# A Discrete EVent system Simulator

Jim Nutaro

September 29, 2015

# Contents

# Chapter 1

# About this manual

The purpose of this manual is to get you working with Adevs as quickly as possible. To that end, this manual documents the major features of the simulation engine with an emphasis on how they are used. The table of contents summarizes which aspects of the simulator are described here.

Among the features omitted from this manual are the Java language bindings for the Adevs simulator. Build instructions for the Java bindings are given in the "Build and Install" section. How these bindings are used will (I hope) be obvious once you have perused the C++ interface: the interfaces for building models and running simulations with Java are essentially the same as with C++.

The Java bindings have three limitations. First, you pay a (usually unnoticeable) cost in execution time for the extra work that Adevs must to do manage memory associated with input and output objects and models that are orphaned during a change of structure. Second, the facilities for combined simulation of discrete event and continuous models are not implemented for Java. Third, this is not a 'pure Java' simulation engine: it uses a great deal of native code to do its work (though this is invisible to the programmer).

There are at least two positive aspects of the Java bindings. The first is it omits the need for explicitly managing memory. The Java garbage collector (plus some extra work by the simulation engine) takes care of this for you. Second, you have access to the nice features and standard libraries of the Java programming language.

Other topics not included in this manual are theory (why the simulator is built as it is) and some experimental features of the simulation engine. Among the latter are support for simulating hybrid differential-algebraic systems and conservative, parallel simulation using multicore processors. If you are interested in any of these subjects, I offer the following (greatly abridged) list of books and articles:

1. A. M. Uhrmacher. Dynamic structures in modeling and simulation: a reflective approach, ACM Transactions on Modeling and Computer Simulation, Vol. 11, No. 2 , pp. 206-232. April 2001. This paper describes the approach used by Adevs to model and simulate dynamic structure systems.

2. Bernard P. Zeigler, Tag Gon Kim and Herbert Praehofer. Theory of Modeling and Simulation, Second Edition. Academic Press. 2000. This book develops the Discrete Event System Specification (DEVS) from its roots in abstract systems theory.

3. James J. Nutaro. Building Software for Simulation: Theory and Algorithms, with Applications in C++. Wiley. 2010. This book presents the Discrete Event Systems Specification with code for the (slightly abridged) Adevs simulator and has several examples of its use. This book also describes the conservative, parallel simulator and discusses the construction of new ODE solvers and event finding modules for Adevs.

Question and comments about this software can be sent to Jim Nutaro at nutarojj@ornl.gov.

# Chapter 2

# Building and Installing

The Adevs package is organized into the following directory structure:

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

The Adevs simulation engine is comprised almost entirely of template classes, and these are implemented in the header files located in the *include* directory. The exceptions are the random number generators, the Java language bindings, and some aspects of the parallel simulation engine. If you do not want to use these features then its sufficient for your program to include *adevs.h* and to be sure your compiler can find the *include* directory that *adevs.h* is in.

If you want to use the random number generators or parallel simulation engine, then you must build the adevs static library. To do this, enter the *src* directory and execute the command 'make' if you are using a Linux system or 'build' if you are using Windows. On Windows, the batch file creates a static library called *adevs.lib*. For Linux systems, the makefile creates a static library called *libadevs.a*.

If you are using a Windows system, the batch file must be executed from the Visual C++ command prompt. This will ensure the batch file can find the compiler, linker, and necessary system header files. For Linux systems, make sure you have a recent version of the GNU C++ compiler and GNU make. You may need to edit the makefile (i.e., the file *Makefile*) to set compiler flags, etc. but the defaults should work in most cases.

To build the Java language bindings, you need to have the Oracle JDK or something that is compatible with it (such as the OpenJDK). On a Windows system, from the *src* directory enter the *adevs_jni* directory and then execute the command 'build'. This creates three files: *adevs.jar*, *java_adevs.dll*, and *java_adevs.lib*. To build and run your Java programs, you will need to put *adevs.jar* into your classpath and *java_adevs.dll* into your java.library.path (or make sure it is in your regular PATH for finding executables and dynamic link libraries).

On a Linux system, stay in the *src* directory and execute the command 'make java_adevs'. This creates two files: *adevs.jar* and *libjava_adevs.so*. As before, you need to put *adevs.jar* into your classpath and *libjava_adevs.so* into your java.library.path or make sure it is in your LD_LIBRARY_PATH for locating dynamic link libraries.

Adevs includes some support for simulating importing models that support the Functional Mockup Interface for Model Exchange standard (models built with most Modelica based tools support this standard). If you want to experiment with this feature you can get and build the OpenModelica compiler. For this purpose, the shell script *build-omc.sh* is provided in the *util* directory. This shell script does the following: 1) creates

the directory *openmodelica* at the location where the script is run, 2) fetches the minimal set of packages that are needed to compile the OpenModelica compiler, and 3) fetches and builds a bare-bones OpenModelica compiler. The compiler is called *omc* and it is located in the directory *openmodelica/trunk/build/bin*. You will also need to get the FMI header files from `https://www.fmi-standard.org/downloads` and include *adevs_fmi.h* in your program files.

To run the test suite (which is not required to use the software), you must build the static 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). If you want to test the Java bindings then you will need to build them; to test the FMI support you will need the OpenModelica compiler. There are four sets of tests that can be executed: one for the serial simulation engine, one for the parallel simulation engine, one for the Java language bindings, and one for the FMI support.

To run the tests for the serial simulation engine, use 'make check_cpp'. To run the tests for the parallel simulation engines use 'make check_par' (note: the environment variable OMP_NUM_THREADS must be set to at least four). To run the Java test cases, use 'make java_test'. To run the FMI test cases, put the omc compiler into your PATH and run 'make check_fmi'. To run all of the test cases, use 'make'. The test script will abort when any test fails. If the test script run to completion, then all of the tests passed.

The test cases can be a bit of a bear to run on a Windows computer. If you need to edit compiler settings, executable directives, etc. to make it work, then modify the file *make.common*. For Linux systems using the GNU tools the test cases should work out of the box. Otherwise, edit *make.common* to fix things to fit your development environment.

# Chapter 3

# 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 with Adevs) is that their dynamic behavior is defined by events. An event is any change that is significant within the context of the model being developed.

The modeling of discrete event systems is most easily introduced with an example. Suppose we want to model the checkout line at a convenience store. There is a single clerk who serves customers in a first come-first served fashion. The time required for the clerk to ring up each customer's bill depends on the number of items purchased. We are interested in determining the average and maximum amount of time that customers spend waiting in line at the clerk's counter.

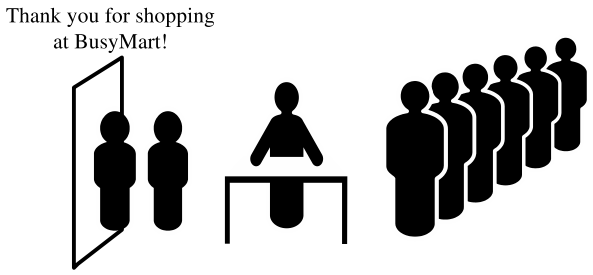


Figure 3.1: Customers waiting in line at BusyMart.

To simulate this system, we need an object to represent each customer in the line. A class called **Customer** is created for this purpose. Each customer object has three attributes. One attribute is the time needed to ring up the customer's bill. Because we want to know how long a customer has been waiting in line, we also include two attributes that record when the customer entered the line and when the customer left the line. The difference of these times is the amount of time the customer spent waiting in line. Below is the code for the customer class. This code is in a single header file called *Customer.h*.

```
#include "adevs.h"
/// A Busy-Mart customer.
struct Customer
{
    /// Time needed for the clerk to process the customer
    double twait;
    /// Time that the customer entered and left the queue
    double tenter, tleave;
```

[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.

```
};
/// Create an abbreviation for the Clerk's input/output type.
typedef adevs::PortValue<Customer*> IO_Type;
```

Customers are served (processed) by the clerk, which is our first example of an atomic model. The clerk has a line of people waiting at her counter. When a customer is ready to make a purchase, that customer enters the end of the line. If the clerk is not busy and the line is not empty, then the clerk rings up the bill of the customer that is first in line. That customer then leaves the line and the clerk processes the next customer or, if the line is empty, the clerk sits idly at her counter.

The DEVS model of the clerk is as follows. First, we specify the type of object that the model consumes and produces. For this model, we use **PortValue** objects. The **PortValue** class describes a port-value pair. In this case, **Customer** objects are the value and they appear at the clerk's "arrive" port. Customers depart via the clerk's "depart" port. Second, we specify the state variables that describe the clerk. In this case, the state comprises the customers that are in line. We use a **list** from the C++ Standard Template Library for this purpose.

To complete the model of the clerk, we implement the four methods that model her behavior. First, lets construct the header file for the clerk. Then we can 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 an archetype for almost any atomic model that we want to create. The **Clerk** class is derived from the Adevs **Atomic** class. The **Clerk** implements six virtual methods that it inherits from **Atomic**. These are the state transition functions delta_int, delta_ext, and delta_conf; the output function output; the time advance function ta, and the garbage collection method gc. The **Clerk** also has 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 its **Atomic** base class. The template argument of the base class defines the type of object that the clerk uses for input and output. The **Clerk** state variables are defined as private class attributes. These are the list of customers (line), the clerk's clock (t), and the time spent so far on the first customer in line (t_spent).

The ports "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 to 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 starts with an empty line, the only interesting thing that can happen is for a customer arrive. Arriving customers appear on the clerk's "arrive" input port. The arrival of a customer causes the clerk's external transition method to be invoked. The arguments to this method are the time that has elapsed since the clerk last changed state and a bag of **PortValue** objects.

The external transition method updates the clerk's clock by adding to it the elapsed time. The time spent working on the current customer's order is updated by adding the elapsed time to the time spent so far. After updating these values, the input events are processed. Each **PortValue** object has two attributes. The first is the port. It contains the number of the port that the event arrived on and is equal to "arrive" in this case. The second is the **Customer** that arrived. The clerk records the time of arrival for the new customer and places him 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 describes the amount of time that will elapse before the clerk's next internal (self, autonomous) event, barring an input that arrives in the interim. In this case, the time advance is the time remaining for the clerk to process the current customer. If there are no customers in line, then the clerk will not do anything and so the time advance returns infinity (in Adevs represented by DBL_MAX). Otherwise, the clerk's next action is when the first customer in line has been rung up, and so the time advance is the difference of the **Customer**'s twait and the clerk's t_spent.

```
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;
}
```

Two things happen when the clerk finishes ringing up a customer. First, the clerk sends that customer on his way. This is accomplished by the clerk's output_func method, which is invoked when the time advance expires. The output_func method places the departing customer onto the Clerk's "depart" port by creating a **PortValue** object and putting it into the bag yb of output objects. The clerk's output_func method is shown below.

```
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;
}
```

Second, the clerk begins to process the next customer in the line. If, indeed, there is another customer waiting in line, then the clerk begins ringing that customer. Otherwise, the clerk becomes idle. These actions are accomplished by the **Clerk**'s internal transition method, which is called immediately after the output_func method. The **Clerk**'s internal transition method updates the Clerk's clock and removes the departing customer from the line. The code for this method is 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;
    cout << "Clerk: The next customer will leave at t = " << t+ta() << "." << endl;
}
```

We have almost completed the model of the clerk; only one thing remains to be done. Suppose a customer arrives at the clerk's line at the same time that the clerk finishes 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 simultaneously arriving and departing customers). This conflict is resolved by the confluent transition function.

The clerk handles simultaneous arrivals and departures by first handling the departures and then the arrivals. To do this, the confluent transition function calls the internal transition function first (to remove the departed customer from the list) and then the external transition function (to add new customers to the end of the list and begin ringing up the first customer). The confluent transition function is shown below.

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

| 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 times needed to process the customer orders.

To see how this model behaves, suppose customers arrive according to the schedule shown in Table 3.1. The first customer appears on the clerk's "arrive" port at time 1, the next customer 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 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 activated when the simulation clock has reached the model's time of next event. The internal transition function describes the autonomous behavior of the model (i.e., how the model responds to events that it has scheduled for itself). Internal transitions are scheduled with the time advance function.

A call to the internal transition function is always immediately preceded by a call to the output function. Consequently, a model produces output by scheduling events for itself. The value of the output is computed using by the output function using the model's current state.

To complete our simulation of the convenience store, we need two other **Atomic** models. The first model creates customers for the **Clerk** to serve. The rate at which customers arrive could be modeled using a random variable or it with a table such as the one used in the example above. In either case, we hope that the model of the customer arrival process accurately reflects what happens in a typical day at the convenience store. Data used in the table for this example 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 create an **Atomic** model called a **Generator** to create customers. This model is driven by a table formatted like Table 3.1. The input file contains a line for each customer. Each line has the customer's time of arrival followed by the customer's time for service. The **Generator** does not need to process input events because all of its activities are scripted in the input file. The **Generator** has a single output port "arrive" through which it exports arriving customers. The model state is the list of **Customer**s yet to 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 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 constructor 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 does not respond to input events, its external transition function is empty. Similarly, the confluent transition function merely calls the internal transition function (though, in fact, it could be empty because the confluent transition will never be called).

```
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 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, 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' inter-arrival times, not the absolute arrival times, and the time advance function simply returns this value.

```
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**'s output must be connected to the **Clerk**'s input. When these are connected, output from the **Generator**'s "arrive" port becomes input on the **Clerk**'s "arrive" port. These inputs cause the **Clerk**'s external transition function to be activated. The relationship between input and output events can be understood by viewing the whole model as a block diagram with 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 3.2.

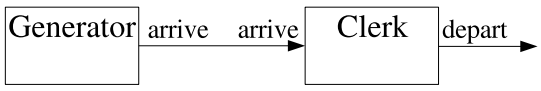


Figure 3.2: A block-diagram view of **Generator** and **Clerk** models.

The **Clerk**, **Generator**, and their interconnections constitute a coupled (or network) model. The coupled model depicted in Figure 3.2 is realized with a **Digraph** that has the **Generator** and **Clerk** as components. 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 connected by coupling the "arrive" output port of the **Generator** to the "arrive" input port of the **Clerk**.

Having created a coupled model to represent the store, all that remains is to perform the simulation. Here is the code snippet that simulates the 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 generated the execution traces shown in the example above.

```
#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 objects that we create 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 (in the example above, immediately prior to the return of the method execNextEvent()). The argument to the garbage collection method is a bag of objects created as output in the current simulation cycle.

In our example, the **Clerk** and **Generator** models use their garbage collection method to 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 question 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 customers are processed. This approach works but has the unfortunate effect of cluttering up the **Clerk** with code specific to our experiment.

A better approach is to have an **Observer** that is coupled to the **Clerk**'s "depart" port. The **Observer** records the desired statistics as it receives **Customer** objects on its "depart" input port. The advantage of this approach is that we can create new types of clerks to perform the same experiment, using, for example, different queuing strategies, without changing the experimental setup (i.e., customer generation and data collection). Similarly, we can change the experiment (i.e., how customers are generated and what data is collected) without changing the clerk.

Below is the code for the **Observer** class. This model is driven solely by external events. The observer reacts to an external event by recording the time that the **Customer** departed the **Clerk**'s queue (i.e., the current simulation time) and how long 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 3.3.


Figure 3.3: The **Generator**, **Clerk**, and **Observer** model.

Given the customer arrival data in Table 3.1, the consequent customer departure and waiting times are

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

Table 3.2: Customer departure times and waiting times.

shown in Table 3.2. With this output, we can use a spreadsheet to find the maximum and average times that the customers spent waiting in line.

Again notice that the customer departure times correspond exactly with the production of customer departure events by the **Clerk** model. Each entry in Table 3.2 is the result of executing the **Observer**'s external transition function. Also notice that the **Observer**'s internal and confluent transition functions are never executed because the **Observer**'s time advance method always returns infinity.

This section has demonstrated the most common parts of a simulation program 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 4

# 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 virtual methods that correspond to the internal, external, and confluent transition functions, the output function, and the time advance function.

The state of an atomic model is realized by the attributes of the object that implements the model. The internal transition function describes the model's autonomous behavior; that is, how its state evolves in the absence of input. These types of events are called internal events because they are self-induced; i.e., internal to the model. The time advance function schedules these autonomous changes of state. The output function gives the model's output when these internal events occur.

The external transition function describes how the model changes state in response to input. The confluent transition function handles the simultaneous occurrence of an internal and external event. The types of objects that are accepted as input and produced as output are specified with a template argument to the **Atomic** base class.

The **Clerk** described in Section 3 demonstrates all the aspects of an **Atomic** model. We'll use it here to demonstrate how an **Atomic** model generates output, processes input, and schedules internal events. Below 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;
    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 the simulation of the convenience store described in Section 3 (i.e., with the arrivals listed in Table 3.1). The arrival data is listed again here:

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

Table 4.1: Customer arrival times and times to process customers' orders.

Table 4.1 describes an input sequence that is input to the **Clerk** model. The algorithm for processing this, or any other, input sequence is listed below. The **Atomic** model being simulated is called 'model', t is the current simulation time (i.e., the time of the last event - internal, external, or confluent), and t_input is the time of the next 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 then compute the next state by calling model.delta_conf(x) where x contains the input 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 at time t_input.

5. Set t equal to tN.

6. Repeat if there are more input or internal events to process.

The first step of this algorithm computes the time of the next event as the sooner of the next input event and the next internal event. If the next internal 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 is produced in this case. The elapsed time argument given to the external transition function is the amount of time that has passed since the previous event - internal, external, or confluent - at that model.

If the next input and internal event happen at the same time, then the model produces an output and its next state is computed with the confluent transition function. The simulation clock is then advanced to the event time. These steps are repeated until there are no internal or external events remaining to process.

The output trace resulting from the input sequence in Table 4.1 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 lets create a more sophisticated clerk. This clerk interrupts the checkout of a customer with a large order to more quickly serve a customer with a small order. The clerk, however, does this only occasionally. To be precise, let a small order be one requiring no more than one unit of time to process. Moreover, the clerk interrupts the processing of an order at most once in every 10 units of time.

This new clerk has two state variables. The first records the time remaining before the clerk is willing to interrupt the processing of a customer. The second is the list of customers waiting to be served. Here is the header file for the new clerk model, which is called **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 interruption timer to zero.

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

The output function of this model sets the exit time of the departing customer and then ejects that customer via the "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 external transition function works as follows. When a new customer arrives, the clerk first advanced its clock by the elapsed time. Next, she reduces the time remaining to process the current customer. This reduction reflects the amount of time that has already been spent on the customer's order, which is the time elapsed since the clerk's last change of state. Then the clerk decrements the time remaining before she is willing to interrupt the processing of a large order. This timer is also decremented by the elapsed time.

Now the clerk 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 clerk is willing to interrupt the present order, then that customer with the small order goes to the front of the line. This preempts the current customer, who now has the second place in line, and causes the preempt timer 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 begins by decrementing the time remaining before the clerk will interrupt an order. The customer that just departed the store via the output function is then removed from the front of the line. If the line is empty, then there is nothing to do and the clerk sits idly behind her counter. If the line is not empty and the preemption 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. Then 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 at the front of the line, or infinity (i.e., 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** in section 3:

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

The behavior of the **Clerk2** model is more complex than that of the **Clerk** model. To exercise the **Clerk2**, we replace the **Clerk** model in the example from section 3 with the **Clerk2** model and perform the same experiment. Here is the output trace for the **Clerk2** model in response to the input sequence shown in Table 4.1. 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. 4.1. 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.

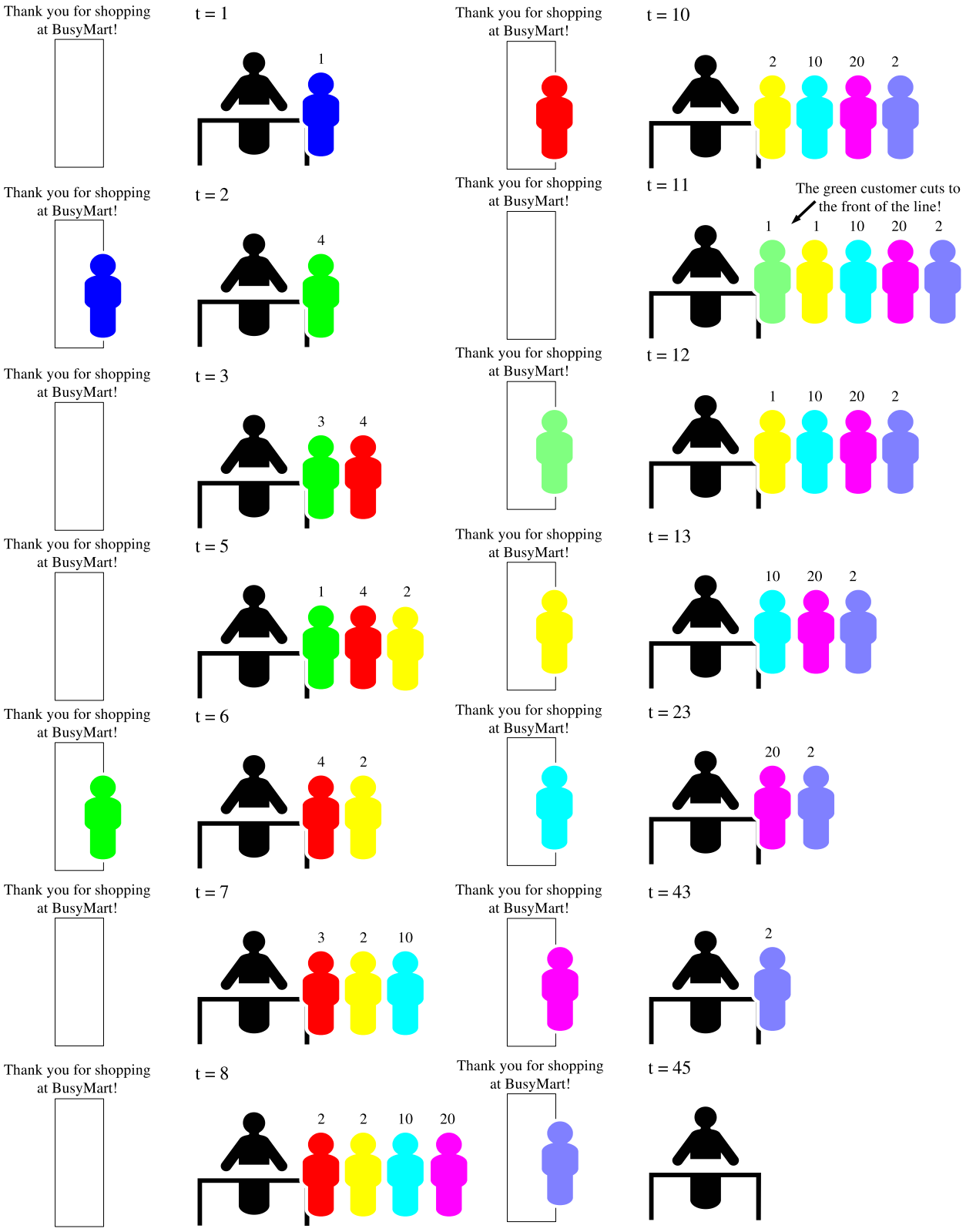


Figure 4.1: The evolution of the **Clerk2** line in response to the customer arrival sequence listed in Table 4.1.

# Chapter 5

# Network Models

A network model comprises atomic models and other network models that are interconnected. Network models can be components of other network models, thereby enabling the construction of multi-level systems. 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 fundamental behaviors; network models define structure.

## 5.1 Parts of a Network Model

Network models are derived from the **Network** class. This class has two virtual methods: ***route*** and ***getComponents***. The ***route*** method implements connections between the components of the network and between these components and the inputs and outputs of the network itself. The ***getComponents*** method provides the set of components that constitute the network.

### 5.1.1 The *route* method

The ***route*** method realizes three types of connections. The first are connections between components of the network. The second are connections from the network's inputs to the inputs of its component models. The third are connections from the component outputs to the outputs of the network. The signature of the ***route*** method is

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

The value argument is the object to route, the model argument is the **Network** or **Atomic** model that is the source of the value object, and the r argument is a bag to be filled with models that should receive the value object as input. Each target is described by an **Event** object that carries two pieces of information: a pointer to the model that is the target and the object to be delivered to that target. The simulator uses the **Event** objects in one of three ways depending on the relationship between the source of the object and its target. These uses are

1. If the source is a component of the network and the target is the network itself, then the value becomes an output from the network.

2. If the source is the network and the target is a component of the network, then the value becomes an input to that component.

3. If the source and target are both components of the network, then the value becomes an input to the target.

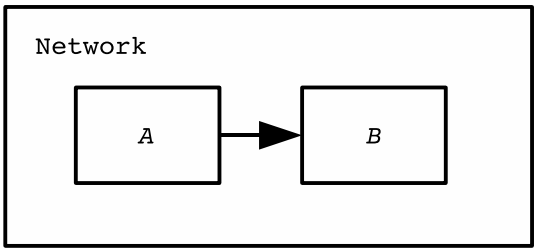


Figure 5.1: Two connected **Atomic** components in a single **Network**.

Any other relationship between the source and the target is illegal and causes the simulator to raise an exception.

The simplest example of the ***route*** method converts output from one **Atomic** component into input for another **Atomic** component. Figure 5.1 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, calling the **Network**'s ***route*** method for each one. The arguments passed to ***route*** at each call are

1. the output object itself, which is the value argument,
2. a pointer to $A$, which is the model argument, and
3. an empty **Bag** for holding **Event** objects.

Two things are done by the ***route*** method to cause **Atomic** model $B$ to receive the output object from $A$. First, an **Event** object is created that contains the output object and a pointer to $B$. Second, this **Event** object is 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 for this example 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 models. This route method implements the network shown in Fig. 5.1.

A more complicated example is the network receiving input destined for one of its atomic components. This can happen, for instance, when the network is a component of another network. Suppose the input to the network is to become input for **Atomic** model $A$. Figure 5.2 extends Fig. 5.1 to include this connection.

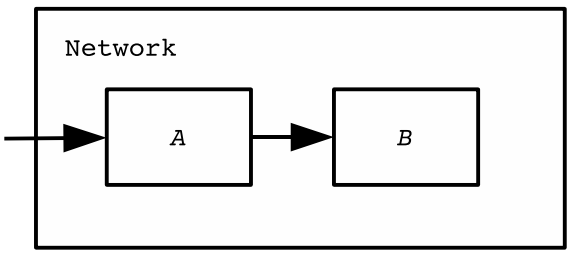


Figure 5.2: Two connected **Atomic** components with external input coupling to component $A$.

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, which is the value argument,
2. a pointer to the **Network** that is receiving the input, and
3. an empty **Bag** for holding **Event** objects.

As before, ***route*** creates an **Event** object that indicates the target model and value of the input. This **Event** object is put into the **Bag** r of receivers. The code below implements the network shown in Fig. 5.2; note that 'this' points to the **Network** itself (i.e., to the network that is receiving the initial input).

```
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);
   }
}
```

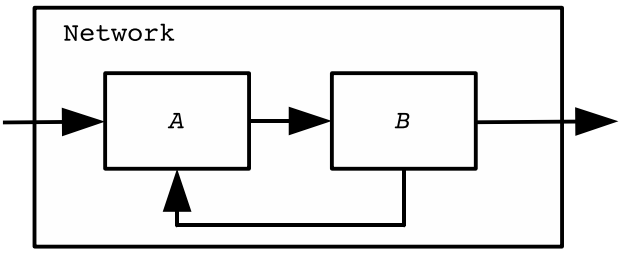


Figure 5.3: A network with external input, external output, and internal coupling.

For a complete example, the network is extended 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. 5.3. These new connections require an additional case in the route method. This case checks for output from $B$ and, if such an output is found, directs it to both $A$ and the network. An **Event** object is created for each target and added to the **Bag** r of receivers: one of these **Event**s results in an input to $A$ and the other in an output from the network. 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);
    }
}
```

Though not demonstrated above, the ***route*** method is allowed to modify the value object before sending it to a target. This can be useful in some instances.

### 5.1.2 The *getComponents* method

The ***getComponents*** method is the other virtual method that must be implemented by any class that is derived from **Network**. The simulator passes to this method an empty **Set** of pointers to models, and this set must be filled with the network's components. The signature of the getComponents method is

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

where c is the set to be filled. The code below shows how this method is implemented for the two component network shown in Fig. 5.3. This code, of course, also works for the networks shown in Figs. 5.2 and 5.1.

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

### 5.1.3 Illegal networks

There are two rules that must be followed when building networks. First, components cannot be connected to themselves. This means that direct feedback loops and connections directly through a network model are illegal. The former can always be replaced with an internal event and the latter by simply bypassing the network. These two cases are illustrated in Fig. 5.4. Second, direct coupling can only occur between components belonging to the same network, and every component must belong to at most one network.

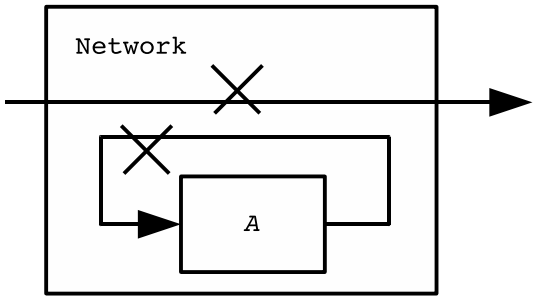


Figure 5.4: Illegal coupling in a **Network** model.

## 5.2 Simulating a Network Model

There are four steps in each iteration of the simulation algorithm. These are

1. Advance the simulation clock to the time of the next event.
2. Compute the outputs from atomic models that will change state (i.e., that will undergo an internal or confluent event) and convert these outputs into inputs for other models.

3. Calculate the next state of each model with events - internal, external, or both to process.

4. Cleanup garbage left over from the output calculations.

These four steps are repeated until the time of the next event is at infinity (i.e., DBL_MAX) or you decide to stop the simulation.

There are no special rules for simulating hierarchical models. The simulator considers the entire collection of atomic models when determining the next event time, output from atomic models are recursively routed to their atomic destinations, and the 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 flattens (indirectly, via its recursive routing of events) multi-level networks during simulation.

Algorithm 1 sketches the simulation procedure. Note that the procedure for simulating atomic models (see section 4) is embedded in the procedure for simulating network models. The rules for atomic models do not change: 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.

**Algorithm 1** The simulation procedure for a network model.

```
Initialize the state of every Atomic model
Set the time of last event t_{l,i} of every Atomic model i to 0
Set the simulation time t to 0
Set the time of next event for model i to t_{l,i} + ta_i()
while The smallest time of next event for the Atomic models is less than DBL_MAX do
  Set t to the smallest time of next event for the Atomic models
  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 output from imminent models to input for other models using the Networks' route methods
  (do this recursively if the network 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 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 input
    end if
    Set t_{l,i} to t
    Set the time of next event for model i to t_{l,i} + ta_i()
  end for
end while
```

## 5.3 A complete example of a network model

I'll use the **SimpleDigraph** class to illustrate how to build a network model. 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 from $A$ becomes input to $B$.

The **SimpleDigraph** has two methods for building a network. The ***add*** method takes an **Atomic** or **Network** model and adds it to the set of components. The ***couple*** method accepts a pair of components and connects the first to the second. Below is the class definition for the model. Note that it has a template parameter for setting its input and output type.

```
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. The first is a set of pointers to the components of the network. These are stored in the **Set** called models. The components can be **Atomic** objects, **Network** objects, or both. These components of the **SimpleDigraph** are the nodes of its directed graph. The second member variable is the network's links. These are stored in the **map** called graph.

The **SimpleDigraph** has four methods plus the required ***route*** and ***getComponents***. One of these is the constructor, which creates an empty network. Another is the destructor, which deletes all of the network's components. The remaining two are ***add*** and ***couple***.

The ***add*** method does three things. First, it checks that the network is not being added to itself. This is illegal and will cause the simulator to throw an exception. Next, it adds the new component to its set of components. Last, the **SimpleDigraph** makes itself the component's parent. This needed so that the simulator can climb up and down the model tree. If this step is omitted then the recursive routing of events will 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. First, it adds the source (src) and destination (dst) models to the set of components. We could simply have required that the user call the ***add*** method before calling the ***couple*** method, but adding the components here doesn't hurt and might prevent an error. Second, ***couple*** adds the src $\rightarrow$ dst link to the graph. Notice that the ***SimpleDigraph*** itself is a node in the network, but it is not in the set of components!. Components that are connected to the network cause outputs from the network. Similarly, connecting the network to a component causes input to the network to become input to the component. Here is the implementation of the ***couple*** method.

```
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 ***route*** are an object to be routed, the network element (i.e., either the **SimpleDigraph** or one of its components) that created that object, and the **Bag** to be filled with **Event** objects that indicate the object's receivers. The method begins by finding the collection of components that are connected to the source of the object. Next, it iterates through this collection and for each receiver adds an **Event** to the **Bag** of receivers. 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 a **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 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 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;
   }
}
```

## 5.4 Digraph Models

This section introduces the **Digraph** model, which is part of the Adevs simulation library. The **Digraph** is a tool for building networks described by a block diagram. The model of the convenience store, developed in section 3, was 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. 5.5.

```
// 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);
```

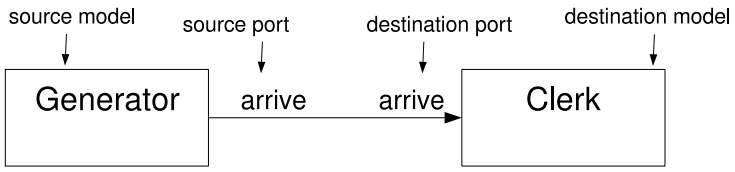


Figure 5.5: A **Digraph** model with two components.

The components of a **Digraph** must use **adevs::PortValue** objects for their input and output type. The **Digraph** is a template class with two template parameters. The first parameter is the type of object used for a value in its **PortValue** objects. The second parameter is the type of object used for a port in its **PortValue** objects. The port parameter is of type 'int' by default.

The **Digraph** has two methods that are used to construct a network. The ***add*** method adds a component to the network. The argument to the add method is the model to be included in the network. The ***couple*** method connects components of the network. The first two arguments to the ***couple*** method are the source model and port. The second two arguments are the destination model and port.

The effect of coupling a source model to a destination model is that output produced by the source model on the source port appears as input to the destination model on the destination port. To illustrate this, consider the output function of the **Generator** model shown in Fig. 5.5.

```
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 output function places a value of type 'Customer*' on the "arrive" port of the **Generator**; recall that 'IO_Type' is a typedef for 'PortValue¡Customer*¿'. A corresponding **PortValue** object appears in the input bag of the **Clerk**. The value attribute of the **PortValue** object received by the clerk points to the **Customer** object created by the **Generator**. The port attribute of the **PortValue** object is the **Clerk**'s "arrive" port.

The components for the network need not consist only of **Atomic** models; the **Digraph** can also have other **Network** models as its components. For instance, suppose we want to model a convenience store

that has two clerks. When customers are ready to pay their bill, they enter the shortest line. To build this model, we reuse the **Clerk**, **Generator** and **Observer** models introduced in section 3. We add a model called **Decision** of how customers select a line.

The code for the **Decision** model is shown below. This model has two output ports, one for each line, and there are three input ports. One input port accepts new customers. The others are used to track the number of customers departing each line: a customer departing either clerk generates an event on the appropriate input port. In this way, the model is able to track the number of customers in each line and assign new customers to the shortest one. 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 of the store and its multiple clerks is shown in Fig. 5.6. The external interface for this block diagram is identical to that of the clerks. It has the same inputs and outputs as the **Clerk** and **Clerk2** models, and we can therefore 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 producing customers for a distinct clerk. These output ports are coupled to the "arrive" port of the appropriate clerk. The **Clerk**'s "depart" output ports are coupled to the appropriate "departures" port of the decision model.

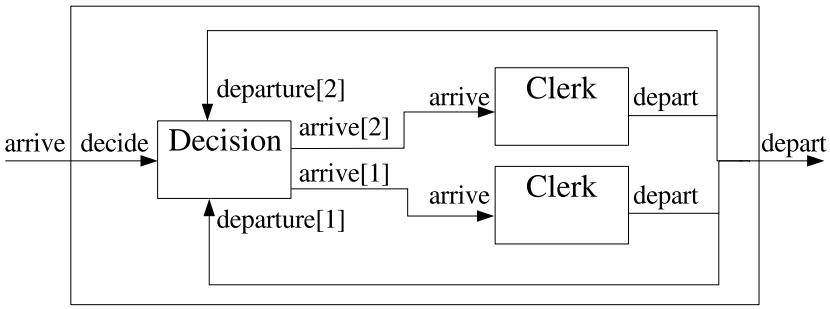


Figure 5.6: Component models and their interconnections in the multi-clerk convenience store model.

The multi-clerk model is implemented by deriving a new class from **Digraph**. The constructor of this new class creates and adds the component models and establishes their interconnections. Here is the header file for this 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 its component models. This is because the components are adopted by the base class when they are added using the **Digraph**'s ***add*** method. Consequently, the component models are deleted by the base class destructor, rather than the destructor of the derived class.

## 5.5 Cell Space Models

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

1. (Death rule). If a cell is alive and it has less than two or more than three living neighbors then the cell dies.

2. (Rebirth rule). 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: atomic models that implement the individual cells and a **CellSpace** that contains the cells. The **CellSpace** is a type of **Network**, and its components exchange **CellEvent** objects that have four attributes: the x, y, and z coordinates of the target cell (the cell space can have three dimensions; the Game of Life uses just two) and the object to deliver to that target. The **CellEvent** class is a template class whose template argument sets the type of object that the event delivers. The size of the **CellSpace** is determined when the **CellSpace** object is created, and it has methods for adding and retrieving cells by location.

The **Atomic** cells in our Game of Life have two state variables: the dead or alive status of the cell and its count 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 output of the cell is its new dead or alive state. In order to produce properly targeted **CellEvent**s, each cell knows its own location in the cell space. The internal transition function causes the cell to change its dead/alive state. The external transition function updates the cell's count of living neighbors as those neighbors change their dead/alive state. Here is 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 attribute has the type **Phase**. The ***check_born_rule*** method tests the rebirth condition and the ***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 (i.e., if the cell is dead then check the rebirth rule; if alive, check the death rule). The number of living cells is updated by the cell's ***delta_ext*** method when neighboring cells report a change in their state. 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 nested for loops create a **CellEvent** targeted at each adjacent cell. The location of the targeted cell is written to the x, y, and z attributes of the **CellEvent** object. Just like arrays, the locations range from zero to the cell space's size minus one. The **CellSpace** routes the **CellEvent** objects to their targets. However, 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 simulation programs that we've seen. A **CellSpace** object is created and we add cells to it. Then a **Simulator** object is created 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** objects used as input and output. The **CellSpace** constructor sets the dimensions of the space. Every **CellSpace** is three dimensional, and the constructor accepts three arguments for its x, y, and z dimensions. Omitted arguments default to 1. The signature of the constructor 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, and 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;
```

Also like the **Digraph**, the **CellSpace** deletes its components when it is deleted.

The **CellSpace** has five methods for retrieving cells and the dimensions 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 distinct patterns. Some of these patterns are fixed and unchanging. Others oscillate, cycling through a set of patterns that always repeats itself. Still others seem to crawl or fly. One common pattern is the Block, which is shown in Fig. 5.7. 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 states are constant and so are their neighbor counts.

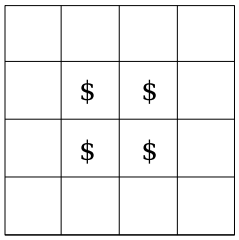


Figure 5.7: The Block.

The Blinker in Fig. 5.8 is more interesting. This oscillating pattern has just two stages: a vertical and a horizontal. Table 5.1 shows the input, output, and state transitions that are computed for the cell marked with a * in Fig. 5.8. Just like the pattern it is a part of, the cells oscillates between two different states.

The confluent transition function plays an important role in the Blinker. All but the first row in Table 5.1 has simultaneous input and output, which means that an internal and external event coincide. Consequently, the next state of the cell is determined by its ***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.

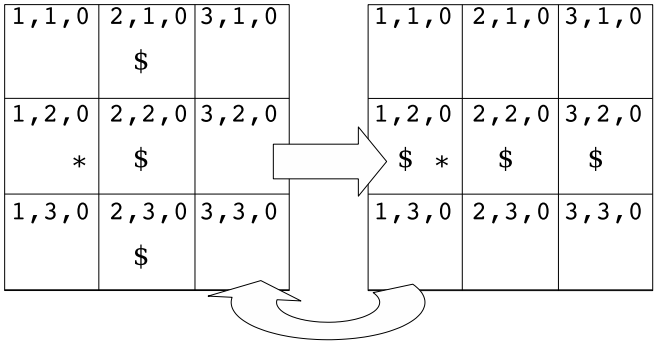


Figure 5.8: The Blinker. The input, output, and state transitions for the cell marked with a * are shown in Table 5.1. The address of each cell is shown in its upper left corner. Living cells are indicated with a $.

| 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 5.1: State, input, and output trajectory for the cell marked with * in Fig. 5.8.

# Chapter 6

# Variable Structure Models

The composition of a variable structure model changes over time. New components are added as, for example, machinery is installed in a factory, organisms reproduce, or shells are fired from a cannon. Existing components are removed as machines break, organisms die, or shells in flight find their targets. Components are rearranged as, for example, parts move through a manufacturing process, organisms migrate, or a command and control network loses communication lines.

For modeling systems with a variable structure, Adevs provides a simple but effective mechanism to coordinate changes in structure and changes in state. This mechanism is 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.

## 6.1 Building and Simulating Variable Structure Models

Every **Network** and **Atomic** model has a virtual 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 (that is, after computing the models' new states but prior to the garbage collection step) the simulator invokes the ***model_transition*** method of every **Atomic** model that changed state in that cycle. When the ***model_transition*** method is invoked, the **Atomic** model can do anything except alter the set of components of a **Network** model.

If a model's ***model_transition*** method returns true, then the simulator calls the ***model_transition*** method of that model's parent. The parent is, of course, a **Network** model, and its ***model_transition*** method may add, remove, and rearrange the network's components. But it must not delete any components! The simulator will automatically delete components that are removed from the model when the structure change calculations are finished.

As before, if the **Network**'s ***model_transition*** method returns true then the simulator invokes the ***model_transition*** method of its parent. Note, however, that the ***model_transition*** method of any model is invoked at most once in each simulation cycle. This invocation, if it occurs, takes place after every component of the network qualifying for the evaluation of its ***model_transition*** method has computed its change of structure.

After invoking every eligible model's ***model_transition*** method, the simulator performs a somewhat complicated cleanup process. During this process the simulator constructs two sets. The first set contains 1) the components that belonged to all of the **Network** models whose ***model_transition*** method was invoked

and 2) all of the components of every model in this set (i.e., this set is constructed recursively: if any model is in the set, so are its component models). The second set is defined in the same way, but it is computed using sets of components as they are 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.

The procedure for calculating a change of structure can be summarized as follows:

1. Calculate the ***model_transition*** method of every atomic model that changed state.
2. Construct the set of network models that contain an atomic model from step 1 whose ***model_transition*** method returned true. These network models are sort by their depth in the tree of models with the bottom-most first and top-most last. This ensures that structure changes are calculated from the bottom up.
3. Calculate the ***model_transition*** methods of the networks in order. On completing each transition, do the following:
   (a) Remove the network from the list.
   (b) If the network's ***model_transition*** method returns true, put the parent of the network into the sorted list of networks from step 2. This ensures that a network's ***model_transition*** method is invoked only after all of its eligible components have had their ***model_transition*** method invoked.
4. When there are no more networks in the list, do the following:
   (a) Delete the components removed from the model (i.e., the models without a parent).
   (b) Initialize the components that were added to the model.

The procedure for calculating a change of structure is illustrated in Fig. 6.1. The black models' ***model_transition*** methods returned true. The set of components examined before and after the structure change are listed above the before (left) and after (right) trees. Notice that these models are in the sub-tree below the model $C$, which is the top-most model in that sub-tree that returned false from its ***model_transition*** method. Also note that while the leaves of the tree may have had their model transition method invoked, none returns true and so their parents' model transition methods are not invoked nor are their sets of components considered when determining what models have been added and removed from the model. 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 usually observed when building **Network** models. Any **Network** model can, in principle, modify the set of components of any other model, regardless of proximity or hierarchy. The potential for anarchy is great, and so the design of a variable structure model should be considered carefully. There are two approaches to such a design 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. The **Atomic** model's ***model_transition*** method returns true causing its parent's ***model_transition*** method to be invoked. The parent **Network** model then retrieves and acts on the requests posted by its components. The **Network** repeats this process if it wants to effect structure changes involving models other than its immediate children; i.e., it posts a request for its parent and returns true from its ***model_transition*** method.

The second approach allows arbitrary changes in structure by forcing the model at the 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 change of structure

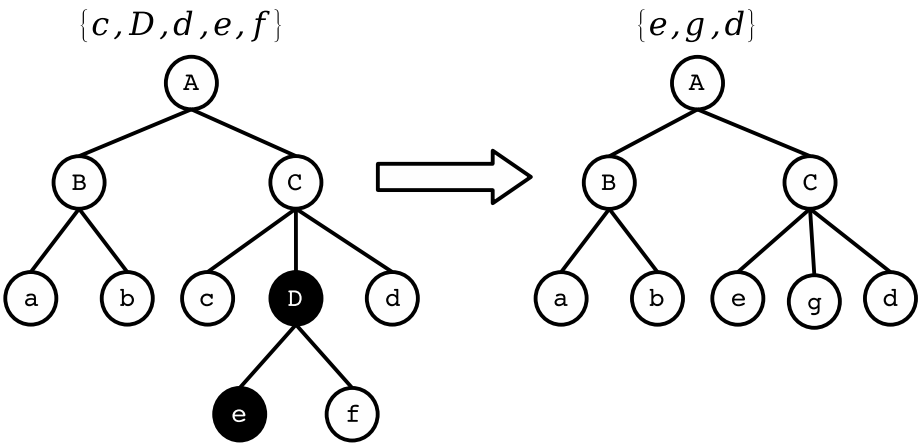


Figure 6.1: Illustration of a change of structure in a variable structure model.

uses its ***model_transition*** method to post a request for its parent. This request 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. The scope of change must, however, be carefully restricted. With the second approach, every structure change requires the simulator to include every part of the model in its calculations, regardless of the actual extent of the change in structure. In this case, however, the scope of a structure change may be unlimited.

## 6.2 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 to change the company name to The Custom Widget and Gizmo Company). The machines for the factory are expensive to operate. To keep costs down, the factory will operate just enough machinery to fill orders for gizmos. 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 answers to two questions: how many machines are needed and how much will it costs to operate them.

We will use a variable structure model to answer these questions. This model has three components: a generator that creates orders for gizmos, a model of a machine, and a model of the factory that contains the machines and that activates and deactivates machines as required to satisfy demand. The model of the factory is illustrated in Fig. 6.2.

The generator creates new orders for the factory. Each order is identified with an integer label, and the generator produces orders at the rate anticipated by the factory engineers. Demand at the factory is expected to be steady with a new order arriving every 1/2 to 2 days. This expected demand is modeled with a random variable that is uniformly distributed in [0.5,2]. Here is the code for the generator:

```
#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.
```

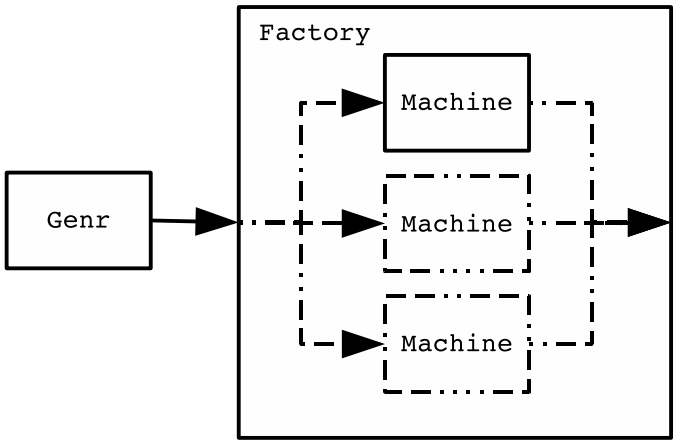


Figure 6.2: Block diagram of the model of the factory. The broken lines indicate structural elements that are subject to change.

```
         */
        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 model of a machine is similar to the **Clerk** in section 3. A machine requires 3 days to make a gizmo, and orders for gizmos are processed first come, first serve. The **Machine**'s ***model_transition*** method is inherited from its **Atomic** base class. I'll discuss the role of the ***model_transition*** method after introducing the **Factory** class. Here is the code for the **Machine**.

```
#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 contained in the **Factory** model at any time is determined by the current demand for gizmos. The real factory, of course, will have a fixed number of machines on the factory floor, but the planners do not 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.

The Custom Widget and Gizmo Company has built its reputation on a guaranteed time of service, 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 at any time. However, 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 time. To accomplish this goal, the factory's 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. If a **Machine** becomes idle or if its queue is full (i.e., the machine is working on one order and has another order waiting in its queue), then that machine's ***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. Then it 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 simulator's structure calculations are done only when the **Factory**'s ***model_transition*** method is invoked, and these calculations are therefore limited to instants when **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** models must watch their own status and inform the **Factory** when there is a potential shortage of capacity.

If we had used the second approached to build our variable structure model, then the **Machine**s' ***model_transition*** method could have simply returned true: no need for a status check. The **Factory** would iterate through its list of **Machine**s, adding and deleting **Machine**s as needed. This is more computationally expensive: the simulator looks for changes in the **Factory**'s set of components at each simulation cycle. However, the design of the model is simpler, albeit only marginally so in this instance.

The **Factory** is a **Network** model and must implement all of the **Network**'s virtual methods: ***route***, ***getComponents***, and ***model_transition***. The ***route*** method is responsible for assigning orders to machines. When an order arrives, it is sent to the machine that will most quickly satisfy the order. The ***getComponents*** method puts the current set of machines into the **Set** c of components. The ***model_transition*** method examines the status of each machine, deleting idle machines and adding new machines if they are needed to maintain reserve capacity. The **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 works, let us 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, on 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, and the machine begins processing the order. Next the **Machine**'s ***model_transition*** method is invoked. It discovers that the machine is working and has space in its queue, and so the ***model_transition*** method 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 sent to that **Machine**. The **Machine**'s ***delta_ext*** method is invoked and it queues the order. The **Machine**'s ***model_transition*** method is invoked next, and because the queue is full 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 is overloaded, and creates a new **Machine**.

At this time, the working **Machine** needs two more days to finish the first order, and it will not complete its second order until a total of five days have elapsed.

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 produces the first completed order (i.e., the order begun on day zero). Next the **Factory**'s ***route*** method is called twice. The first call converts the **Machine**'s output into an output from the **Factory**. The second call routes the new order to the idle **Machine**.

Now 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. Lastly, the ***model_transition*** methods of both **Machine**s are invoked. Both **Machine**'s have room in their queue and so both return false.

Suppose no orders arrive in the next three days. At day six, both machines finish their work. The **Machine**s' ***output_func*** methods are invoked, producing the finished orders. These become output from the **Factory** via the **Factory**'s ***route*** method.

Next, the **Machine**s' ***delta_int*** methods are called and both **Machine**s become idle. After this, the **Machine**s' ***model_transition*** methods are invoked and these return true because the machines are idle. This causes the **Factory**'s ***model_transition*** method to be called. It examines each **Machine**, sees that they are idle, and deletes both of them. Lastly the **Factory** computes its available capacity, which is now zero, and creates a new machine. This returns the **Factory** to its initial configuration with 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 count of active machines 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 the one year time integral of the active machine count. A plot of the active machine count versus time is shown in Fig. 6.3. The maximum count of active machines is four and the annual operating cost is $944 (this plot is from the first simulation run listed in Table 6.1).

Because new orders arrive at random, the annual operating cost and maximum machine count are themselves random numbers. Consequently, data from several simulation runs are needed to make an informed decision. Table 6.1 shows the outcomes of ten simulation runs. Each uses a different sequence of random numbers and therefore produces a different result (i.e., another sample of the maximum active machine count and annual operating cost). From this data, the factory engineers conclude that 4 machines are required and the average annual operating cost will be $961.

| Random number 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 6.1: Outcomes of ten factory simulations.

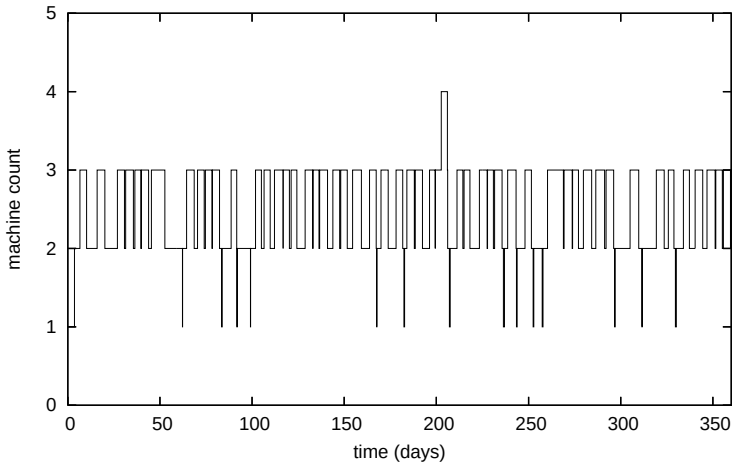


Figure 6.3: Active machine count over one year.

# Chapter 7

# Continuous Models

Many engineered systems comprise computers, communications networks, and other digital systems to monitor and control physical (electrical, mechanical, thermodynamic, etc.) processes. Models of these systems have some parts modeled as discrete event systems, other parts modeled with continuous (differential or differential-algebraic) equations, and the interaction of these parts is crucial to understanding the system's behavior.

The interaction of continuous and discrete event models is necessarily discrete. For example, a digital thermometer reports temperature in discrete increments, electrical switches are either open or closed, a threshold sensor is either tripped or it is not. Discrete interactions in a combined continuous-discrete event simulation are managed just as before: the models interact by producing output events and reacting to input events.

If, on the other hand, two systems interact continuously, then those interacting parts are modeled with continuous equations. In this case, accurate simulation is 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. The essence of this approach to combined simulation in Adevs consists therefore of building atomic models that i) approximate the behavior of the continuous systems and ii) generate and consume events at the instants when the continuous system interacts with a discrete event one.

There are three possibly outcomes of this lumping process. 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 very simple, yielding to analytical solutions that are easily transformed into discrete event models. Between these extremes are models with continuous dynamics that are not simple but do not dominate the modeling problem. The continuous system simulation part of Adevs is aimed at this type of model.

## 7.1 Differential equation modeling with the ode_system class

Models described by ordinary differential equations are implemented by sub-classing the **ode_system** class. This class has two sets of methods: the first is for the model's continuous dynamics and the second is for the model's discrete event dynamics. I'll illustrate both with the simple, if somewhat contrived, example of a cherry bomb[1]. This bomb is dropped from a height of 1 meter and bounces until it explodes or is doused with water. We'll assume that the cherry bomb only bounces up and down and is perfectly elastic. The cherry bomb explodes 2 seconds from the time it is lit unless doused first. Dousing the cherry bomb puts out the fuse[2]. Dousing is a discrete input event and the cherry bomb produces a discrete output event if it explodes.

[1] A cherry bomb is a small red firecracker. They are dangerous and illegal in the United States. Nonetheless, every school seems to have at least one obnoxious kid who likes to put them into the toilets.

[2] Cherry bomb fuses are frequently water proofed.

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

$$\dot{v} = -9.8 \tag{7.1}$$

$$\dot{h} = v \tag{7.2}$$

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

$$\dot{t} = 1 \tag{7.3}$$

whose solution is $t_0 + t$ or just $t$ if we set $t_0 = 0$. The ball bounces when it hits the floor, and bouncing causes the ball's velocity to change sign. Specifically

$$h = 0 \ \& \ v < 0 \implies v \leftarrow -v \tag{7.4}$$

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

Equations 7.1, 7.2, and 7.3 are the state variable derivatives, and these equations are implemented in the ***der_func*** method of the **ode_system** class. The signature for this method is

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

The q pointer is the array of state variable values: $h$, $v$, and $t$. The dq pointer is the array of state variable derivatives: $\dot{h}$, $\dot{v}$, and $\dot{t}$. When the simulator calls ***der_func***, it supplies q. In response, the method computes the values of $\dot{h}$, $\dot{q}$, and $\dot{t}$ and stores them in the dq array.

Equation 7.4 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. The signature of this method is

```
void state_event_func(const double *q, double *z)
```

Again, the supplied q array contains the current state variable values. These are used 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. Note that the event condition should be continuous in the state variables on which it depends. In the case of the cherry bomb this is simple to do. We simply use $z = h$ if $v < 0$ and $z = 1$ if $v >= 0$.

The 'then' part (right hand side) is implemented with the ***internal_event*** method, which the simulator invokes when the state event condition is true. The signature of this method is

```
void internal_event(double *q, const bool *state_event)
```

where the q array contains the value of the state variables at the event. The entries of the array state_event are true for each z in the state event condition array that evaluates to zero. This array therefore has one entry for each state event condition, and it has one additional entry to indicate time events, which are described below.

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. The ***time_event_func*** method is used to schedule the explosion by returning the time remaining until the fuse burns out. The signature of the of this method is

```
double time_event_func(const double* q)
```

As before, the q array has the current value of the state variables. The ***time_event_func*** is similar to the ***ta*** method. It is used to schedule autonomous events based on the current value of the model's state variables. When this time expires, the simulator calls the ***internal_event*** method with the last flag in the state event array set to true.

The second event that can change the state of the fuse is dousing with water. This an external event. External events occur when input arrives at the model. The ***external_event*** method implements the response of the cherry bomb to dousing with water. Its signature is

```
void external_event(double *q, double e, const Bag<X> &xb)
```

The array q contains the values of the continuous state variables, e is the time since the last discrete event, and xb is the bag of input. The douse event is an input and it appears in the input bag xb when and if the event occurs.

As before, it is possible for an external and internal event to coincide. When this happens, the simulator calls the method ***confluent_event***. Its signature is

```
void confluent_event (double *q, const bool *state_event, const Bag<X> &xb)
```

and its arguments are as described for the internal and external methods.

The cherry bomb produces an output event when it explodes, and the ***output_func*** method is used for this purpose. Its signature is

```
void output_func(const double *q, const bool *state_event, Bag<X> &yb)
```

The q and state_event arguments are as described for the ***internal_event*** method, and the bag yb is to be filled with the model's output. As with an **Atomic** model, the ***output_func*** is always invoked immediately prior to the ***internal_event*** and ***confluent_event*** methods.

All that remains in the implementation is the ***gc_output*** for collecting garbage, a constructor, and a method for initializing the continuous state variables. The ***gc_output*** method works identically to that of the **Atomic** class. The constructor for the cherry bomb must call the constructor of its **ode_system** base class. The signature of this method is

```
ode_system (int N_vars, int M_event_funcs)
```

where N_vars is the number of entries in the q and dq arrays (i.e., the number of continuous state variables) and M_event_funcs is the number of entries in the z and state_event arrays (plus one for the time event). For the cherry bomb, N_vars is three and M_event_funcs is one.

The constructor for the cherry bomb does not initialize the continuous state variables. Instead, the simulator calls its ***init*** method whose signature is

```
void init(double* q)
```

where q is an array that should be filled with the initial values for the continuous variables $h$, $v$, and $t$. The complete implementation of the **CherryBomb** is listed below.

```
#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 ode_system<string> {
   public:
      CherryBomb():ode_system<string>(
            3, // three state variables including time
            1 // 1 state event condition
            ) {
         phase = FUSE_LIT; // Light the fuse!
```

```
      }
      void init(double *q) {
         q[H] = 1.0; // Initial height
         q[V] = 0.0; // Initial velocity
         q[T] = 0.0; // Start time at zero
      }
      void der_func(const double* q, double* dq) {
         dq[V] = -9.8;
         dq[H] = q[V];
         dq[T] = 1.0;
      }
      void state_event_func(const double* q, double *z) {
         // Test for hitting the ground.
         if (q[V] < 0.0) z[0] = q[H];
         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 external_event(double* q, double e, const Bag<string>& xb) {
         phase = DOUSE; // Any input is a douse event
      }
      void internal_event(double* q, const bool* state_event) {
         if (state_event[0]) q[V] = -q[V]; // Bounce!
         if (state_event[1]) phase = EXPLODE;
      }
      void confluent_event(double* q, const bool* state_event,
         const Bag<string>& xb) {
         internal_event(q,state_event);
         external_event(q,0.0,xb);
      }
      void output_func(const double *q, const bool* state_event,
            Bag<string>& yb) {
         if (state_event[1] && phase == FUSE_LIT)
            yb.insert("BOOM!"); // Explode!
      }
      void postStep(const double* q) {
         // Write the current state to std out
         cout << q[T] << " " << q[H] << " " << q[V] << " " << phase << endl;
      }
      // No garbage collection is needed
      void gc_output(Bag<string>&){}
      // Get the current value of the discrete variable
      Phase getPhase() { return phase; }
   private:
      Phase phase;
};
```

The **CherryBomb** itself is not derived from **Atomic** and so cannot be simulated directly. Rather, it is given to a **Hybrid** object, which is a kind of **Atomic**, that generators the trajectories for the model. This **Hybrid** object is used just like any other **Atomic** model. Input to this **Hybrid** object triggers an

input event for the **ode_system** that is contains. Likewise, output from the **ode_system** becomes output from the **Hybrid** object. Most importantly, the hybrid model can be part of any network of discrete event models.

A **Hybrid** object is provided with three things when it is constructed. First is the **ode_system** itself. Second is an **ode_solver** that produces the model's continuous trajectories. Adevs has two types of **ode_solvers**: a **corrected_euler** solver that uses the corrected Euler method and a **rk_45** solver that uses a fourth/fifth order Runge-Kutta method. Third is an **event_locator** that finds the location of state events as the simulation progresses. Adevs has two these: the **linear_event_locator** and **bisection_event_locator**. The code below shows how these are used to create and simulate a **Hybrid** object.

```
int main() {
   // Create the model
   CherryBomb* bomb = new CherryBomb();
   // Create the ODE solver for this model. Maximum error
   // tolerance at each step is 1E-4 and the maximum
   // size of an integration step is 0.01.
   ode_solver<string>* ode_solve =
      new corrected_euler<string>(bomb,1E-4,0.01);
   // Create the event locator for this model. Maximum
   // error tolerance for the location of an event in
   // the state space is 1E-8.
   event_locator<string>* event_find =
      new linear_event_locator<string>(bomb,1E-8);
   // Create an atomic model that puts all of these
   // together to simulate the continuous system.
   Hybrid<string>* model =
      new Hybrid<string>(bomb,ode_solve,event_find);
   // Create and run a simulator for this model
   Simulator<string>* sim = new Simulator<string>(model);
   while (bomb->getPhase() == FUSE_LIT)
      sim->execNextEvent();
   delete sim; delete bomb;
   return 0;
}
```

Figure 7.1 shows the cherry bomb's trajectory from $t = 0$ to its explosion at $t = 2$. This plot was produced using the simulation program listed above. 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 descent.

## 7.2 Modeling hybrid systems with the Functional Mockup Interface

The Functional Mockup Interface (FMI) is a standard API for encapsulating piecewise continuous models. The intent of this standard is to allow model developers to exchange their models in a way that is amenable to accurate simulation. The FMI interface specification can be found at `http://www.fmi-standard.org`. The practical advantage of this standard for modelers using adevs is that all major Modelica compilers can export their models in this form, and these can be wrapped in an ode_system object.

You will need three things to use this capability:

1. The FMI header files, which can be downloaded from the FMI website `http://www.fmi-standard.org`.
2. A Modelica compiler or other continuous system modeling tool that will export models in the FMI format.

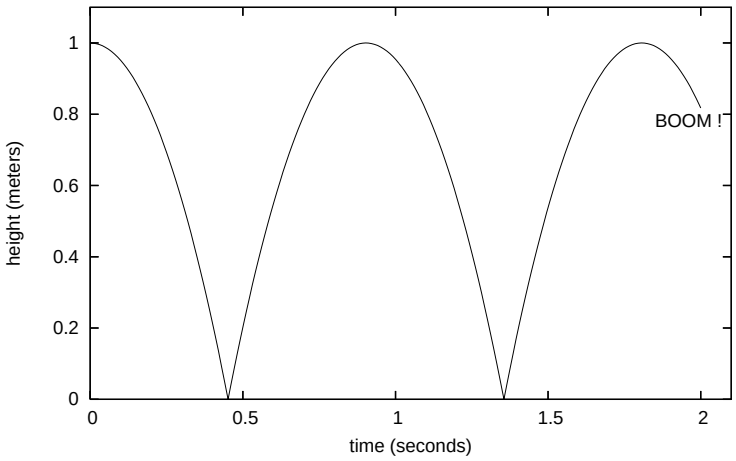


Figure 7.1: A simulation of the cherry bomb model that terminates when the cherry bomb explodes.

It is also helpful to have a passing familiarity with the FMI markup language for describing a model. Documentation for this markup language can be found in the FMI standard.

The cherry bomb from the previous section will illustrate how the FMI import feature is used. This example can be found in the examples/fmi/Example3 directory. We will create a bomb that begins at its initial height of 1 meter and remains there until it is dropped. Dropping the ball also starts the fuse. The falling and bouncing behavior of the bomb and its explosion after 2 seconds are implemented as a Modelica model. For this example we will omit the dousing. The Modelica code for this model is shown below.

```
model CherryBomb
   Real h(start=1); // Height of the bomb
   Real v(start=0); // Velocity of the bomb
   parameter Real g = 9.8; // Gravity
   Real fuseTime(start=2.0); // Fuse duration
   input Boolean dropped(start=false);
   output Boolean exploded(start=false);
equation
   // Accelerate downward if we have been lit
   der(v) = if (not dropped) then 0 else -g;
   der(h) = v;
   // Burn the fuse
   der(fuseTime) = if (not dropped) then 0 else -1;
   when {h <= 0} then
      reinit(v,-pre(v));
   end when;
   when {fuseTime <= 0} then
      exploded = true;
   end when;
end CherryBomb;
```

I will use the OpenModelica compiler to demonstrate how this is converted to an FMI compliant object file that can be used with adevs. Assume that the compiler omc is in our path. We begin by creating a Modelica script file called *makefmi.mos* and put into this file the following commands.

```
loadFile("CherryBomb.mo");
translateModelFMU(CherryBomb,"2.0");
```

The first line loads the CherryBomb model that is listed above. The second line generates and FMI object from this model that is compliant with the FMI v2 standard. This FMI object will be packed with several other files in a zipped file called *CherryBomb.fmu*. The OpenModelica compiler will also produce a bunch of other junk that you don't need, and these extra files can be deleted with the following set of command.

```
rm -f CherryBomb_*
rm -f CherryBomb.c
```

To get at the files you need, decompress the zip file with the command

```
unzip -o -qq CherryBomb.fmu
```

This will create directory called *sources*, which you can delete, a file called *modelDescription.xml*, which you will need, and a file *CherryBomb.so*, which you will need. This shared object file will be in the directory called *binaries/linux64*.

The file *modelDescription.xml* contains all of the information you will need to program your simulator to access the variables of the Modelica model. The listing of the description file for the cherry bomb is shown below. The significance of the data in this file will become apparent as we construct the rest of the simulation software that will interact with this model.

```
<?xml version="1.0" encoding="UTF-8"?>
<fmiModelDescription
  fmiVersion="2.0"
  modelName="CherryBomb"
  guid="{8c4e810f-3df3-4a00-8276-176fa3c9f9e0}"
  description=""
  generationTool="OpenModelica Compiler 1.9.2+dev (r24043)"
  generationDateAndTime="2015-02-08T18:41:22Z"
  variableNamingConvention="structured"
  numberOfEventIndicators="3">
  <ModelExchange
    modelIdentifier="CherryBomb">
  </ModelExchange>
  <TypeDefinitions>
  </TypeDefinitions>
  <LogCategories>
    <Category name="logEvents" />
    <Category name="logSingularLinearSystems" />
    <Category name="logNonlinearSystems" />
    <Category name="logDynamicStateSelection" />
    <Category name="logStatusWarning" />
    <Category name="logStatusDiscard" />
    <Category name="logStatusError" />
    <Category name="logStatusFatal" />
    <Category name="logStatusPending" />
    <Category name="logAll" />
    <Category name="logFmi2Call" />
  </LogCategories>
  <DefaultExperiment startTime="0.0" stopTime="1.0" tolerance="1e-06"/>
  <ModelVariables>
  <!-- Index of variable = "1" -->
  <ScalarVariable
    name="fuseTime"
    valueReference="0"
```

```
  variability="continuous"
  causality="local"
  initial="approx">
  <Real start="2.0"/>
</ScalarVariable>
<!-- Index of variable = "2" -->
<ScalarVariable
  name="h"
  valueReference="1"
  variability="continuous"
  causality="local"
  initial="approx">
  <Real start="1.0"/>
</ScalarVariable>
<!-- Index of variable = "3" -->
<ScalarVariable
  name="v"
  valueReference="2"
  variability="continuous"
  causality="local"
  initial="approx">
  <Real start="0.0"/>
</ScalarVariable>
<!-- Index of variable = "4" -->
<ScalarVariable
  name="der(fuseTime)"
  valueReference="3"
  variability="continuous"
  causality="local"
  initial="calculated">
  <Real derivative="1"/>
</ScalarVariable>
<!-- Index of variable = "5" -->
<ScalarVariable
  name="der(h)"
  valueReference="4"
  variability="continuous"
  causality="local"
  initial="calculated">
  <Real derivative="2"/>
</ScalarVariable>
<!-- Index of variable = "6" -->
<ScalarVariable
  name="der(v)"
  valueReference="5"
  variability="continuous"
  causality="local"
  initial="calculated">
  <Real derivative="3"/>
</ScalarVariable>
<!-- Index of variable = "7" -->
<ScalarVariable
```

```
  name="g"
  valueReference="6"
  variability="fixed"
  causality="parameter"
  initial="exact">
  <Real start="9.800000000000001"/>
</ScalarVariable>
<!-- Index of variable = "8" -->
<ScalarVariable
  name="_D_whenCondition1"
  valueReference="0"
  variability="discrete"
  causality="local"
  initial="calculated">
  <Boolean/>
</ScalarVariable>
<!-- Index of variable = "9" -->
<ScalarVariable
  name="_D_whenCondition2"
  valueReference="1"
  variability="discrete"
  causality="local"
  initial="calculated">
  <Boolean/>
</ScalarVariable>
<!-- Index of variable = "10" -->
<ScalarVariable
  name="dropped"
  valueReference="2"
  variability="discrete"
  causality="input"
  initial="approx">
  <Boolean start="false"/>
</ScalarVariable>
<!-- Index of variable = "11" -->
<ScalarVariable
  name="exploded"
  valueReference="3"
  variability="discrete"
  causality="output"
  initial="approx">
  <Boolean start="false"/>
</ScalarVariable>
</ModelVariables>
<ModelStructure>
  <Outputs>
    <Unknown index="11" dependencies="1" dependenciesKind="dependent" />
  </Outputs>
  <Derivatives>
    <Unknown index="4" dependencies="10" dependenciesKind="dependent" />
    <Unknown index="5" dependencies="3" dependenciesKind="dependent" />
    <Unknown index="6" dependencies="10" dependenciesKind="dependent" />
```

```
    </Derivatives>
    <InitialUnknowns>
    </InitialUnknowns>
  </ModelStructure>
</fmiModelDescription>
```

We will embed this model in a simple simulation program that has two components: the bomb and a separate model that lights the fuse and reports the explosion. The mischievous miscreant that lights the fuse and chuckles when it goes off is described with an atomic model just like those that we've seen before. Its source code is listed below.

```
/**
 * A miscreant drops the ball and reports the explosion.
 */
class Miscreant:
   public adevs::Atomic<std::string>
{
   public:
      Miscreant():
         adevs::Atomic<std::string>(),
         start(true),
         tstart(1.0)
      {
      }
      double ta() { return ((start) ? tstart : adevs_inf<double>()); }
      void delta_int() { start = false; }
      void delta_ext(double e, const Bag<std::string>& xb)
      {
         cout << (tstart+e) << " " << (*(xb.begin())) << endl;
      }
      void delta_conf(const Bag<std::string>&){}
      void output_func(Bag<std::string>& yb)
      {
         if (start) yb.insert("light");
      }
      void gc_output(Bag<std::string>&){}
   private:
      bool start;
      const double tstart;
};
```

Our next goal will be to create code for the CherryBomb model that appears in the main function of the simulation program. As you can see from the code below, this model is used just like any other model that is derived from the ode_system class, and can be coupled to atomic and other models just like any other component in a simulation program.

```
int main()
{
   // Create our model of the bomb
   CherryBomb* bomb = new CherryBomb();
   // Wrap a set of solvers around it
   Hybrid<std::string>* hybrid_model =
      new Hybrid<std::string>
      (
```

```
        bomb, // Model to simulate
        new corrected_euler<std::string>(bomb,1E-5,0.01), // ODE solver
        new discontinuous_event_locator<std::string>(bomb,1E-5) // Event locator
        // You must use this event locator for OpenModelica because it does
        // not generate continuous zero crossing functions
     );
  // Couple the miscreant and the bomb
  SimpleDigraph<std::string>* model = new SimpleDigraph<std::string>();
  Miscreant* miscreant = new Miscreant();
  model->add(miscreant);
  model->add(hybrid_model);
  model->couple(miscreant,hybrid_model);
  model->couple(hybrid_model,miscreant);
  // Create the simulator
  Simulator<std::string>* sim =
     new Simulator<std::string>(model);
  // Run the simulation for ten seconds
  while (sim->nextEventTime() <= 4.0)
  {
     cout << sim->nextEventTime() << " ";
     sim->execNextEvent();
     cout << bomb->get_height() << " " << bomb->get_fuse() << endl;
  }
  // Cleanup
  delete sim;
  delete hybrid_model;
  // Done!
  return 0;
}
```

The CherryBomb class is constructed as a subclass of the adevs FMI class. The FMI class contains all of the instructions needed to load, initialize, update, and access the variables of an FMI v2 object. The construction of the FMI class requires four arguments:

1. The model name that appears in the *modelDescription.xml* within the tag "modelName".
2. The GUID string for the FMI object that appears within the tag "guid" just beneath "modelName" in *modelDescription.xml*.
3. The number of continuous state variables in the Modelica model, which is determined by counting the number of variable descriptions containing the tagged "Real derivative". There are three of these for the CherryBomb model.
4. The number of state event conditions in the model, which is contained in the tag "numberOfEventIndicators".
5. The name of the shared object file that contains the equations, etc. for the compiled Modelica model.

The CherryBomb class inherits the functions of the ode_system class described in the previous section, and the default implementation of these methods by the FMI class is sufficient to simulate the parts of the model's behavior that are described in the Modelica file. If these methods are overridden, then it is necessary to first invoke the method of the base class before performing any other action. If any of the Modelica variables are modified by the overridden method, then it is also necessary to invoke the method of the base class after the modifications have been made. Variables that are part of the Modelica model can be read and written using methods inherited from the FMI class. These methods have the forms ***set_int(int index, int***

***val)***, ***get_int(int index): int***, ***set_double(int index, double val)***, ***get_double(int index): double***, ***set_bool(int index, bool val)***, and ***get_bool(int index): bool***. The index variable for each method should be valueReference number that appears in the file *modelDescription.xml*. Each of these elements is illustrated in the implementation of the CherryBomb class below.

```
class CherryBomb:
   // Derive model from the adevs FMI class
   public FMI<std::string>
{
   public:
      // Constructor loads the FMI
      CherryBomb():
         // Call FMI constructor
         FMI<std::string>
         (
            "CherryBomb", // model name from modelDescription.xml
            "{8c4e810f-3df3-4a00-8276-176fa3c9f9e0}", // GUID from modelDescription.xml
            3, // Number of derivative variables
            3, // numberOfEventIndicators from modelDescription.xml
            "binaries/linux64/CherryBomb.so" // Location of the shared object file produced by omc
         )
      {
      }
      // Internal transition function
      void internal_event(double* q, const bool* state_event)
      {
         // Call the method of the base class
         FMI<std::string>::internal_event(q,state_event);
      }
      // External transition function
      void external_event(double* q, double e, const Bag<std::string>& xb)
      {
         // Call the method of the base class
         FMI<std::string>::external_event(q,e,xb);
         // Drop the ball
         set_dropped();
         // Call the base class method again to
         // recalculate variables that depend on
         // the dropped variable
         FMI<std::string>::external_event(q,e,xb);
      }
      // Time remaining to the next sample
      double time_event_func(const double* q)
      {
         // Return the value of the base class
         return FMI<std::string>::time_event_func(q);
      }
      // Print state at each output event
      void output_func(const double* q, const bool* state_event, Bag<std::string>& yb)
      {
         // Output on the state event that is the explosion.
         // The state event number is in modelDescription.xml
```

```
         // as a whenCondition variable.
         if (state_event[1])
            yb.insert("boom!");
      }
      // The exploded variable in the Modelica model is its
      // third boolean variable. This information is in the
      // modelDescription.xml file generated by the Modelica
      // compiler.
      bool get_exploded() { return get_bool(3); }
      // See modelDescription.xml
      double get_height() { return get_real(1); }
      // See modelDescription.xml
      void set_dropped() { set_bool(2,true); }
      // See modelDescription.xml
      bool get_dropped() { return get_bool(2); }
      // See modelDescription.xml
      double get_fuse() { return get_real(0); }
};
```

# Chapter 8

# The Simulator Class

The **Simulator** class has four functions: to determining the model's time of next event, to extract output from the model, to inject input into the model, and to advance the simulation clock. The first function is implemented by the ***nextEventTime*** method with which we are already familiar. I'll address the other three functions in turn.

There are two essential steps for extracting output from your model. The first step is to register an **EventListener** with the simulator. This is done by creating a subclass of the **EventListener** and then passing this object to the **Simulator**'s ***addEventListener*** method. When the **EventListener** is registered with the simulator, its ***outputEvent*** method intercepts output originating from **Atomic** and **Network** models.

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 signature of ***computeNextOutput*** is

```
void computeNextOutput()
```

and it computes the model output at the 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** registered with the **Simulator**. The ***computeNextOutput*** method anticipates the output of your model from its current state assuming that no input events will intervene between now and the time returned by ***nextEventTime***.

The ***computeNextState*** method is used to inject events into a model 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 **Event** objects 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. However, 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 does its job using ***computeNextOutput*** and ***computeNextState***. The implementation of ***execNextEvent*** has only two lines; the **Bag** bogus_input is empty.

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

The ***computeNextOutput***, ***computeNextState***, and ***execNextEvent*** methods throw an exception if a model violates either of two constraints: i) the time advance is negative and ii) the coupling constraints described in section 5.1 and illustrated in Figure 5.4 are violated. The Adevs **exception** class is derived from the standard C++ **exception** class. Its method ***what*** returns a string that describes the exception condition and the method ***who*** returns a pointer to the model that caused the exception.

The Adevs **exception** class is intended to assist with debugging simulations. 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 identify problems before they become obscure and difficult to find bugs.

# Chapter 9

# Simulation on multi-core computers

Adevs has a ParSimulator class that is designed specifically to take advantage of processors with multiple cores and shared memory machines with several processors. The parallel simulator is in most respects identical to the sequential simulator, and this section of the manual therefore focuses on where it is different.

The **ParSimulator** class is designed specifically for symmetric, shared memory multiprocessors (SMPs). The multi-core processors that have become ubiquitous in recent years are an important instance of this class of machines. The software technology that underlies the **ParSimulator** is OpenMP (see `http://www.openmp.org`), which is an extension of the C and C++ compilers and runtime systems to support multi-threaded computing. The OpenMP standard is now supported by most (probably all) major compilers: the GNU C++ Compiler and professional editions of Microsoft Visual Studio are important examples (important to me, that is, because those are what I use for most of my simulation work).

The critical first step, therefore, to using the **ParSimulator** is to enable the OpenMP extensions for your compiler. For the GNU C++ compiler, simply add the flag '-fopenmp' to your linker and compiler arguments. For MS Visual Studio, this is a build option. For other compilers and development environments, see your documentation. Prior to executing a simulation, the maximum number of threads that will be used by OpenMP (and, therefore, the simulator) can be set with the OMP_NUM_THREADS environment variable (this works for the GNU compilers, at least). The default in most cases is to use a number of threads equal to the number of processors or cores in your computer.

Having enabled the OpenMP options for your compiler, you are ready to start preparing your model to work with the parallel simulator. As a first step, you can do the following. This example assumes that your main simulation routine looks something like this:

```
...
Simulator<IO_Type>* sim = new Simulator<IO_Type>(my_model);
/**
 * Register listeners with the Simulator to collect statistics
 * ....
 */
while (sim->nextEventTime() < t_end)
    sim->execNextEvent();
...
```

or this

```
...
Simulator<IO_Type>* sim = new Simulator<IO_Type>(my_model);
/**
 * Register listeners with the Simulator to collect statistics
 * ....
```

```
 */
sim->execUntil(t_end);
...
```

which does the same thing. The reason for this assumption is described in the section on limitations. Note, however, that any Listeners you have registered with your Simulator instance will work normally (almost, I'll get to that).

Assuming you have code like the above, replace it with code like the following:

```
...
ParSimulator<IO_Type>* sim = new ParSimulator<IO_Type>(my_model);
/**
 * Register listeners with the ParSimulator to collect statistics
 * ....
 */
sim->execUntil(t_end);
...
```

This should work just like your previous code with the following caveats.

First, your models must not share variables. All information exchanged between models must occur via events at their input and output. Moreover, the order of items in the bags of input received by your model may change from run to run, though the contents are guaranteed to be the same. Therefore, the repeatability of your simulation runs depends on your models being insensitive to the order of elements in their input bags.

Second, reports produced by your listeners may be formatted in ways you do not expect. For any individual atomic model, the listing of state transitions and output events will be in time order. Across models, however, this may not be the case. For example, suppose that you have two atomic models arranged as follows: A-¿B. You may see all state transitions and output for A listed first, followed by all state transitions and outputs for B. Most likely, these will be intermingled. Note too that the output reported for network models may not be in its proper time order, though the output reported for its atomic components will be.

The reason for these orderings is that, though the simulation of each model is done in the proper time order of its events, the ParSimulator overlaps simulation of models whenever this is possible. In the above example, it is possible to simulate model A without worrying about what B is doing because B never provides input to A. The simulator (if properly configured) will take advantage of this to simulate A and B in parallel. Hence, we may see all of the output from A before we see anything for B. Once again, however, all callbacks to registered Listeners for any particular atomic model will be in the proper time order.

Note too that this implies that callbacks to your listeners may occur in parallel. In the above example, it may be that the same listener receives concurrent notifications about a state change for model A and state change for model B, or output from A and state change of B, or any such combination. It is imperative therefore that the callbacks in your listeners be thread safe.

This implies also that your network models must have routing methods that are thread safe. This is true for the network models that are included with Adevs. If you have implemented your own network models, be sure that their ***route*** methods are thread safe.

Third, and most critical, your atomic components must implement the new ***lookahead*** method, which is inherited from the **Devs** base class. This method must return a positive value subject to the guarantees described in section 9.2. For the purposes of getting your code to compile and run, your lookahead methods can simply return a very small value (i.e., something positive but close to zero; say 1E-8 or 1E-9). In this case, your simulator should compile and (very slowly) execute. Even if you have the patience to wait for it to complete, however, the outcome will likely be wrong. Nonetheless, such a test will determine if your build environment is setup properly.

The above changes are sufficient in most cases to make your existing, sequential simulator execute with the parallel simulator. In summary, these mandatory steps are:

1. Replace your **Simulator** with a **ParSimulator**.
2. Use the **ParSimulator**'s ***execUntil*** method to advance time.
3. Make sure your **Listener**s are thread safe.
4. Make sure your **Network** ***route*** methods are thread safe.
5. Implement the ***lookahead*** method for your **Atomic** models.

These steps alone are unlikely to yield an improvement in performance (or, indeed, correct results). As a general rule, correctly speeding up your simulation requires giving the **ParSimulator** specific information about your model; information that only you can provide. Without this information, the synchronization overhead incurred by the parallel simulation algorithm is staggeringly huge. The majority of this document deals with the problem of creating parallel simulations that are fast and execute correctly.

## 9.1 Limits of the parallel simulator

Before continuing any effort to make your simulator work with the algorithms used by the **ParSimulator**, you should be aware of specific capabilities of the sequential simulator that the parallel simulator does not support. These are:

1. The ***execNextEvent***, ***computeNextState***, and ***computeNextOutput*** methods are not provided. Only the ***execUntil*** method is provided for advancing the simulation.
2. As noted above, callbacks to a **Listener** object for each individual atomic model will be given in the proper time order, but these may be arbitrarily interleaved with the callbacks for other atomic models.
3. ***Listener*** callbacks must be made thread safe using the OpenMP synchronization features.
4. The parallel simulator does not support models that change structure. If your models implement their ***model_transition*** method, then you cannot use the parallel simulator.

Beyond these purely technical limits, also note that making effective use of this (or any) parallel discrete event simulator is difficult. Applications of practical interest require the identification of lookahead for the model's atomic components, partitioning of the model among the available processors, and implementing code to enable the parallel simulator to take advantage of your model's lookahead and partitioning.

So while this guide addresses issues specific to using the **ParSimulator** class, I strongly recommend that, if you are not already intimately familiar with parallel simulation, that you obtain a book on the topic. "Building Software for Simulation", authored by James Nutaro (the author of this manual) and published by Wiley in 2010, contains a chapter on conservative discrete event simulation with DEVS in general and Adevs in particular. You may find this book to be a useful introduction, though there are other excellent texts on the subject.

## 9.2 Modifying your models to exploit lookahead

The **ParSimulator** takes advantage of a property intrinsic to many models: their strong causality. A model, network or atomic, is strongly causal if its output can be predicted with certainty to some future time without knowledge of its input. The length of time into the future for which this prediction can be made is called lookahead.

To illustrate, consider a very simple, atomic model that acts as a transcriber. In the absence of input, the model neither changes state nor produces output: its time advance is infinite. Upon receiving an input, the model retains it for exactly one unit of time and then expels it. So, for example, if the input to the system is the series of letters 'A', 'B', and 'C' at times 1, 2, and 3 respectively, then its output is 'A', 'B',

and 'C' at times 2, 3, and 4 respectively. If the model receives an input while transcribing, then that input is discarded. So, for example, if 'A', 'B', and 'C' arrive at times 2, 2.5, and 3 then the output from the model is 'A' and 'C' at times 3 and 4.

If I know the input to this model until some time t, then I can determine its output until time t+1. Significantly, I do not need to know its input in the interval from t to t+1. For instance, suppose I know that the input at time zero is 'A'. Clearly, the only output of the model in the interval 0 to 1 (inclusive) is the value 'A' at time 1. Moreover, suppose I know that the input at t=0 is 'A' and that there is no other input until at least time 0.5. In this case, I know that the output until (but not including) time 1.5 consists only of 'A' at time 1. Any input following time 0.5 cannot occur until, at its earliest, time 1.5. This model has a lookahead of 1. Given its input to time t, its output is fixed to time t+1. Output in this interval does not depend on input in the same interval.

In network models, lookahead may accumulate. As an example, suppose that two transcribers are connected in series. In this case, the lookahead of the composite is two; i.e., the sum of the lookaheads of the components. If, however, a network model comprises two transcribers in parallel, then the lookahead of the network is only one. Generally speaking, however, larger networks tend to have larger lookaheads, and large lookahead is essential for getting good performance from the simulator.

To take advantage of lookahead in a model, the simulator must be told that it exists. This is done by overriding the lookahead method of the **Devs** class. All atomic and network models inherit this method from the **Devs** base class, and its default implementation is to return zero.

Lookahead is most useful to the simulator if it is coupled with a capability to actually calculate the model's outputs to this time. To calculate those outputs, however, requires knowledge of the intervening states. An atomic model enables the simulator to project its output into the future by implementing two method: ***beginLookahead*** and ***endLookahead***.

The ***beginLookahead*** method is called to notify the model that further calculations of its outputs and state transitions are speculative. The default behavior of the ***beginLookahead*** method is to throw an exception, which notifies the simulator that this model does not support projecting its output into the future. An atomic model overriding this method must be capable of restoring its state variables to the instant that ***beginLookahead*** was called. The ***endLookahead*** method is called by the simulator when it is done projecting that model's output. This method must restore the model to the same state it was in when the ***beginLookahead*** method was called.

These methods are demonstrated by the **Transcribe** class shown below. This atomic model implements the transcriber described above.

```
/**
 * This model copies its input to its output following a
 * delay.
 */
class Transcribe:
    public adevs::Atomic<char>
{
    public:
        Transcribe():adevs::Atomic<char>(),ttg(DBL_MAX),to_transcribe(' '){}
        void delta_int() { ttg = DBL_MAX; }
        void delta_ext(double e, const adevs::Bag<char>& xb)
        {
            if (ttg == DBL_MAX)
            {
                ttg = 1.0;
                // Find the largest input value
                adevs::Bag<char>::const_iterator iter = xb.begin();
                to_transcribe = *iter;
                for (; iter != xb.end(); iter++)
```

```
                {
                    if (to_transcribe < *iter)  to_transcribe = *iter;
                }
            }
            else ttg -= e;
        }
        void delta_conf(const adevs::Bag<char>& xb)
        {
            delta_int();
            delta_ext(0.0,xb);
        }
        void output_func(adevs::Bag<char>& yb)
        {
            yb.insert(to_transcribe);
        }
        double ta() { return ttg; }
        void gc_output(adevs::Bag<char>&){}
        double lookahead() { return 1.0; }
        void beginLookahead()
        {
            // Save the state
            chkpt.ttg = ttg;
            chkpt.to_transcribe = to_transcribe;
        }
        void endLookahead()
        {
            // Restore the state
            ttg = chkpt.ttg;
            to_transcribe = chkpt.to_transcribe;
        }
        char getMemory() const { return to_transcribe; }
    private:
        double ttg;
        char to_transcribe;
        struct checkpoint_t { double ttg; char to_transcribe; };
        checkpoint_t chkpt;
};
```

A model comprising two transcribers connected in series is defined as follows.

```
/**
 * This model defines a pair of transcribers connected
 * in series as shown: -> t1 -> t2 ->.
 */
class Series:
    public adevs::Network<char>
{
    public:
        Series():
            Network<char>(),
            t1(),t2()
        {
            t1.setParent(this);
```

```
            t2.setParent(this);
        }
        void getComponents(adevs::Set<adevs::Devs<char>*>& c)
        {
            c.insert(&t1);
            c.insert(&t2);
        }
        void route(const char& x, adevs::Devs<char>* model,
            adevs::Bag<adevs::Event<char> >& r)
        {
            adevs::Event<char> e;
            e.value = x;
            if (model == this) e.model = &t1;
            else if (model == &t1) e.model = &t2;
            else if (model == &t2) e.model = this;
        }
        double lookahead()
        {
            return t1.lookahead()+t2.lookahead();
        }
    private:
        Transcribe t1, t2;
};
```

The above code examples illustrate all possible changes to your models - network and atomic - that might be needed to facilitate parallel simulation. Of these changes, only the ***lookahead*** method of the atomic model is actually required. The **ParSimulator** calculates default (and very conservative) lookaheads for the **Network** models if these are required. Atomic models that provide the ***endLookahead*** and ***beginLookahead*** methods may improve the execution time of the simulation. But only the ***lookahead*** values of the atomic models (or their parent if the model is partitioned by hand; see the next section) are actually required for correct execution.

## 9.3 Partitioning your model

Each thread in your simulator is assigned to the execution of a subset of the atomic components of your model. Models within a thread are executed sequentially. This simulation proceeds just as with the sequential **Simulator** class. The threads execute in parallel, each stopping to synchronize with its neighbors only as necessary to exchange essential information.

In an ideal partitioning of the model, each thread is assigned roughly the same number of models, each model requires roughly the same amount of computational effort, and models assigned to separate threads rarely exchange inputs and outputs. This is the ideal that you should strive for in assigning your models to threads.

The actual assignment of a model to a thread is straightforward. **Network** and **Atomic** objects inherit the ***setProc*** method from their ***Devs*** base class. To assign the model to a particular thread, pass the number of that thread to the ***setProc*** method before the **ParSimulator** is created. Threads are numbered from 0 to the maximum number of threads (i.e., OMP_NUM_THREADS) minus one.

As the **ParSimulator** setups the simulation, it examines the thread to which each model is assigned and take the following actions:

1. If the model's parent is assigned to thread k, then the model is also assigned to thread k, regardless of the argument passed to its ***setProc*** method.

2. If the model's parent was not assigned to specific thread, then the model will be assigned to the thread indicated by argument to its ***setProc*** method.

3. If neither the model nor its parent were assigned to a thread by calling the ***setProc*** method, then the model is assigned to a thread selected at random.

This partitioning of the model tells the simulator how to distribute its computational workload. It does not tell the simulator which parts of the workload communicate with each other. For example, if half of your model is assigned to thread 0 and the other half to thread 1, the simulator does not yet know if the models in 0 send input to the models in 1, vice versa, or both. Without further information, the simulator assumes that every thread contains models that must communicate with all other threads. This is the most conservative assumption, and it imposes a substantial synchronization overhead.

If you know something about how the models assigned to the threads communicate, then you can provide this information to the simulator. This is done by passing a **LpGraph** object to the constructor of the **ParSimulator**. The **LpGraph** is a directed graph and the presence of an edge from node k to node j indicates that the models in thread k send input to the models in thread j. Absence of an edge indicates no flow of information along the missing edge. An edge is added to the **LpGraph** by calling its ***addEdge(A,B)*** method. This creates an edge from thread A to thread B.

The following snippet of code illustrates the partitioning procedure. This segment of code creates the block diagram model shown in Fig. 9.1. This model consists of two atomic components, A and B, and a network with two components C1 and C2. The model A is assigned to thread 0, B to thread 1, and the network C with its components C1 and C2 to thread 2. With this partition, thread 0 sends input to thread 1, thread 1 sends input to thread 2, and thread 2 to thread 0.

```
ModelA* A = new ModelA();
ModelB* B = new ModelB();
NetworkModelC* C = new NetworkModelC();
SimpleDigraph<IO_Type>* model = new SimpleDigraph<IO_Type>();
model.add(A);
model.add(C);
model.add(B);
model.couple(A,B);
model.couple(B,C);
model.couple(C,A);
A.setProc(0);
B.setProc(1);
C.setProc(2);
LpGraph lpg;
lpg.addEdge(0,1);
lpg.addEdge(1,2);
lpg.addEdge(2,3);
ParSimulator<IO_Type> sim(model,lpg);
```

## 9.4 Partitioning and lookahead

In a large model with an explicit partitioning, it is not required that every model provide a lookahead value. The following rules dictate which models must provide positive lookahead.

1. An atomic model whose parent is not assigned to a specific thread must provide a positive lookahead.

2. A network that is assigned to a specific thread, but whose parent is not, must provide a positive lookahead.

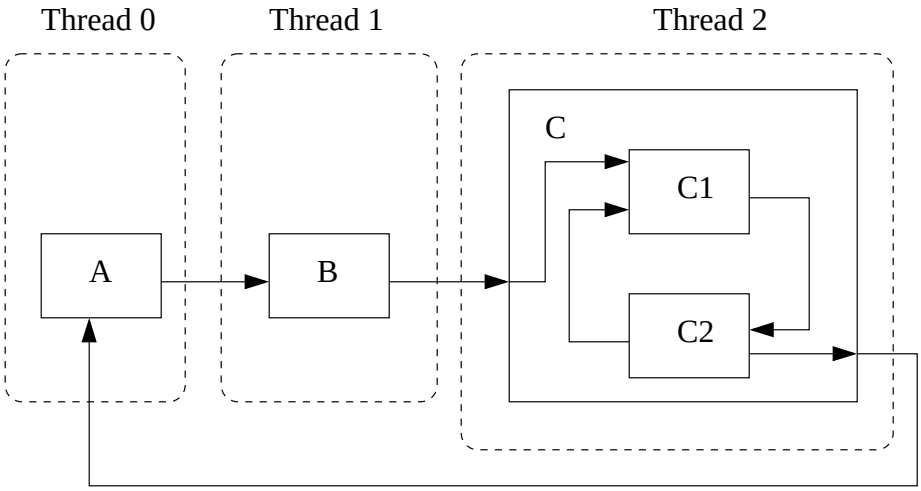


Figure 9.1: Partitioning a model for simulation on three processors.

3. No other model is required to provide a positive lookahead, or indeed any lookahead at all. The simulator will not use lookahead values provided by models except as indicated in cases 1 and 2 above.

## 9.5 A complete example

This example builds and simulates the network shown in Fig. 9.1 using the **Transcribe model** for components B, C1, and C2 and a **SimpleDigraph** for the network C. The network that is component A has two sub-components: a generator and a transcriber. Input to the network goes to the generator. Output from the generator goes to the transcriber, and output from the transcriber becomes an output from the network.

The generator operates as follows. It produces output at regular intervals of 1/2 units of time until it receives an input. At that time it stops. Note that the lookahead of the generator is zero because its pending output may be canceled at any time. The lookahead of the network, however, is one. That is, the lookahead of the network is the sum of its series components.

A **Listener** is constructed for this model to record the state and output trajectories of every component. This listener is made thread safe by the critical section placed around writes to standard output. It reports state and output trajectories for each atomic model in proper time order, but interleaves the trajectories of these models with each other.

The complete implementation of the model, listener, and simulator are shown below.

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

/**
 * Extend the SimpleDigraph class to allow its lookahead
 * to be set manually.
 */
class SimpleDigraphWithLookahead:
    public SimpleDigraph<char>
{
```

```
    public:
        SimpleDigraphWithLookahead():
            SimpleDigraph<char>(),
            look_ahead(0.0)
            {
            }
        void setLookahead(double look_ahead)
        {
            this->look_ahead = look_ahead;
        }
        double lookahead() { return look_ahead; }
    private:
        double look_ahead;
};

/**
 * Listener to record the output and state trajectories of the
 * component models.
 */
Genr* A_g;
Transcribe *A_t, *B, *C1, *C2;
SimpleDigraphWithLookahead *A, *C;

class Listener:
    public EventListener<char>
{
    public:
        Listener(){}
        void outputEvent(Event<char> y, double t)
        {
            string which = which_model(y.model);
            #pragma omp critical
            cout << which << " @ t = " << t << ", y(t)= " << y.value << endl;
        }
        void stateChange(Atomic<char>* model, double t)
        {
            if (model == A_g)
                #pragma omp critical
                cout << which_model(A_g) << " @ t = " << t << ", running= "
                    << A_g->isRunning() << ", next output= " <<
                    A_g->getNextOutput() << endl;
            else if (model == A_t)
                #pragma omp critical
                cout << which_model(A_t) << " @ t = " << t << ", memory= "
                    << A_t->getMemory() << ", ta()= " <<
                    A_t->ta() << endl;
            else if (model == C1)
                #pragma omp critical
                cout << which_model(C1) << " @ t = " << t << ", memory= "
                    << C1->getMemory() << ", ta()= " <<
                    C1->ta() << endl;
            else if (model == C2)
```

```
                #pragma omp critical
                cout << which_model(C2) << " @ t = " << t << ", memory= "
                    << C2->getMemory() << ", ta()= " <<
                    C2->ta() << endl;
            else if (model == B)
                #pragma omp critical
                cout << which_model(B) << " @ t = " << t << ", memory= "
                    << B->getMemory() << ", ta()= " <<
                    B->ta() << endl;
            else assert(false);
        }
    private:
        string which_model(Devs<char>* model)
        {
            if (model == A_g) return "A.A_g";
            if (model == A_t) return "A.A_t";
            if (model == A) return "A";
            if (model == B) return "B";
            if (model == C1) return "C.C1";
            if (model == C2) return "C.C2";
            if (model == C) return "C";
            assert(false);
            return "";
        }
};

int main(int argc, char** argv)
{
    // Component A
    A_g = new Genr();
    A_t = new Transcribe();
    A = new SimpleDigraphWithLookahead();
    A->setLookahead(A_t->lookahead()+A_g->lookahead());
    A->add(A_g);
    A->add(A_t);
    A->couple(A,A_g); // A -> A_g
    A->couple(A_g,A_t); // A_g -> A_t
    A->couple(A_t,A); // A_t -> A
    A->setProc(0); // Assign to thread zero
    // Component B
    B = new Transcribe();
    B->setProc(1); // Assign to thread one
    // Component C
    C1 = new Transcribe();
    C2 = new Transcribe();
    C = new SimpleDigraphWithLookahead();
    C->setLookahead(C1->lookahead()+C2->lookahead());
    C->add(C1);
    C->add(C2);
    C->couple(C,C1); // C -> C1
    C->couple(C2,C); // C2 -> C
    C->couple(C1,C2); // C1 -> C2
```

```
    C->couple(C2,C1); // C2 -> C1
    C->setProc(2); // Assign to thread two
    // Create the overarching model
    SimpleDigraph<char>* model  = new SimpleDigraph<char>();
    model->add(A);
    model->add(B);
    model->add(C);
    model->couple(A,B);
    model->couple(B,C);
    model->couple(C,A);
    // Create the corresponding LPGraph
    LpGraph lpg;
    lpg.addEdge(0,1);
    lpg.addEdge(1,2);
    lpg.addEdge(2,0);
    // Create the simulator
    ParSimulator<char>* sim = new ParSimulator<char>(model,lpg);
    // Register the listener
    Listener* listener = new Listener();
    sim->addEventListener(listener);
    // Run the simulation until t=10
    sim->execUntil(10.0);
    // Cleanup and exit
    delete sim;
    delete listener;
    delete model;
    return 0;
}
```

A subset of the output produced by this simulation is shown below. The intermingling of reported events in time is immediately apparent.

```
...
C.C2 @ t = 7.5, y(t)= G
C @ t = 7.5, y(t)= G
C.C1 @ t = 7.5, y(t)= I
C.C2 @ t = 7.5, memory= I, ta()= 1
C.C1 @ t = 7.5, memory= G, ta()= 1
C.C1 @ t = 8.5, y(t)= G
C.C2 @ t = 8.5, y(t)= I
C @ t = 8.5, y(t)= I
C.C1 @ t = 8.5, memory= I, ta()= 1
C.C2 @ t = 8.5, memory= G, ta()= 1
A.A_g @ t = 7.5, running= 0, next output= I
A.A_g @ t = 8.5, running= 0, next output= I
A.A_g @ t = 9.5, running= 0, next output= I
C.C1 @ t = 9.5, y(t)= I
C.C2 @ t = 9.5, y(t)= G
C @ t = 9.5, y(t)= G
C.C1 @ t = 9.5, memory= G, ta()= 1
C.C2 @ t = 9.5, memory= I, ta()= 1
...
```

However, a search for just the events for model C1 gives the expected result: all of its events are listed

in their proper time order. Specifically, the command 'grep C2 output', where 'output' is the file with the results of the simulation, yields the following:

```
C.C2 @ t = 3.5, memory= A, ta()= 1
C.C2 @ t = 4.5, y(t)= A
C.C2 @ t = 4.5, memory= C, ta()= 1
C.C2 @ t = 5.5, y(t)= C
C.C2 @ t = 5.5, memory= E, ta()= 1
C.C2 @ t = 6.5, y(t)= E
C.C2 @ t = 6.5, memory= G, ta()= 1
C.C2 @ t = 7.5, y(t)= G
C.C2 @ t = 7.5, memory= I, ta()= 1
C.C2 @ t = 8.5, y(t)= I
C.C2 @ t = 8.5, memory= G, ta()= 1
C.C2 @ t = 9.5, y(t)= G
C.C2 @ t = 9.5, memory= I, ta()= 1
```

So too for the output of model 'A_g', which is shown below:

```
A.A_g @ t = 0.5, y(t)= A
A.A_g @ t = 0.5, running= 1, next output= B
A.A_g @ t = 1, y(t)= B
A.A_g @ t = 1, running= 1, next output= C
A.A_g @ t = 1.5, y(t)= C
A.A_g @ t = 1.5, running= 1, next output= D
A.A_g @ t = 2, y(t)= D
A.A_g @ t = 2, running= 1, next output= E
A.A_g @ t = 2.5, y(t)= E
A.A_g @ t = 2.5, running= 1, next output= F
A.A_g @ t = 3, y(t)= F
A.A_g @ t = 3, running= 1, next output= G
A.A_g @ t = 3.5, y(t)= G
A.A_g @ t = 3.5, running= 1, next output= H
A.A_g @ t = 4, y(t)= H
A.A_g @ t = 4, running= 1, next output= I
A.A_g @ t = 4.5, y(t)= I
A.A_g @ t = 4.5, running= 0, next output= I
A.A_g @ t = 5.5, running= 0, next output= I
A.A_g @ t = 6.5, running= 0, next output= I
A.A_g @ t = 7.5, running= 0, next output= I
A.A_g @ t = 8.5, running= 0, next output= I
A.A_g @ t = 9.5, running= 0, next output= I
```

So the individual traces for these components appear in the proper order in the output, but they are intermingled in an arbitrary way.

## 9.6 Managing memory across thread boundaries

Because the atomic models in a simulator are executed at different rates, it often happens that an output object produced by a model in one thread will be used at some later time by models in another thread. To manage the memory associated with these output objects, it is necessary for the simulator to be able to determine when any such object can be safely deleted. This is done most easily when every thread has its own copy of the object, and the **MessageManager** interface is used by the simulator for this purpose.

If your input and output types are primitive objects (ints, chars, etc.) or simple structure, then the default approach to memory management is sufficient. Indeed, the default memory manager should be sufficient for any types of objects for which the compiler's default copy constructor and assignment operator produce deep copies. If you are passing pointers to complex objects or objects that use their own internal scheme for managing memory (e.g., that use copy on write semantics, reference counting, etc.), then you will need to build a custom memory manager. The **MessageManager** interface is used for this purpose, and your custom **MessageManager** is provided to the **ParSimulator** as the final argument to its constructor.

The **MessageManager** has two virtual methods that must be overridden by any derived class. The first is the ***clone*** method, which has the signature

```
X clone(X& value)
```

where X is the type of object that your simulator uses for input and output. This method must create and return a deep copy of the value. The second method is ***destroy***, and it has the signature

```
void destroy(X& value)
```

where X is as before. This method must free the memory associated with the value.

The implementation of the default memory manager is as follows:

```
template <typename X> class NullMessageManager:
   public MessageManager<X>
{
   public:
      /// Uses the objects default copy constructor
      X clone(X& value) { return value; }
      /// Takes no action on the value
      void destroy(X& value){}
};
```

To illustrate the construction of a new **MessageManager**, the implementation below is for a model that uses C style strings (i.e., null terminate arrays of characters) for input and output. The ***clone*** method allocates memory for a string and then copies to it the contents of the value. The ***destroy*** method frees the memory allocated for the string.

```
class CStringMessageManager:
   public adevs::MessageManager<char*>
{
   public:
      char* clone(char* & value)
      {
         char* new_string = new char[strlen(value)];
         strcpy(new_string,value);
         return new_string;
      }
      void destroy(char* & value)
      {
         delete [] value;
      }
};
```

If this message manager were supplied to the simulator in the example of the previous section, then the **ParSimulator** would be constructed as follows:

```
. . .
CStringMessageManager* msg_mngr = new CStringMessageManager();
ParSimulator<char*>* sim = new ParSimulator<char*>(model,lpg,msg_mngr);
. . .
```

The simulator would then use the supplied message manager for handling input and output objects that exist simultaneously in the simulator's many threads.

## 9.7 Notes on repeatability and performance

If your model and simulator have been setup properly then the outcomes produced by the parallel and sequential simulators will be identical. To this end, keep the following rules in mind:

1. Models must not shared variables.
2. The state transition functions of your models must not depend on the order of items in their input bag.
3. Listeners must be thread safe.
4. Listeners must not expect events to be reported in a global time order. Only the events associated with individual atomic models will be reported in their proper order. All other events will be interleaved in time.

In general you should not expect to speedup relatively small simulations by use of the parallel simulator. Rather, its purpose is chiefly to enable your model to grow in its size, complexity, or both without a corresponding increase in execution time. With this in mind, to achieve good execution times requires the following.

1. The size of the model must be sufficient to keep all of your processors busy all of the time with useful work. Moreover, the amount of useful work to be done by each processor must substantially exceed the overhead of parallel simulation algorithm.
2. Your model must have parallelism that the parallel algorithm can exploit. In practice, this means that your model must be partitioned to both maximize the lookahead of the network assigned to each processor and to minimize the communication between processors.

# Chapter 10

# Models with Many Input/Output Types

It would be surprising if every component in a large model 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 then to derive specific types from the common base. The simulator and all of its components exchange pointers to the base class and downcast 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 **CellEvent** objects into the **PortValue** objects. A solution to this problem is to use 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 **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 while 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_outpu
```

The cleanup of converted objects is managed with the ***gc_output*** method, which is inherited from the **ModelWrapper**'s **Atomic** base class, plus a new ***gc_input*** method to cleanup objects created by the ***translateInput*** method: its signature is

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

The model to encapsulate is passed to the constructor of the **ModelWrapper**. 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 own 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*** methods 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 this 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*** method 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** objects 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 **WrapperModel** class. The **Wrapper** is derived from the **WrapperModel** and implements its four virtual methods: ***translateInput***, ***translateOutput***, ***gc_input***, and ***gc_output***. This class wraps an **Atomic** model that uses int* objects for input and output. The **Wrapper** uses C strings for its input and output. The translation methods 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 11

# Alternate types for time

The second template argument for the simulator and model classes is used to select an alternate representation of time for your simulation. The default type for time is double. This may be replaced with the primitive types int or the Adevs class **double_fcmp** by supplying those as the second template argument. If you want to use your own class for time, it must support the following:

1. Default constructor, copy constructor, and assignment operator.
2. All addition and subtraction operators.
3. All comparison operators.
4. A method ***adevs_inf*** that returns a value for infinity.
5. A method ***adevs_zero*** that returns a value for zero.
6. A method ***adevs_sentinel*** that returns a value less than zero.

The three methods ***adevs_inf***, ***adevs_zero***, and ***adevs_sentinel*** must be template functions defined as follows (these examples are for using the int primitive for time).

```
template <> inline int adevs_inf() {
    return std::numeric_limits<double>::max(); }
template <> inline int adevs_zero() { return 0; }
template <> inline int adevs_sentinel() { return -1; }
```

# Chapter 12

# Random Numbers

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

The **random_seq** class is the 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 method ***next_long*** returns a random number as an unsigned long. The method ***next_dlb*** refines the ***next_long*** method by reducing that 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 virtual 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]. This 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. Because 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 13

# Interpolation

Data is often made available as a set of tabulated points. Hourly temperature data, millisecond samples of voltage in an electric circuit, and the 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 radar tracks at quarter minutes then a method for approximating values between tabulated points is very useful. The **InterPoly** class exists for this purpose.

The **InterPoly** class approximates a continuous signal by fitting a polynomial to the available data points. The approximating polynomial passes 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) a better approximation.

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 the array of data values, t is the array of time points, and n is the number of points (i.e., the size of the u and t array). The constructor builds an $n - 1$ degree polynomial that fits the supplied data.

The second way is to 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 the 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 of these 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 the optional t array is 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

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

The ***interpolate*** method and ***operator()*** method 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, which may be used 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 an 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. 13.1 and 13.2. 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;
}
```

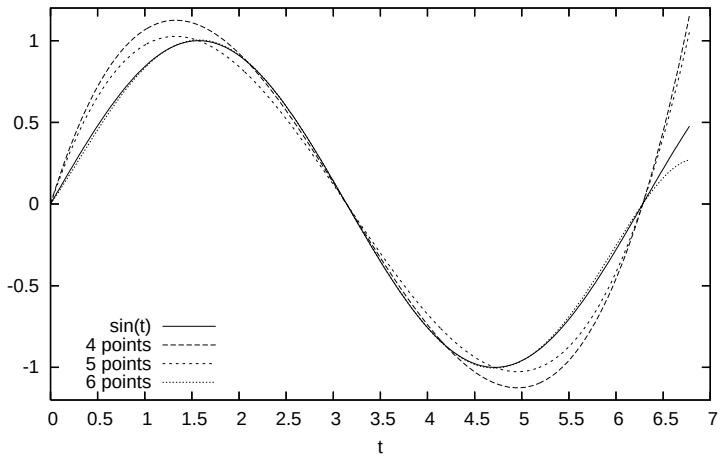


Figure 13.1: The function $\sin(t)$ and some interpolating polynomials with data spanning the interval $[0, 2\pi]$.

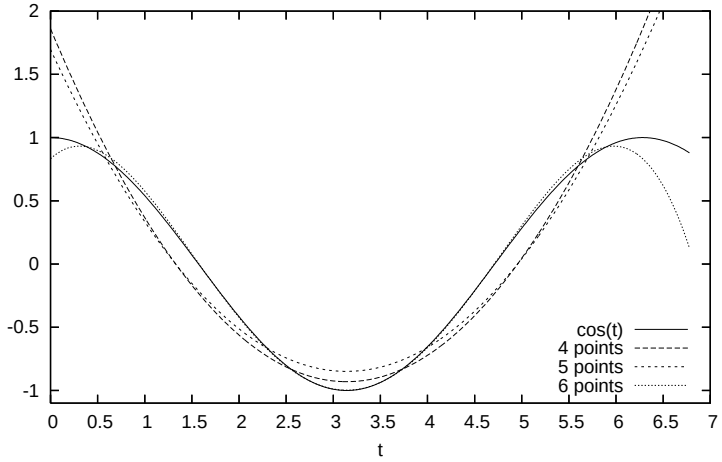


Figure 13.2: The time derivative of $\sin(t)$ and the time derivative of some interpolating polynomials with data spanning the interval $[0, 2\pi]$.