described in A.M. Uhrmacher's paper "Dynamic structures in modeling and simulation: a reflective approach", ACM Transactions on Modeling and Computer Simulation (TOMACS), Volume 11, Issue 2, pgs. 202-232, April 2001.

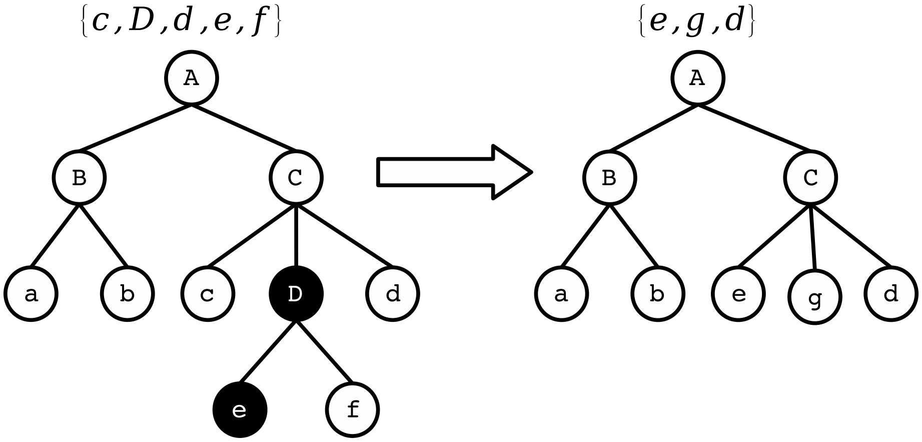

Figure 5.1: The black models' ***model_transition*** methods returned true. The set of components considered before and after the structure change are shown in the before (left) and after (right) trees. The set of deleted components is $\{c, D, d, e, f\} - \{e, g, d\} = \{c, D, f\}$. The set of new components is $\{e, g, d\} - \{c, D, d, e, f\} = \{g\}$.

The ***model_transition*** method can break the strict hierarchy and modularity that is usual observed when building **Atomic** and **Network** models. Any **Network** model can modify the component set of any other model regardless of proximity or hierarchy. The potential for anarchy is great; the design of a variable structure model should be carefully considered. There are two approaches that are simple and, in many cases, entirely adequate.

The first approach is to allow only **Network** models to effect structure changes and to restrict those changes to the **Network**'s immediate sub-components. With this approach, an **Atomic** model initiates a structure change by posting a structure change request for its parent **Network**. The **Atomic** model's ***model_transition*** method then returns true causing its parent's ***model_transition*** method to be invoked. The parent **Network** model then retrieves and acts on the posted structure change request. The **Network** repeats this process if it wants to effect structure changes involving models other than its immediate children.

The second approach allows arbitrary structure changes by forcing the model at the very top of the hierarchy to invoke its ***model_transition*** method. This causes the simulator to consider every model in the aftermath of a structure change. As in the first approach, an **Atomic** model that wants to effect a structure change uses its ***model_transition*** method to post a change request for its parent. This is percolated up the model hierarchy by the **Network** models whose ***model_transition*** methods always return true.

The first approach trades flexibility for execution time; the second approach trades execution time for flexibility. With the first approach, structure changes that involve a small number of components require a small amount of work by the simulator. With the second approach, every structure change requires the simulator to include every part of the model in its set calculations regardless of the structure change's actual extent, but the scope of a structure change is unlimited.

5.2 A Variable Structure Example

The Custom Widget Company is expanding its operations. Plans are being drawn for a new factory that will make custom gizmos (and the company name will be changed to The Custom Widget and Gizmo Company). The factory machines are expensive to operate. To keep costs down, the factory will operate just enough machinery to fill outstanding gizmo orders in sufficient time. The factory must have enough machinery to meet peak demand, but much of the machinery will be idle much of the time. The factory engineers want to answer two questions: how many machines are needed and how much will it costs to operate the them.

We are going to use a variable structure model to answer these two questions. The model will have three components: a generator that creates factory orders, a model of a single machine, and a model of the factory which contains the machine models and activates and deactivates machines as required to satisfy demand. The complete factory model is illustrated in Fig. 5.2.

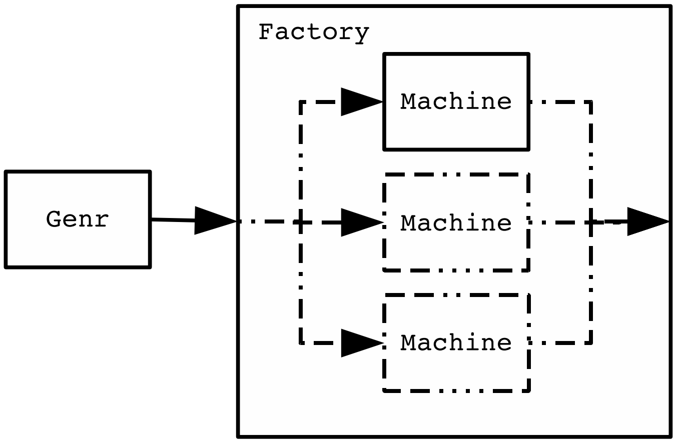

Figure 5.2: Block diagram of the variable structure factory model. The broken lines indicate structural elements that are subject to dynamic changes.

The generator creates new orders for the factory. Each order is identified with its own integer label, and the generator produces orders at the rate anticipated by the factory planners. The order arrival rate and, consequently, the time advance of the generator are not constants. Demand at the factory is expected to be fairly steady with a new order arriving every 1/2 to 2 days; demand is modeled with a random variable uniformly distributed in the range [0.5,2]. Here is the generator code:

```
#include "adevs.h"
// The Genr models factory demand. It creates new orders every 0.5 to 2 days.
class Genr: public adevs::Atomic<int>
{
    public:
        /**
         * The generator requires a seed for the random number that determines
         * the time between new orders.
         */
        Genr(unsigned long seed):adevs::Atomic<int>(),next(1),u(seed){ set_time_to_order(); }
        // Internal transition updates the order counter and determines the next arrival time
        void delta_int() { next++; set_time_to_order(); }
        // Output function produces the next order
        void output_func(adevs::Bag<int>& yb) { yb.insert(next); }
        // Time advance returns the time until the next order
        double ta() { return time_to_order; }
        // Model is input free, so these methods are empty
        void delta_ext(double,const adevs::Bag<int>&){}
        void delta_conf(const adevs::Bag<int>&){}
        // No explicit memory management is needed
        void gc_output(adevs::Bag<int>&){}
    private:
        // Next order ID
        int next;
        // Time until that order arrives
        double time_to_order;
        // Random variable for producing order arrival times
        adevs::rv u;
        // Method to set the order time
        void set_time_to_order() { time_to_order = u.uniform(0.5,2.0); }
};
```

The machine model is similar to the **Clerk** model that appeared in section 2. Each machine requires 3 days make a gizmo and orders are processed first come first serve. The **Machine**'s ***model_transition*** method is inherited from the **Atomic** class, which inherited it from the **Devs** class (the inheritance hierarchy is **Devs** ← **Atomic Machine**). I'll discuss the role of the ***model_transition*** method after introducing the **Factory** class; here is the **Machine** model code.

```
#include "adevs.h"
#include <cassert>
#include <deque>
/**
 * This class models a machine as a fifo queue and server with fixed service time.
 * The model_transition method is used, in conjunction with the Factory model_transition
 * method, to add and remove machines as needed to satisfy a 6 day turnaround time
 * for orders.
 */
class Machine: public adevs::Atomic<int>
{
    public:
        Machine():adevs::Atomic<int>(),tleft(DBL_MAX){}
        void delta_int()
        {
            q.pop_front(); // Remove the completed job
            if (q.empty()) tleft = DBL_MAX; // Is the Machine idle?
            else tleft = 3.0; // Or is it still working?
        }
        void delta_ext(double e, const adevs::Bag<int>& xb)
        {
            // Update the remaining time if the machine is working
            if (!q.empty()) tleft -= e;
            // Put new orders into the queue
            adevs::Bag<int>::const_iterator iter = xb.begin();
            for (; iter != xb.end(); iter++)
            {
                // If the machine is idle then set the service time
                if (q.empty()) tleft = 3.0;
                // Put the order into the back of the queue
                q.push_back(*iter);
            }
        }
        void delta_conf(const adevs::Bag<int>& xb)
        {
            delta_int();
            delta_ext(0.0,xb);
        }
        void output_func(adevs::Bag<int>& yb)
        {
            // Expel the completed order
            yb.insert(q.front());
        }
        double ta()
        {
            return tleft;
        }
        // The model transition function returns true if another order can not
        // be accommodated or if the machine is idle.
        bool model_transition()
        {
            // Check that the queue size is legal
            assert(q.size() <= 2);
            // Return the idle or full status
            return (q.size() == 0 || q.size() == 2);
        }
        // Get the number of orders in the queue
        unsigned int getQueueSize() const { return q.size(); }
        // No garbage collection
        void gc_output(adevs::Bag<int>&){}
    private:
        // Queue for orders that are waiting to be processed
        std::deque<int> q;
        // Time remaining on the order at the front of the queue
        double tleft;
};
```

The number of **Machine** models in the **Factory** any time is determined by the current demand for gizmos. The real factory, of course, will have a set number of physical machines on the factory floor, but the planners do not yet know how many machines are needed. A variable structure model that creates and destroys machines as needed is a good way to accommodate this uncertainty (a design decision similar to using a linked list in place of a fixed size array).

The Custom Widget and Gizmo Company has built its reputation on a guaranteed service time, from order to delivery, of 15 days. This leaves only 6 days for the manufacturing process (the remaining time being consumed by order processing, delivery, etc.). A single machine can meet this schedule if it has at most one order waiting in its queue. But it costs a dollar a day to operate a machine and so the factory engineers want to minimize the number of machines working at any particular time. With this goal, the factory operating policy has two rules:

1. assign incoming orders to the active machine that can provide the shortest turn around time and
2. keep just enough active machines to have capacity for one additional order.

The **Factory** model implements this policy in the following way. When a **Machine** becomes idle or its queue is full (i.e., the machine is working on one order and has another order waiting in its queue) then its ***model_transition*** method returns true. This causes the **Factory**'s ***model_transition*** method to be invoked. The **Factory** first looks for and removes machines that have no work and then examines each remaining machine to determine if the required one unit of additional capacity is available. If the required unit of additional capacity is not available then the **Factory** creates a new machine.

This is an example of the first approach to building a variable structure model. With this design, the set calculations that are done when the **Factory**'s ***model_transition*** method is invoked are limited to instances where **Machine** models are likely to be created or destroyed. Our design, however, is complicated somewhat by the need for **Machine** and **Factory** objects to communicate (i.e., the **Machine**s must watch their own status and inform the **Factory** when there is a potential capacity shortage). If we had used the second approached to build our variable structure model, then the **Machine**s' ***model_transition*** methods could have merely returned true; no need for a status check. The **Factory** would have iterated through its list of **Machine**s, adding and deleting **Machine**s as needed. This is more computationally expensive; the simulator would look for changes in the **Factory**'s component set at the end of every simulation cycle. But the software design is simpler, albeit only marginally so in this instance.

The **Factory** is a **Network** model, and we need to implement all of the **Network**'s virtual methods: ***route***, ***getComponents***, and ***model_transition***. The ***route*** method is responsible for assigning orders to the proper **Machine**. When an order arrives, it is sent to the machine with the shortest total service time. The ***getComponents*** method puts the current machine set into the output **Set** c. The ***model_transition*** method examines the status of each machine, deleting idle machines and adding a new machine if it is needed to maintain reserve capacity. The complete **Factory** implementation is shown below.

```
#include "adevs.h"
#include "Machine.h"
#include <list>

class Factory: public adevs::Network<int> {
   public:
      Factory();
      void getComponents(adevs::Set<adevs::Devs<int>*>& c);
      void route(const int& order, adevs::Devs<int>* src,
            adevs::Bag<adevs::Event<int> >& r);
      bool model_transition();
      ~Factory();
      // Get the number of machines
      int getMachineCount();
   private:
      // This is the machine set
      std::list<Machine*> machines;
      // Method for adding a machine to the factory
      void add_machine();
      // Compute time needed for a machine to finish a new job
      double compute_service_time(Machine* m);
};
```

```
#include "Factory.h"
using namespace adevs;
using namespace std;

Factory::Factory():
Network<int>() { // call the parent constructor
   add_machine(); // Add the first machine the the machine set
}

void Factory::getComponents(Set<Devs<int>*>& c) {
   // Copy the machine set to c
   list<Machine*>::iterator iter;
   for (iter = machines.begin(); iter != machines.end(); iter++)
      c.insert(*iter);
}

void Factory::route(const int& order, Devs<int>* src, Bag<Event<int> >& r) {
   // If this is a machine output, then it leaves the factory
   if (src != this) {
      r.insert(Event<int>(this,order));
      return;
   }
   // Otherwise, use the machine that can most quickly fill the order
   Machine* pick = NULL;  // No machine
   double pick_time = DBL_MAX; // Infinite time for service
   list<Machine*>::iterator iter;
   for (iter = machines.begin(); iter != machines.end(); iter++) {
      // If the machine is available
      if ((*iter)->getQueueSize() <= 1) {
         double candidate_time = compute_service_time(*iter);
         // If the candidate service time is smaller than the pick service time
         if (candidate_time < pick_time) {
            pick_time = candidate_time;
            pick = *iter;
         }
      }
   }
   // Make sure we found a machine with a small enough service time
   assert(pick != NULL && pick_time <= 6.0);
   // Use this machine to process the order
   r.insert(Event<int>(pick,order));
}

bool Factory::model_transition() {
   // Remove idle machines
   list<Machine*>::iterator iter = machines.begin();
   while (iter != machines.end()) {
      if ((*iter)->getQueueSize() == 0) iter = machines.erase(iter);
      else iter++;
   }
   // Add the new machine if we need it
   int spare_cap = 0;
   for (iter = machines.begin(); iter != machines.end(); iter++)
         spare_cap += 2 - (*iter)->getQueueSize();
   if (spare_cap == 0) add_machine();
   return false;
}

void Factory::add_machine() {
   machines.push_back(new Machine());
   machines.back()->setParent(this);
}

double Factory::compute_service_time(Machine* m) {
   // If the machine is already working
   if (m->ta() < DBL_MAX) return 3.0+(m->getQueueSize()-1)*3.0+m->ta();
   // Otherwise it is idle
   else return 3.0;
}
```

```
int Factory::getMachineCount() {
   return machines.size();
}

Factory::~Factory() {
   // Delete all of the machines
   list<Machine*>::iterator iter;
   for (iter = machines.begin(); iter != machines.end(); iter++)
      delete *iter;
}
```

To illustrate how the ***model_transition*** method is used, let's manually simulate the processing of a few orders: the first order arrives at day zero, the second order at day one, and the third order at day three. At the start of day zero there is one idle **Machine**. When the first order arrives the **Factory**'s ***route*** method is invoked and it sends the order to the idle **Machine**. The **Machine**'s ***delta_ext*** method is invoked next and the **Machine** begins processing the order. Then the **Machine**'s ***model_transition*** method is invoked, discovers that the **Machine** is working and has space in its queue, and returns false.

When the second order arrives on day one, the **Factory**'s ***route*** method is called again. There is only one **Machine** and it has space in its queue so the order is routed to that **Machine**. The **Machine**'s ***delta_ext*** method is invoked next, and the second order is queued. The **Machine**'s ***model_transition*** method is now invoked; the queue is full and so the method returns true. This causes the the **Factory**'s ***model_transition*** method to be invoked; it examines the **Machine**'s status, sees that it overloaded, and creates a new **Machine**. At this time, the working **Machine** needs two more days to finish the first order and needs a total of five days to complete its second order.

There is a great deal of activity when the third order arrives on day three. First, the working **Machine**'s ***output_func*** method is called and it spits out the completed order (the order begun on day zero). Then the **Factory**'s ***route*** method is called twice. First it converts the **Machine** output into a **Factory** output, and then it routes the new order to the idle **Machine** (the order of these ***route*** calls could have been switched). Next the state transition methods for the two **Machine**s are invoked. The working **Machine**'s ***delta_int*** method is called and it starts work on its queued order. The idle **Machine**'s ***delta_ext*** method is called and it begins processing the new order. Finally, the ***model_transition*** methods of both **Machine**s are invoked; both **Machine**'s have room in their queue and so both methods return false.

For the sake of illustration, suppose no orders arrive in the next three days (this is impossible when orders arrive every one half to two days, but bear with me). At day six, both machines will finish their orders. The **Machine**s' ***output_func*** methods will be invoked, producing the finished orders which are sent to the **Factory** output via the **Factory**'s ***route*** method. Next, the **Machine**s' ***delta_int*** methods will be called and both **Machine**s will become idle. Then the **Machine**s' ***model_transition*** methods will be invoked and these will return true. This will cause the **Factory**'s ***model_transition*** method to be called. It will examine the status of each **Machine**, see that they are idle, and delete both of them. Then the **Factory** will compute its available capacity, which is now zero, and create a new machine. Incidentally, this returns the **Factory** to its initial state of having one idle **Machine**.

The factory engineers have two questions: how many machines are needed and what is the factory's annual operating cost. These questions can be answered with a plot of the active machine count versus time. The required number of machines is the maximum value of the active machine count. Each machine costs a dollar per day to operate, and so the operating cost is just the one year time integral of the active machine count.

A plot of the active machine count versus time is shown in Fig. 5.3. The maximum active machine count in this plot is 4 and the annual operating cost is $944 (this plot is from the first simulation run listed in Table 5.1). The arrival rate is a random number, and so the annual operating cost and maximum machine count are themselves random numbers. Consequently, data from several simulation runs is needed to make an informed decision. Somewhat arbitrarily, I have listed ten simulation runs; each run uses a different random number generator seed and produces a different outcome (i.e., another sample of the maximum active machine count and annual operating cost). The maximum active machine count and annual operating cost generated by each run is shown in Table 5.1. From this data, the factory engineers conclude that 4 machines are required and the average annual operating cost will be $961.

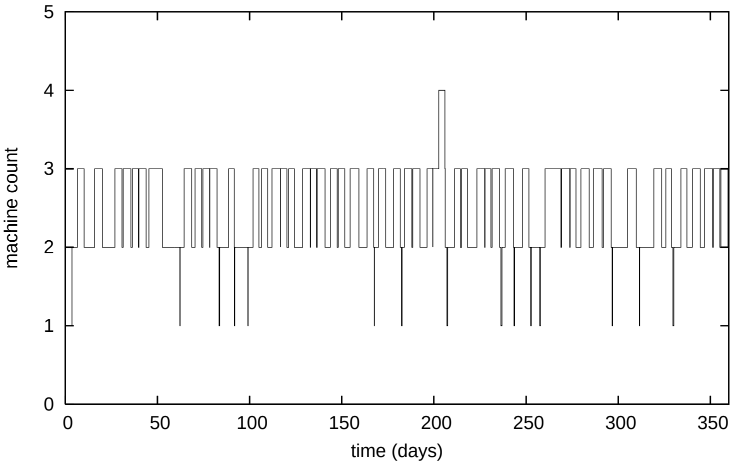

Figure 5.3: Active machine count over one year.

Seed	Maximum machine count	Annual operating cost
1	4	$944.05
234	4	$968.58
15667	4	$980.96
999	3	$933.13
9090133	4	$961.65
6113	4	$977.33

Table 5.1: Outcomes of ten factory simulation runs.

Chapter 6

Continuous Models

A complicated system is likely to have parts that are best modeled with continuous equations. Where continuous models interact with discrete event models, these interactions are necessarily discrete. For example, a digital thermometer reports temperature in discrete increments, circuit breakers and electrical switches are either open or closed, a threshold sensor is either tripped or it is not. If, on the other hand, two systems interact continuously, then the both systems are probably best modeled with continuous mathematics. In this case, accurate calculations are greatly facilitated by lumping the two systems into a single assembly; in Adevs this assembly is an **Atomic** model that encapsulates the system's continuous dynamics.

There are three possibly outcomes if we follow this lumping process to its conclusion. One possibility is that we end up with a single assembly; in this case our model is essentially continuous and we are probably better off using a simulation tool for continuous systems. At the other extreme, we find that the continuous parts of our model are relatively simple; they yield to analytical techniques and can be be easily transformed into discrete event models. Between these two extremes are models with continuous dynamics that are not simple but which do not dominate the modeling problem. The continuous system simulation part of Adevs is aimed at this third type of model.[1]

6.1 Using the Runge-Kutte Integration Modules

Adevs has two pre-built **Atomic** models that can be used to simulation continuous systems. These two models are essentially the same; one uses a fixed step size, fourth order Runge-Kutte integration scheme to solve a set of ordinary differential equations and the other uses a variable step size, fourth/fifth order Runge-Kutte scheme to do the same thing. In general, you will probably prefer to use the variable step size scheme because it, unlike the fixed step size scheme, has a built in error control mechanism. Both models are used in exactly the same way, differing only in the parameters that are passed to their constructors.

The Adevs RK models are abstract classes with seven abstract methods that must be implemented by your derived class. The RK models are derived from the **Atomic** class, but you will not be implementing the five familiar methods ***delta_int***, ***delta_ext***, ***delta_conf***, ***ta***, and ***output_func***. But you will implement the familiar ***gc_output*** method. It performs the same function in this new context, differing only in that it operates on objects produced by the new ***discrete_output*** method. The remaining six new methods are used to describe the continuous dynamics of your model, to describe how your model generates and responds to discrete events, and to record the model's continuous trajectory. The methods are

`void der_func(const double* q, double* dq)`

---

[1] The method used here for adding continuous models to a discrete event simulation is described and illustrated in the following papers: James Nutaro, Teja Kuruganti, and Mallikarjun Shankar. Seamless Simulation of Hybrid Systems with Discrete Event Software Packages. In the Proceedings of the 40th Annual Simulation Symposium, pp. 81-87, March 2007 and James Nutaro, Phani Teja Kuruganti, Laurie Miller, Sara Mullen and Mallikarjun Shankar. Integrated Hybrid-Simulation of Electric Power and Communications Systems. In Proceedings of the 2007 IEEE Power Engineering Society General Meeting, pp. 1-8, June 2007.

```
void state_event_func(const double* q, double* z)
double time_event_func(const double* q)
void discrete_action(double* q, const Bag<X>& xb)
void discrete_output(const double* q, Bag<X>& yb)
void state_changed(const double* q)
```

which are used, as the names suggest, to implement the state variable derivative functions and state event conditions, to schedule time events, to implement discrete state changes, to generate discrete outputs, and to take an action (usually recording the state trajectory in a file) when the integration scheme changes a state variable.

I'll use a simple, if contrived, example to introduce the parts of a continuous model and the corresponding use of the RK model methods. A cherry bomb[2] is dropped from a height of 1 meter and bounces until it either explodes or is doused with water. We'll assume that the cherry bomb only bounces up and down and that it is perfectly elastic. The cherry bomb will explode 2 seconds from the time it is lit and dropped. Dousing the cherry bomb will put out the fuse[3]. Dousing is an input event and the cherry bomb will produce an output event if it explodes.

This model has two continuous state variables: the height and velocity of the cherry bomb. Between events, these variables are governed by the pair of differential equations

$$\dot{v} = -9.8 \tag{6.1}$$

$$\dot{h} = v \tag{6.2}$$

where 9.8 is acceleration due to gravity, v is velocity, and h is height. In this example, it will also be useful to know the current time. We can keep track of this by adding one more differential equation

$$\dot{t} = 1 \tag{6.3}$$

whose solution is $t_0 + t$ or just t if we set $t_0 = 0$. The ball bounces when it hits the floor; the effect of a bounce is to instantaneously reverse the cherry bomb's velocity; specifically

$$h = 0 \;\&\; v < 0 \implies v \leftarrow -v \tag{6.4}$$

where $\implies$ is logical implication and $\leftarrow$ indicates an assignment.

Equations 6.1 and 6.2 (and 6.3) are the state variable derivatives and our cherry bomb class implements them in its ***der_func*** method. The q parameter is a state variable array that contains, in our case, the values of h and v (and t), and the dq parameter is the state variable derivative array. The method computes the values of $\dot{h}$ and $\dot{q}$ (and $\dot{t}$) and store then in the dq array. Equation 6.4 is a state event condition and it is implemented in two parts. The ***state_event_func*** method implements the 'if' part (left hand side) of the condition. Again, the supplied q array contains the current state variable values, h and v (and t) in this case. These are use to evaluate the state event condition and store the result in the z array. The simulator detects state events by looking for changes in the sign of the z array entries (i.e., from -1 to 0, 0 to 1, -1 to 1, and vice versa). The 'then' part (right hand side) is implemented with the ***discrete_action*** method, which the simulator invokes when the state event condition is true.

The cherry bomb has one discrete state variable with three possible values: the fuse is lit, the fuse is not lit, and the bomb is exploded. This variable changes in response to two events. The first event is when the bomb explodes; this is a time event that we know will occur 2 seconds from the time that the fuse it lit. Time ***time_event_func*** method is used to schedule the explosion by returning the time remaining until the fuse burns out. The ***time_event_func*** is similar to the familiar ***ta*** method; it is used to schedule autonomous events based on the current value of the model's state variables. The second event is an external event; this event is the fuse being doused with water. External events, of course, are not scheduled; they occur when and if the input event arrives.

---

[2] A cherry bomb is a small red firecracker. They are dangerous and illegal in the United States. None the less, every school seems to have at least one obnoxious kid who likes to put them into toilets.

[3] Cherry bomb fuses are frequently water proofed.

The ***discrete_action*** method implements the response of the cherry bomb to explosion and douse events in addition to the bounce event (i.e., the right hand side of Equation 6.4). The array q contains the values of the continuous state variables at the event time. The bounce and explosion events are both internal events and the input bag xb will be empty. The douse event is an input and it will appear in the input bag xb if the event occurs.

The cherry bomb model produces an output event when it explodes. The ***discrete_output*** method is used to implement the model's output behavior. As with the other methods, the q array contains the current value of the continuous state variables. The method fills the output bag yb with the model's output events (just as with the familiar ***output_func***). Because the cherry bomb is derived from the **Atomic** class, its output method is always invoked immediately prior to an internal event; internal events occur when the ***time_event_func*** duration expires or the ***state_event_func*** indicates that a state event condition is true.

The cherry bomb model can be implemented in two ways: as a sub-class of the **rk4** class or as a sub-class of the **rk45** class. The **rk4** class uses a fixed step size, fourth order Runge-Kutte integration scheme to solve the differential equations that describe a model's continuous dynamics; the **rk45** is an adaptive step size variant of the same scheme. Both schemes use a very simple interval bisection technique to locate state events[4] The only out difference between the **rk4** and **rk45** class is in their constructors; the **rk45** class requires one extra parameter that defines the error tolerance of the integration scheme.

The **rk45** derived cherry bomb model called **CherryBomb** is shown below. The base class constructor specifies five things: the number of continuous state variables (i.e., the size of the q and dq arrays), the largest integration time step that you will allow, the absolute error permitted at each integration step[5], the number of state event conditions (i.e., the size of the z array), and the time error tolerance for the event detection scheme (the default value is 10^{-12})[6] The **CherryBomb** constructor sets the initial value of its continuous variables h and v (and t) by using the **rk45**'s ***init*** method; its signature is

```
void init(int i, double q0)
```

The first parameter is the index (start from zero) of the continuous state variable and the second parameter is the variable's initial value. In this example, the initial height is 1 meter and the initial velocity is zero. The remainder of the **CherryBomb** implementation is just as described in the previous paragraphs.

```
#include "adevs.h"
#include <iostream>
using namespace std;
using namespace adevs;

// Array indices for the CherryBomb state variables
#define H 0
#define V 1
#define T 2
// Discrete variable enumeration for the CherryBomb
typedef enum { FUSE_LIT, DOUSE, EXPLODE } Phase;

class CherryBomb: public rk45<string> {
   public:
      CherryBomb():rk45<string>(
            3, // three state variables including time
```

---

[4] This state event detection in not particular robust; it can fail to detect a state event in some circumstances. You should be careful when employing it. Nonetheless, the scheme is sufficient in many cases.

[5] The error at each integration step is estimated by the integration algorithm and the step size is adjusted in an effort to keep the error at this tolerance. Beware that the actual error could be larger than your specified tolerance! As a rule of thumb I often set the error tolerance to one tenth of the value I actually want. For example, if I want errors less than 0.01, I'll use an error tolerance of 0.001.

[6] If this value is too small, then the simulator can get stuck.The default is 10^{-12} because that seems to be the smallest robust value when time is represented with a double precision floating point number.

```
            0.01, // maximum time step
            0.001, // error tolerance for one integration step
            1 // 1 state event condition
            ) {
         init(H,1.0); // Initial height
         init(V,0.0); // Initial velocity
         init(T,0.0); // Start time at zero
         phase = FUSE_LIT; // Light the fuse!
      }
      void der_func(const double* q, double* dq) {
         dq[V] = -9.8; // Equation 5.1
         dq[H] = q[V]; // Equation 5.2
         dq[T] = 1.0; // Equation 5.3
      }
      void state_event_func(const double* q, double *z) {
         // Test condition 5.4. The test uses h <= 0 instead of h = 0 to avoid
         // a problem if h, which is a floating point number, is not exactly 0.
         // For instance, it might be computed at 1E-32 which is close enough.
         if (q[H] <= 0.0 && q[V] < 0.0) z[0] = 1.0;
         else z[0] = -1.0;
      }
      double time_event_func(const double* q) {
         if (q[T] < 2.0) return 2.0 - q[T]; // Explode at time 2
         else return DBL_MAX; // Don't do anything after that
      }
      void discrete_action(double* q, const Bag<string>& xb) {
         if (xb.size() > 0 && phase == FUSE_LIT) phase = DOUSE; // Any input is a douse event
         else if (q[T] >= 2.0 && phase == FUSE_LIT) phase = EXPLODE; // Explode at time 2
         if (q[H] <= 0.0) q[V] = -q[V]; // Bounce
      }
      void discrete_output(const double *q, Bag<string>& yb) {
         if (q[T] >= 2.0 && phase == FUSE_LIT) yb.insert("BOOM!"); // Explode!
      }
      void state_changed(const double* q) {
         // Write the current state to std out
         cout << q[T] << " " << q[H] << " " << q[V] << " " << phase << endl;
      }
      void gc_output(Bag<string>&){} // No garbage collection is needed
      Phase getPhase() { return phase; } // Get the current value of the discrete variable
   private:
      Phase phase;
};

int main() {
   CherryBomb* bomb = new CherryBomb();
   Simulator<string>* sim = new Simulator<string>(bomb);
   while (bomb->getPhase() == FUSE_LIT)
      sim->execNextEvent();
   delete sim; delete bomb;
   return 0;
}
```

Figure 6.1 shows the cherry bomb trajectory from $t = 0$ to its explosion at $t = 2$. This plot was produced using the simulation program listed above. There is nothing particular surprising about it, but you can observe the discrete changes in the cherry bomb's trajectory. There are two bounce events at $t \approx 0.45$ and $t \approx 1.4$. The cherry bomb explodes abruptly at the start of its third decent.

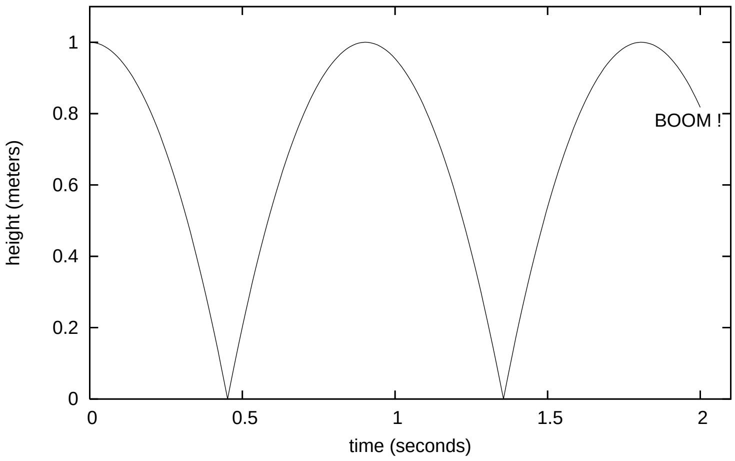

Figure 6.1: A simulation of the cherry bomb model that terminates when the cherry bomb explodes.

6.2 Building Numerical Integration Blocks for Adevs

The **rk45** and **rk4** classes are not directly derived from the **Atomic** class; they are actually derived from the **DESS** class, which is derived from the **Atomic** class. The **DESS** (Differential Equation System Specification) class[7] replaces the five familiar **Atomic** model methods with five new methods:

```
virtual void discrete_output_func(Bag<X>& yb)
virtual void discrete_action_func(const Bag<X>& xb)
virtual void evolve_func(double h)
virtual double next_event_func(bool& is_event)
virtual void state_changed()
```

Only the **Atomic** model's ***gc_output*** method is retained for the purposes of cleaning up objects created by the ***discrete_output_func*** method. The **rk4** and **rk45** classes were built by specializing these five methods; you can add new continuous system simulation algorithms to Adevs in the same way.

The method names are indicative of their function. The ***state_changed*** method is used to notify the derived class when a new state is computed. The method is intended as a tool for saving state trajectories as a simulation progresses, and as such it is not really essential for modeling. The ***state_changed*** method is invoked once at time zero (so that you can record the initial state), after every integration step, and before and after every discrete action (i.e., discrete state changes due to state, time, and input events).

The ***next_event_func*** method takes the place of the **Atomic** class's ***ta*** method. The ***next_event_func*** returns the smallest of the next integration step size and the time remaining until next internal event time. The integrations step size is selected by the derived class; it could be a fixed value or chosen dynamically to satisfy error or stability constraints. The next internal event time is also selected by the derived class; it can be the known discrete event time (i.e., at time event) or the time of the next state event. The is_event

---

[7] The classification of dynamic systems into discrete event, discrete time, and continuous models is formalized by Zeigler in his book "Theory of Modeling and Simulation". The acronyms for each class are DEVS (Discrete EVent system Specification), DTSS (Discrete Time System Specification), and DESS (Differential Equation System Specification).

flag is an output argument; its value must be true if the next event is a time or state event and false if it is just an integration step.

The ***evolve_func*** is responsible for advancing the model's continuous trajectories. The method's role is to integrate the differential or differential algebraic equations that describe the model's continuous motion. The argument h is the step size that must be used; h is always less than or equal to the value given by the ***next_event_func***. It is important to distinguish between the trial evaluations that might be required to pick the next integration step size and actually advancing the continuous trajectories. Many trial integration steps might be needed to implement the ***next_event_func***; these trial steps might use different steps sizes to find one the gives a tolerable error or to locate state events in the continuous solutions. But when an appropriate time step has been found and given to the simulator via the ***next_event_func*** the ***evolve_func*** may still require you to use a smaller (but never a larger) time step. Calculations used to find a value for the ***next_event_func*** are tentative; only the ***evolve_func*** can evolve the solution.

The ***discrete_action_func*** is responsible for making discrete changes to the system state in response to time, state, and input events. There are two conditions that cause this method to be invoked. The first condition is the ***next_event_func*** has set its is_event flag to true and the returned time expires without an input event. The second condition is the arrival of an input event prior to the ***next_event_func*** time expiring. In all cases the continuous variables are advanced to the event time by the ***evolve_func*** before the ***discrete_action_func*** is invoked (this is why integration step sizes suggested by the ***next_event_func*** are only tentative). If the first condition is true and the second condition is false then the input **Bag** xb will be empty; the discrete state change is autonomous. If the second condition is true, regardless of the first condition, then an input event has occurred and the input values are contained in the input **Bag**. If both conditions are true simultaneously the ***discrete_action_func*** is only invoked once, not twice. The ***discrete_action_func*** can change any of the model's state variables, continuous and discrete. The q array contains the model's continuous variables and changes to its elements change the corresponding continuous state variables.

The ***discrete_output_func*** is the counterpart to the ***Atomic*** class's ***output_func***. It is invoked whenever an autonomous event occurs and just prior to the invocation of the ***discrete_action_func***. Output values are placed by the model into the output **Bag** yb. The simulator will invoke the model's ***gc_output*** method when these objects can be safely deleted.

To illustrate the construction process, let's build a simple continuous system simulation module. Our simple modules will only allow for one event condition z and one state variable x whose behavior is described by the differential equation

$$\dot{x} = f(x, \bar{q})$$

where $\bar{q}$ are our discrete variables. The integration scheme will be the implicit Euler method with a fixed step size; state events will be detected by looking for points where z is equal to zero. Here is the header file for our new simulation module which we will call **ie** for implicit Euler. Its interface is similar to that of the **rk45** class, but with fewer parameters to the constructor and single variables, rather than arrays, for the event condition and state variable parameters.

```
#include "adevs_dess.h"
#include <cmath>

template <class X> class ie: public adevs::DESS<X> {
    public:
        /**
         * The constructor requires an initial value q0 for the continuous
         * state variable and a maximum step size h_max for the implicit Euler
         * integration scheme.
         */
        ie(double q0, double h_max):adevs::DESS<X>(),h_max(h_max),q(q0){}
        // Get the current value of the continuous state variable.
        double getStateVars() const { return q; }
        // Compute the derivative function using the supplied state variable value.
        virtual double der_func(double q) = 0;
        // Compute the value the zero crossing function.
        virtual double state_event_func(double q) = 0;
        // The discrete action function can set the value of q by writing to its reference.
        virtual void discrete_action(double& q, const adevs::Bag<X>& xb) = 0;
        // The discrete output function should place output values in yb.
        virtual void discrete_output(double q, adevs::Bag<X>& yb) = 0;
        virtual void state_changed(double q){};
        // Implementation of the DESS evolve_func method
        void evolve_func(double h);
        // Implementation of the DESS next_event_func method
        double next_event_func(bool& is_event);
        // Implementation of the DESS discrete_action_func method
        void discrete_action_func(const adevs::Bag<X>& xb);
        // Implementation of the DESS dscrete_output_func method
        void discrete_output_func(adevs::Bag<X>& yb);
        // Implementation of the DESS state_changed method
        void state_changed();
        /// Destructor
        ~ie(){}
    private:
        const double h_max; // Maximum integration time step
        double q; // Continuous state variable
        double integ(double qq, double h);
        // Return the sign of x
        static int sgn(double x) {
            if (x < 0.0) return -1;
            else if (x > 0.0) return 1;
            else return 0;
        }
};
```

The numerical integration scheme is implemented in the ***integ*** method; this method is called by the ***evolve_func*** method to advance the continuous solution and by the ***next_event_func*** to search for state events. The implicit scheme

$$x(t+h) = x(t) + hf(x(t+h), q(t)) \tag{6.5}$$

requires that we search for a next value of x that satisfies Equation 6.5. This is a fixed point problem and it can be seen most clearly if write $\tilde{x} = x(t+h)$, $g(\tilde{x}) = x(t) + hf(\tilde{x})$, and then state the problem as finding a value for $\tilde{x}$ such that

$$\tilde{x} = g(\tilde{x})$$

A simple solution method is to start with the initial guess $\tilde{x}_0 = x(t)$ and then compute successive guesses $\tilde{x}_1$, $\tilde{x}_2$, ... by

$$\tilde{x}_{i+1} = g(\tilde{x}_i) \tag{6.6}$$

until the difference between $\tilde{x}_{i+1}$ and $\tilde{x}_i$ is small. If this works, the sequence of $\tilde{x}_i$'s will converge to a single value, the this value is the solution that we are looking for and the final $\tilde{x}_i$ is used for $x(t+h)$ in Equation 6.5. Here is the implementation of the ***integ*** method and its trivial use by the ***evolve_func*** method to advance to continuous solution.

```
template <class X>
double ie<X>::integ(double qq, double h) {
   double q1 = qq;
   double q2 = qq + h*der_func(q1);
   while (fabs(q1-q2) > 1E-12) {
      q1 = q2;
      q2 = qq + h*der_func(q1);
   }
   return q2;
}

template <class X>
void ie<X>::evolve_func(double h) {
   q = integ(q,h);
}
```

The ***discrete_action_func***, ***discrete_output_func***, and ***state_changed*** methods are very simple; they just pass on the current value of the single continuous state variable to corresponding methods of the derived class. The continuous state variable is always up to date because the **DESS** base class calls the ***evolve_func*** before invoking the **ie** class's ***discrete_output_func***, ***discrete_action_func***, or ***state_changed*** methods. Here are the method implementations.

```
template <class X>
void ie<X>::discrete_action_func(const adevs::Bag<X>& xb) {
    discrete_action(q,xb);
}

template <class X>
void ie<X>::discrete_output_func(adevs::Bag<X>& yb) {
    discrete_output(q,yb);
}

template <class X>
void ie<X>::state_changed() {
    state_changed(q);
}
```

All the remains is to implement the ***next_event_func***. This method returns the smaller of our maximum integration time step h_{max} (hmax in the source code) and zero crossing of the state event function z. The state event detection problem is a root finding problem; we want to a value of $x(\zeta)$ such that $z(x(\zeta)) = 0$ and $\zeta \in [t, t + h_{max}]$. If such a point exists, that an event occurs at time ζ, otherwise there are no events in the interval. We'll use a relatively simple method for finding these event points. Assume that z is a line and let δh be the width of the time interval that we are considering. Initial we take $\delta h = h_{max}$, corresponding to the time interval $[t, t + h_{max}]$. We computing z at $x(t)$ and $x(t + \delta h)$ and look to see if its sign has changed. If the answer is no, then there is no event in the interval. Otherwise by assuming that z is the line

$$z(x(t+\tau)) = \frac{z(x(t+\delta h) - z(x(t))}{\delta h}\tau + z(x(t))$$

we can determine the time τ until the next event as

$$\tau = \frac{z(x(t))h}{z(x(t)) - z(x(t+\delta h))}$$

We then set δh to τ and repeat this procedure until either the interval $[t, t + h_{max}]$ does not contain an event or the value of z is suitable small. The ***next_event_func***, which implements this procedure, is shown below. Notice that it uses the ***integ*** method to compute trial values of x.

```
template <class X>
double ie<X>::next_event_func(bool& is_event) {
    double h = h_max;
    double z1 = state_event_func(q);
    double z2 = state_event_func(integ(q,h));
    while (true) {
        if (sgn(z1) == sgn(z2)) {
            is_event = false;
            break;
        }
        else if (sgn(z1) != sgn(z2) && fabs(z2) < 1E-12) {
            is_event = true;
            break;
        }
        h = (h*z1)/(z1-z2);
        z2 = state_event_func(integ(q,h));
    }
    return h;
}
```

Let's demonstrate our new integration scheme on a simple problem whose solution can be worked by hand. Consider a bucket that is being filled with liquid. The bucket is equipped with a computer controlled value that sense the volume of liquid in the bucket and drains it when the volume is v_{max}; our bucket model produces an output event when this occurs. If the spigot that hangs over the bucket is open, then the bucket fills at an exponentially decaying rate (to avoid overfilling); if the spigot is closed then the bucket stops filling. This model has one discrete variable that describes the spigot and one continuous variable that describes the volume of fluid in the bucket. There is a single state event condition that causes the volume to be set to zero when it reaches v_{max}. We'll assume the bucket has an absolute capacity of 1 unit and the computer drains the bucket if the volume reaches 0.75 units. The bucket's dynamics can be written as

$$\dot{v} = \begin{cases} 0 & \text{if the spigot is closed} \\ 1 - v & \text{if the spigot is open} \end{cases}$$

$$v \geq 0.75 \implies v \leftarrow 0$$

where v is the liquid volume, $\implies$ is logical implication, and $\leftarrow$ is an assignment. An output event always occurs when the state event condition is satisfied. The bucket model implemented with our new **ie** class is shown below.

```
#include "ie.h"
#include "adevs.h"
#include <iostream>
using namespace std;
using namespace adevs;

double t = 0.0; // Global simulation time variable that is set in the main simulation loop

class bucket: public ie<bool> {
    public:
        // The initial volume is 0, the integration time step is 0.01, the spigot is closed
        bucket():ie<bool>(0.0,0.01),spigot_open(false){}
        double der_func(double q) { return spigot_open*(1.0-q); }
        double state_event_func(double q) { return 0.75-q; }
        void discrete_action(double& q, const Bag<bool>& xb) {
            if (q >= 0.75) q = 0.0;
            if (xb.size() > 0) spigot_open = *(xb.begin());
        }
        void discrete_output(double q, Bag<bool>& yb) {
            if (q >= 0.75) yb.insert(true);
        }
        void state_changed(double q) {
            cout << t << " " << q << " " << spigot_open << endl;
        }
        void gc_output(Bag<bool>&){}
    private:
        bool spigot_open;
};
```

When the bucket is initially empty the system has a periodic trajectory

$$v(t) = 1 - \exp(-t) \quad \text{where } t \in [0, -\ln(0.25)]$$

that begins when the spigot is opened and repeats itself by setting v and t to zero every $-ln(0.25) \approx 1.37$ units of time. The exact and simulated trajectories are shown in Figure 6.2 for the case where the spigot is opened at $t = 1$ and closed at $t = 4$. The implicit Euler simulation can be seen to lag slightly behind the exact solution. This is due to the relatively poor accuracy of the implicit Euler method and not an error in our implementation. For comparison, I conducted the same simulation using the more accurate ***rk4*** class in place of our ***ie*** class; the improvement is readily apparent following the spigot closing at time 4, but less evident elsewhere.

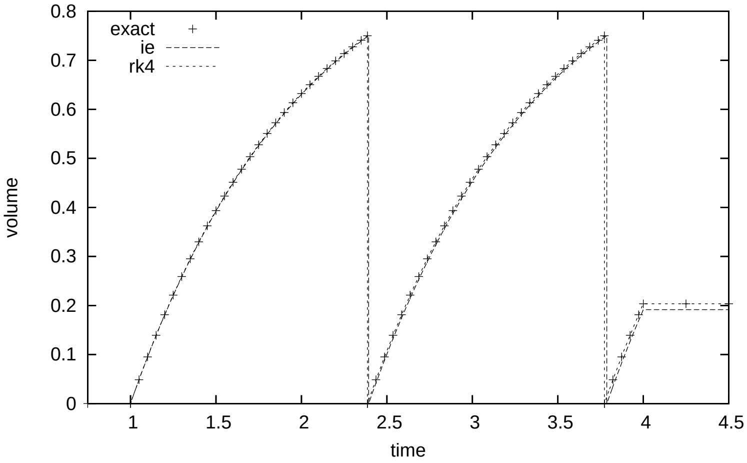

Figure 6.2: Volume of the bucket as a function of time when the spigot is opened at $t = 1$ and closed at $t = 4$.

Chapter 7

The Simulator Class

The functionality of the **Simulator** class is broken into three pieces: determining the model's next event time, extracting output from the model, injecting external input into the model, and advancing the simulation clock. The first piece of functionality is provided by the ***nextEventTime*** method with which we are already familiar. I'll address the three remaining pieces in turn.

There are two essential steps for extracting output from your model. The first step is to register an **EventListener** with the simulator. The **EventListener**'s ***outputEvent*** method is used to intercept output originating from **Atomic** and **Network** models; this first step is already familiar to us. The second step is to invoke the **Simulator**'s ***computeNextOutput*** method, which performs the output calculations and provides the results to registered **EventListener**s.

The ***computeNextOutput*** method is probably not familiar; we have not had cause to discuss it until now. The method signature is

```
void computeNextOutput()
```

and it computes the model output at the next event time given by the ***nextEventTime*** method. The ***computeNextOutput*** method invokes the ***output_func*** method of every imminent **Atomic** model, maps outputs to inputs by calling the ***route*** method of **Network** models, and calls the ***outputEvent*** method of every **EventListener** that has been registered with the **Simulator**. This method anticipates the outcome of your model from its current state assuming that no input events will intervene between now and the ***nextEventTime***.

The ***computeNextState*** method is used to inject intervening events and advance the simulation clock. The method signature is

```
void computeNextState(Bag<Event<X> >& input, double t)
```

where the **Event** class is the same one that the **EventListener** accepts to its ***outputEvent*** method. The **Event** class has two fields: a pointer to a model of type **Devs<X>** (i.e., a **Network** or **Atomic** model) and a value of type X.

The ***computeNextState*** method applies a bag of input **Event**s to the model at time t. If the input bag is empty and t is equal to the next event time, then this method has the same effect as ***execNextEvent***: it calculates the output values at time t using the ***computeNextOutput*** method, computes the next state of all models undergoing internal and external events, computes structure changes, and advances the simulation clock. If the input bag is not empty then the value of each **Event** is applied as an input to the model pointed to by that **Event**. If, in this case, t is equal to the next event time then the method also follows the usual steps of invoking the ***computeNextOutput*** method and calculating state and structure changes; if t is less than the **Simulator**'s next event time, then the procedure is nearly identical excepting that the ***computeNextOutput*** method is not invoked. In this case, the only input events for any model are those provided in the input bag.

The **Simulator**'s ***execNextEvent*** method, the workhorse of most simulation programs, actually defers its job to two these two methods which do the real work. The implementation takes only two lines; the **Bag** bogus_input is empty.

```
void execNextEvent() {
    computeNextOutput();
    computeNextState(bogus_input,nextEventTime());
}
```

The **Simulator**'s ***computeNextOutput***, ***computeNextState***, and ***execNextEvent*** methods throw an exception if a model violates either of two constraints: the time advance is negative or the coupling constraints described in section 4.1 and illustrated in Figure 4.4, are violated. The Adevs **exception** class is derived from the standard C++ **exception** class; the method ***what*** returns a string that describes the exception condition and the method ***who*** returns a pointer to the model that caused the exception to be generated. The Adevs **exception** class is intended to assist with model debugging. There isn't much you can do at run-time to fix a time advance method or reorganize a model's structure (or fix the structure change logic), but the simulator tries to be friendly by identifying a problem before it becomes an obscure and difficult to find bug.

Chapter 8

Models with Many Input/Output Types

It would be surprising if every component in a large model (or even a small one) had the same input and output requirements. Some models can be satisfactorily constructed with a single type of input/output object and, if this is the case, it will simplify the design of your simulator. If not, you'll need to address this problem when you design your simulation program.

One solution to this problem is to establish a base class for all input and output types and derive specific types from the common base. The simulator and all of its components exchange pointers to the base class and down cast specific objects as needed. The C++ dynamic_cast operator is particularly useful for this purpose. Although it is not without its problems, I have used this solution in many designs and it works well.

It is not always possible for every component in a model to share a common base class for its input and output type. This can happen if different sub-model have very different input and output needs or when models from earlier projects are reused. For example, to use a **CellSpace** model as a component of a **Digraph** model requires some means of converting the **CellSpace**'s **CellEvent** objects into the **PortValue** objects required by the **Digraph**. Happily, there is a simple solution to this problem that makes clever use of the **Simulator** and **EventListener** classes to wrap a model with one input and output type inside of an atomic model with a different input and output type.

The Adevs **ModelWrapper** class is an **Atomic** model that encapsulates another model. The encapsulated model can be a **Network** or **Atomic** model. The **ModelWrapper** uses input/output objects of type ExternalType, but the encapsulated class uses input/output objects of type InternalType. Two abstract methods are provided for converting objects with one type into objects with the other type; these methods are

```
void translateInput(const Bag<ExternalType>& external_input, Bag<Event<InternalType> >& internal_input)
void translateOutput(const Bag<Event<InternalType> >& internal_output, Bag<ExternalType>& external_outp
```

Clean up of converted objects are managed with the ***gc_output*** method, which is inherited from the **ModelWrapper**'s **Atomic** base class, and a new ***gc_input*** method for cleaning up objects created by the ***translateInput*** method; its signature is

```
void gc_input(Bag<Event<InternalType> >& g)
```

The model to encapsulate is passed to the **ModelWrapper** constructor. The **ModelWrapper** creates a **Simulator** for the model that is used to control its evolution; the **ModelWrapper** is a simulator inside of a model inside of a simulator! The **ModelWrapper** keeps track of the wrapped model's last event time, and it uses this information and the **Simulator**'s ***nextEventTime*** method to compute its time advance. Internal, external, and confluent events cause the **WrappedModel** to invoke its **Simulator**'s

computeNextState method and thereby advance the state of the wrapped model. Internal events are simplest; the ***computeNextState*** method is invoked with the wrapped model's next event time and an empty input bag.

The ***delta_conf*** and ***delta_ext***, however, must convert the incoming input events, which have the type ExternalType, into input events for the wrapped model, which have the type InternalEvent. This is accomplished with the ***translateInput*** method. The first argument to the method is the input bag passed to the **ModelWrapper**'s ***delta_ext*** or ***delta_conf*** method. The second argument is an empty bag that the method implementation must fill. When the ***translateInput*** method returns this bag will be passed to the ***computeNextState*** method of the **ModelWrapper**'s simulator. Notice that the internal_input argument is a **Bag** filled with **Event** objects; if the wrapped model is a **Network** then the translated events can be targeted at any of the **Network**'s components. The **ModelWrapper** invokes the ***gc_input*** method when it is done with the events in the internal_input bag. This gives you the opportunity to delete objects that you created when ***translateInput*** was called.

A similar process occurs when the **ModelWrapper**'s ***output_func*** is invoked, but in this case it is necessary to convert output objects from the wrapped model, which have type InternalType, to output objects from the **ModelWrapper**, which have type ExternalType. This is accomplished by invoking the ***translateOutput*** method. The method's first argument is the bag of output events produced collectively by all of the wrapped model's components; notice that the contents of the internal_output bag are **Event** objects. The model field points to the component of the wrapped model (or the wrapped model itself) that produced the event and the value field contains an output object produced by that model. These **Event**s must be converted to objects of type ExternalType and stored in the external_output bag. The external_output bag is, in fact, the bag passed to the wrapper's ***output_func***, and so its contents become the output objects produced by the wrapper. The ***gc_output*** method is used in the usual way to clean up any objects created by this process.

The **Wrapper** class shown below illustrates how to use the Adevs **WrapperModel** class. The **Wrapper** is derived from the **WrapperModel** and implements its four abstract methods: ***translateInput***, ***translateOutput***, ***gc_input***, and ***gc_output***. This class wraps an **Atomic** model that uses int* objects as its input/output. The **Wrapper** uses C strings as its input and output type. The translation methods merely convert integers to strings and vice versa. The **Wrapper** can be used just like any **Atomic** model; it can be a component in a network model or simulated by itself. The behavior of the **Wrapper** is identical to the model it wraps; the only change is in the interface.

```
// This class converts between char* and int* event types.
class Wrapper: public adevs::ModelWrapper<char*,int*> {
    public:
        Wrapper(adevs::Atomic<int*>* model):
            // Pass the model to the base class constructor
            adevs::ModelWrapper<char*,int*>(model){}
        void translateInput(const adevs::Bag<char*>& external,
                adevs::Bag<adevs::Event<int*> >& internal) {
            // Iterate through the incoming events
            adevs::Bag<char*>::const_iterator iter;
            for (iter = external.begin(); iter != external.end(); iter++) {
                // Convert each one into an int* and send it to the
                // wrapped model
                adevs::Event<int*> event;
                // Set the event value
                event.value = new int(atoi(*iter));
                // Set the event target
                event.model = getWrappedModel();
                // Put it into the bag of translated objects
                internal.insert(event);
            }
        }
        void translateOutput(const adevs::Bag<adevs::Event<int*> >& internal,
                adevs::Bag<char*>& external) {
            // Iterate through the incoming events
            adevs::Bag<adevs::Event<int*> >::const_iterator iter;
            for (iter = internal.begin(); iter != internal.end(); iter++) {
                // Convert the incoming event value to a string
                char* str = new char[100];
                sprintf(str,"%d",*((*iter).value));
                // Put it into the bag of translated objects
                external.insert(str);
            }
        }
        void gc_output(adevs::Bag<char*>& g) {
            // Delete strings allocated in the translateOutput method
            adevs::Bag<char*>::iterator iter;
            for (iter = g.begin(); iter != g.end(); iter++)
                delete [] *iter;
        }
        void gc_input(adevs::Bag<adevs::Event<int*> >& g) {
            // Delete integers allocated in the translateInput method
            adevs::Bag<adevs::Event<int*> >::iterator iter;;
            for (iter = g.begin(); iter != g.end(); iter++)
                delete (*iter).value;
        }
};
```

Chapter 9

Random Numbers

Adevs has two classes that work together to generate many types of random numbers. These two classes are the **random_seq** class and the **rv** class. The **random_seq** class provides uniformly distributed random numbers to the **rv** class, and the **rv** transforms this uniform stream of random numbers into a variety of random number distributions.

The **random_seq** class is an interface for a random number generator. Its derived classes produce uniformly distributed pseudo-random numbers. The underlying random number stream is accessed with two methods. The ***next_long*** returns a raw random number as an unsigned long. The ***next_dlb*** refines the ***next_long*** method by reducing the raw random number to a double precision number in the interval $[0, 1]$. The random number sequence is initialized with the ***set_seed*** method, and the entire random number generator can be copied with the ***copy*** method. To summarize, the **random_seq** class has four abstract methods

```
void set_seed(unsigned long seed)
double next_dbl()
random_seq* copy() const
unsigned long next_long()
```

that must be implemented by any derived class.

Adevs comes with two implementations of the **random_seq** class: the **crand** class and the **mtrand** class. The **crand** class uses the rand function from the standard C library to implement the required methods. Its implementation is trivial; I've listed it below as an example of how to implement the **random_seq** interface.

```
class crand: public random_seq {
    public:
        /// Create a generator with the default seed
        crand(){}
        /// Create a generator with the given seed
        crand(unsigned long seed) { srand (seed); }
        /// Set the seed for the random number generator
        void set_seed(unsigned long seed) { srand (seed); }
        /// Get the next double uniformly distributed in [0, 1]
        double next_dbl() { return (double)rand()/(double)RAND_MAX; }
        /// Copy the random number generator
        unsigned long next_long() { return (unsigned long)rand(); }

        random_seq* copy() const { return new crand (); }
        /// Destructor
        ~crand(){}
};
```

The **mtrand** class implements the Mersenne Twister random number generator[1]. The code is based on their open source implementation of the Mersenne Twister. Aside from its potential advantages as a random number generator, the **mtrand** class differs from the **crand** class by its ability to make deep copies; every instance of the **mtrand** class has its own random number stream.

The **rv** class uses the uniform random numbers provided by a **random_seq** object to produce several different random number distributions: triangular, uniform, normal, exponential, lognormal, Poisson, Weibull, binomial, and many others. Every instance of the **rv** class is created with a **random_seq**; the default is an **mtrand** object, but any type of **random_seq** object can be passed to the **rv** constructor. The different random distributions are sampled by calling the appropriate method: ***triangular*** for a triangular distribution, ***exponential*** for an exponential distribution, ***poisson*** for a Poisson distribution, etc. Adevs is open source software; if a new distribution is needed then you can add a method that implements it to the **rv** class (and, I hope, contribute the expansion to the Adevs project).

---

[1] M. Matsumoto and T. Nishimura, "Mersenne Twister: A 623-Dimensionally Equidistributed Uniform Pseudo-Random Number Generator", ACM Transactions on Modeling and Computer Simulation, Vol. 8, No. 1, January 1998, pgs. 3-30.

Chapter 10

Interpolation

Data is often made available as a set of tabulated points. Hourly temperature data, millisecond samples of a voltage signal in an electric circuit, and a minute by minute record of a radar track are examples of continuous signals recorded at discrete points in time. But if you need temperate data every half hour, the circuit voltage a fractions of a millisecond, or track information at quarter minutes then a method for approximating values between tabulated points will be very useful. The **InterPoly** class exists for this purpose.

The **InterPoly** class approximates smooth continuous signal by fitting a polynomial to the available data points. The approximating polynomial is guaranteed to pass through every available data point, and in many cases it provides a reasonable approximation to the original signal between the available data points. The most familiar example of an interpolating polynomial is a line that connects two data points (t_1, x_1) and (t_2, x_2); the connecting line is

$$p(t) = \frac{t - t_2}{t_1 - t_2} x_1 + \frac{t - t_1}{t_2 - t_1} x_2$$

and it is easy to check that $p(t_1) = x_1$ and $p(t_2) = x_2$. If more data points are available then quadratic, cubic, quartic, and even higher degree polynomials can be used to obtain (in principle) better approximations.

An interpolating polynomial can be constructed with the **InterPoly** class in three ways. The first way is to provide the sample data to the **InterPoly** constructor

```
InterPoly(const double* u, const double* t, unsigned int n)
```

where u is an array of data values,t is an array of associated time points, and n is the number of data points (i.e., the size of the u and t array). The constructor will build an $n - 1$ degree polynomial that fits the supplied data. The second way is supply just the data values, a time step, the first time value, and number of data points to the constructor

```
InterPoly(const double* u, double dt, unsigned int n, double t0 = 0.0)
```

where u is an array of data values, dt is the time spacing of the data points, n is the number of data points, and t0 is the time instant of the first data point (i.e., the data point i is at time $t_0 + i \cdot dt$). Both constructors make copies of the supplied arrays, and changes to the array values will not be reflected by the **InterPoly** object. The third way is to assign new data point values to an existing polynomial by calling the **InterPoly** method

```
void setData(const double* u, const double* t = NULL)
```

where u is the new set of data values and t is (optionally) the new set of time points. This method requires that the number of data points in u (and t if used) be equal to the number of points supplied to the **InterPoly** constructor.

There are three methods for computing interpolated values: the ***interpolate*** method, the overloaded ***operator()***, and the ***derivative*** method. The method signatures are listed below:

```
double interpolate(double t) const
double operator()(double t) const
double derivative(double t) const
```

The ***interpolate*** method and ***operator()*** give the value of the interpolating polynomial at the time point t. The ***derivative*** method gives the value of the first time derivative of the interpolating polynomial as an approximation of the first time derivative of the original function. For example, if the data describes the position of an object through time then the ***derivative*** method gives and approximation of the object's velocity.

To demonstrate the **InterPoly** class and give you a sense of what the interpolating polynomials look like, I've listed below a program that computes $\sin(t)$, its time derivative $\cos(t)$, and interpolated approximations of both. Interpolating polynomials built with 4, 5, and 6 data point in the interval $[0, 2\pi]$ are illustrated in Figs. 10.1 and 10.2. The quality of the approximation generally improves as more data points are added, but the function and interpolating polynomial diverge significantly outside of the interval spanned by the data points. Be careful if you extrapolate!

```
#include "adevs.h"
#include <cmath>
#include <iostream>
using namespace std;

int main(int argc, char** argv) {
    // Get the number of data points to use and allocate
    // memory for the data points arrays
    int N = atoi(argv[1]);
    double* t = new double[N];
    double* u = new double[N];
    // Compute data points using the sin function
    for (int i = 0; i < N; i++) {
        t[i] = i*(2.0*3.14159/(N-1));
        u[i] = sin(t[i]);
    }
    // Create the interpolating polynomial
    adevs::InterPoly p(u,t,N);
    // The data arrays can be deleted now
    delete [] t; delete [] u;
    // Compute several points with sin, its derivative, and the polynomial
    // inside and a little beyond the interval spanned by the data
    for (double t = 0; t < 2.0*3.14159+0.5; t += 0.01)
       cout << t
           << " " << sin(t) << " " << p(t)
           << " " << cos(t) << " " << p.derivative(t)
           << endl;
    // Done
    return 0;
}
```

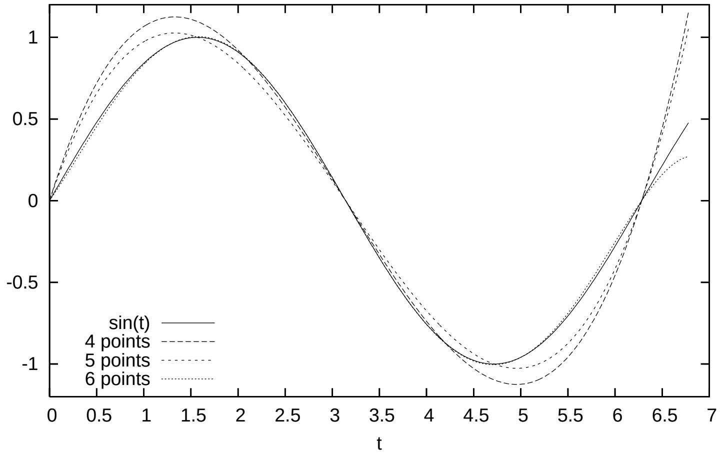

Figure 10.1: The function $\sin(t)$ and some interpolating polynomials with data spanning the interval $[0, 2\pi]$.

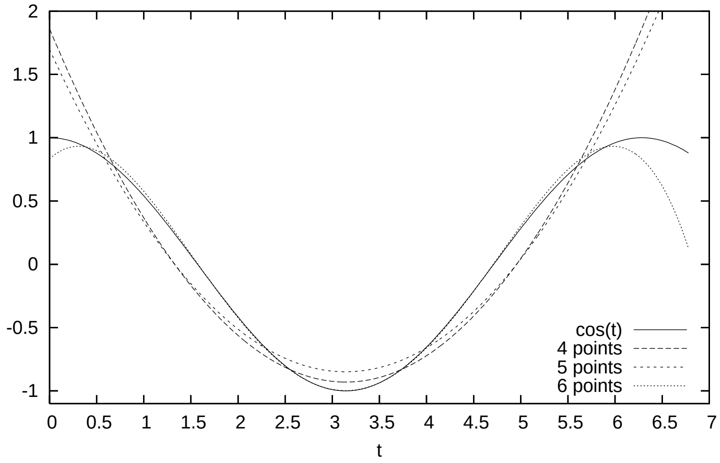

Figure 10.2: The time derivative of $\sin(t)$ and the time derivative of some interpolating polynomials with data spanning the interval $[0, 2\pi]$.