COST: A COMPONENT-ORIENTED DISCRETE EVENT SIMULATOR

Gilbert Chen
Boleslaw K. Szymanski

Department of Computer Science
Rensselaer Polytechnic Institute
110 8th Street
Troy, NY 12180, U.S.A.

ABSTRACT

COST (Component-Oriented Simulation Toolkit) is a general-purpose discrete event simulator. The main design purpose of COST is to maximize the reusability of simulation models without losing efficiency. To achieve this goal, COST adopts a component-based simulation worldview based on a component-port model. A simulation is built by configuring and connecting a number of components, either off-the-shelf or fully customized. Components interact with each other only via input and output ports, thus the development of a component becomes completely independent of others. The component-port model of COST makes it easy to construct simulation components from scratch. Implemented in C++, COST also features a wide use of templates to facilitate language-level reuse.

1   INTRODUCTION

Discrete event simulation is a very effective method for analyzing existing or to-be-built systems. Although all physical systems are indeed continuous, many of them can be viewed as discrete systems if details below a certain level can be abstracted away. For example, computer systems, computer networks, digital logic, traffic systems, to name a few, are all classical subjects of discrete event simulation. The benefit of simulation is that, by constructing a simulation model with the computer, we are able to study the system without actually manipulating or building it physically.

Our attempt to build yet another discrete event simulator is motivated by our recent progress in the concept of component-based simulation (Chen and Szymanski 2001). More specifically, we proposed a component-oriented simulation worldview that takes a divide-and-conquer approach to simulation modeling. In order to make component composable, we introduced a simulation component classification that groups components into three classes: timeless, time-dependent, and time-independent. Timeless

components have no notions of simulation time. Time-dependent components are aware of the existence of the simulation time, but cannot change it, while time-independent components maintain their own simulation clock. It is therefore a natural choice to build an entirely new simulator with these new understandings.

A good simulator possesses two essential features. First, it must support reusable models. A model written for one simulation should be able to be effortlessly embedded into other simulations that require the same kind of a model. Second, the model should be easy to be built from scratch. Interestingly, we observe that most existing simulators do not possess these two features simultaneously. Most commercial simulators provide a reusable model library, often coming with a friendly graphical user interface, but adding new models to the library is always a painful task. On the other hand, most freely available simulators follow a bottom-up approach; writing models from scratch is straightforward, but the reusability is severely limited.

COST attempts to address these two problems simultaneously. The key to the solution is the component-oriented worldview as well as the underlying component-port model, which are described in depth in Section 2. Section 3 mainly discusses the design and implementation issues of COST. Section 4 gives a detailed example of an M/M/1 simulation on top of COST.

2   COMPONENT-BASED SIMULATION

The component-oriented worldview sees a simulation as being composed of a set of components. It takes a divide-and-conquer approach in which the whole simulation is partitioned into a number of smaller simulation tasks, which are modeled by each component individually. The immediate benefit of doing so is that the complexity is significantly reduced. Each component is now a smaller task whose internal logic is much simpler than that the whole simulation. With this approach, reusability can also be

achieved if the components are designed in such a way that context-relevant information is not embedded into the code of the components.

The component-port model ensures that any component is completely independent of the simulation in which it will be used. Components exchange messages with each other only via input or output ports. To send out a message, the component simply places the message in the desired output port. The receiver is not determined until the configuration phase, in which an output port is connected to one or more input ports. Messages placed in the output port in the run time will then be delivered to connected input port(s). Similarly, the component responds to messages arriving at an input port, regardless of the sender, thus, it is able to accept messages at the input port sent by any other component. These properties of input and output ports and the accompanied configuration phase are the source of the independence between the components and the simulations, and such independence results in maximum reusability.

The term component has been used in computer science often without elaboration. We define a component as an object with an interface that enables it to be combined with other objects. The interface, either explicit or implicit, prescribes in what kind of interaction the components would be involved with other components. The interface alone does not distinguish a component from an object, however. The real requirement for our components is that all interaction between components must be reflected in the interface. From this point of view, many current component-based approaches are not truly component-based, like CORBA, DCOM, and JavaBeans, because there, an object can directly call a function of another object. Although the function to be called exists in the callee's interface, it is not reflected in the caller's interface. Consequently, the interaction between objects is not fully captured by the interface. A more serious problem is that this kind of binding does not produce composable objects. The dependency is buried in the code of the caller object and would remain fixed unless the code can be modified.

Output ports are the solution to this problem. Input ports are equivalence of functions. They prescribe what functionalities a component can provide. Output ports, in contrast, prescribe what functionalities a component may require from other component. By delaying their connection until the configuration phase, the binding becomes more flexible. For instance, a component integrator may try to link the output port of a component with the input port of several different other components in order to select the best one.

For simulation modeling, there is another essential aspect that must be taken into consideration: the simulation time. This is where our simulation component classification can help. All components that we referred to so far are timeless components. Since we are only concerned with design of a single simulator, time-dependent components are sufficient for the modeling purpose. Messages exchanged between time-dependent components are timestamped by an implicit argument representing the simulation time at which the message occurs. To deal with the simulation time, we introduce a special entity called timer. Similar to input and output ports, timers are declared in the interface. However, components do not communicate with each other through timers. Rather, timers serve as a messenger between components and the simulation engine. A timer is actually a hybrid of an input and output port. By writing a timer with a value denoting a future time, the component is asking the simulation engine to schedule an event at the preset time. When the preset time is reached, the timer is activated, and the component must respond to the timer.

Similar component-port models have been proposed before (Ferenci, Perumalla, and Fujimoto 2001; Shanmugan and LaRue 1992). Actually, any component model that relies on ports as the only inter-component communication mechanism bears some resemblance with ours. We made two contributions, though, when introducing it as the basis for our component-oriented simulation worldview. First, we point out that the existence of the output ports is fundamental to a true component-based approach. Second, our simulation component classification clarifies the role played by the simulation time, and helps us develop a component-port model specifically for simulation. In the next section, we will describe the implementation of such a simulation component model.

3 IMPLEMENTATION OF COST

The first issue of implementing the aforementioned simulation component model is the choice of the implementing language. Discrete event simulators can be roughly divided into two groups: those based on a special simulation language, such as GPSS and SIMSCRIPT (Law and Kelton 1982), and those based on a general programming language, such as SIMPACK (Fishwick 1992) and SIMKIT (Gomes et al. 1995) Simulation languages contain abundant semantics designed for simulation, but requires a steep learning curve. General programming languages are more familiar to programmers, but lack the essential simulation constructs.

We chose C++ as the implementation language for two reasons. First, general programming languages always have good compiler support, and thus their execution speed is generally faster after optimization. Second, language-level reusability is a factor as important as component-level reusability, and C++ is one of the few languages that support code reuse well. With STL (Austern 1999; Musser and Saini 1996), C++ programs can easily achieve high efficiency while maintaining a high level of code reuse, which matches our design goal.

However, with C++ we ran into a problem. As mentioned in last section, input ports are equivalent to functions, so it is natural to define them as member functions of

the component. But how can we represent output ports? C++ language standard requires that the address of an object must be provided when the member function is being called. This conflicts with the requirement that component development should be completely independent. The classical solution for such a problem is a functor, which is the generalization of the function pointer.

3.1 Functor

A functor, or a function object, is an object "that can be called in the same way that a functions is" (Austern 1999; Musser and Saini 1996). A functor class overloads the operator *()* so that it appears as a function pointer. For instance, the following is declaration of a functor class that takes one function argument.

```
template <class T> class functor {
public:
 typedef funct_t bool (*f)(T );
 functor (funct_t _f): f(_f) {}
 virtual bool operator (T t) { return f(t); }
private:
 funct_t f;
};
```

The class *functor* is a helper class that wraps a function pointer of type *funct_t*. Upon invocation, it calls the actual function pointer and returns the result. The syntax of using the functor is exactly the same as that of a function pointer.

The same idea can be applied to member functions as well. In C++, a member function of a class always takes an implicit parameter *this*, which is a pointer to the object upon which the member function will be invoked. As a result, two member functions that belong to different classes but take the same explicit parameters are treated as functions of different types. In the component level, however, they should be viewed as interchangeable. A *mem_functor* declared below can hide the class type as well as the implicit parameter *this*.

```
template <class C, class T>
class mem_functor : public functor {
public:
 typedef funct_t
  bool (C::*f)(T);
 mem_functor (C* _c, funct_t _f)
  : c(_c), f(_f) {}
 virtual bool operator(T t){return c->f(t);}
private:
 C* c;
 funct_t f;
};
```

With these two classes, *functor* and *mem_functor*, it is now straightforward to implement input and output ports. An input port could be simply an instantiation of the *mem_functor* class. Since an output port does not know the component(s) to which it will be connected, it could be

represented as a pointer to a *functor*. When connecting an input port to an output port implemented in this way, the address of the *mem_functor* object corresponding to the input port is assigned to the functor pointer corresponding to the output port, because the class *mem_functor* is derived from the *functor* class. When the output port is invoked, the operator *()* of the *mem_functor* class is called, because it is declared as virtual.

3.2 Inport and Outport Class

The method of implementing input and output ports directly on top of two functor classes should work well, but there are some practical considerations. For instance, a port should have a name for the purpose of the debugging and a port must be set up properly before it can be used in order to initialize its member variables. Moreover, one to multiple connections would make topology generation more convenient. It is easy to connect an input port to multiple output ports by passing its address to each of them, but when connecting an output port to multiple input ports, the output port must store the addresses of all connected input ports. Those reasons are the main motivation for building the *inport* and *outport* class on top of functor classes.

The *outport* class is declared to be a class with a template parameter that is the type of the events that can be handled by the output port. The function *Setup()* gives the port a string name. The function *Write()* is invoked by the component to output a message. *ConnectTo()* connects an input port to the output port.

```
template <class T>
class outport {
public:
 void Setup(typeii* c, const char* name);
 bool Write(T t);
 void ConnectTo(inport& port);
private:
 std::vector<functor<T>*> inports;
};
```

Similarly, the *inport* class takes one template parameter that is the type of the function argument. It must be bound to a member function of a component, therefore the type of the component is passed as the template parameter of the member template function *Setup()*, as shown below.

```
template <class T>
class inport {
public:
 template <class C>
  void Setup( typeii* c,
   mem_functor<C,T>::funct_t _f,
   const char* name);
 bool Write(T t) { return (*f)(t); };
private:
 functor<T>* f;
};
```

Since the type of the member function bound to the input port must be passed to the *Setup()* function, we need to find a way to construct this type from two template parameters, C and T. Fortunately, this type is declared publicly in the class *mem_functor<C,T>* as *funct_t*.

3.3 Simulation Time and Port Index

Until now, functors in COST take only one function argument, which is the message exchanged between components. However, two more arguments are necessary. First, all the components in COST are time-dependent components, so messages should be timestamped. Hence, an extra argument is needed to denote the simulation time at which the message is generated. Another extra argument is for arrays of input ports, which are convenient if a number of input ports are of the same type. All elements in an input port array share the member function bound to them. Therefore, it is necessary to have an extra argument to distinguish between them by their indices. The index of an input port that is an element of an array is always zero. The resulting *functor* class could be like (other classes must be modified accordingly):

```
template <class T> class functor {
public:
 typedef funct_t bool (*f)(T,double,int );
 functor (funct_t _f): f(_f) {}
 virtual bool operator (T t, double time) {
  return f(t,time,index); }
private:
 funct_t f;
 int index;
};
```

3.4 Timer

The timer class requires two different functor classes, *t_functor* and *mt_functor*, because a time event has empty content, so the binding function of a timer only takes the timestamp argument and the index argument. A *timer* object is actually an array of timers, each of which is identified with a unique integer number, as in the input port arrays. The timer class has two methods: *Set()* to schedule an event and *Cancel()* to cancel an event.

```
class timer {
public:
 void Setup( typeii*,
  mt_functor<C>::funct_t, const char* name);
 void Set(double time, int index=0);
 void Cancel(int index=0);
private:
 t_functor * f;
};
```

So far, we have described techniques that we adopted to implement the component-port model in C++. It should be noted that all these implementation details are transparent to users. Users do not need to have advanced knowledge of C++ templates in order to write simulations in COST.

4 SIMULATION OF AN M/M/1 SYSTEM

To illustrate the modeling process with COST, we will describe in detail how to build an M/M/1 simulation. In an M/M/1 system, packets arrive according to a Poisson distribution and compete for the service in an FCFS (First-Come-First-Served) manner. The service time is also drawn from a Poisson distribution. In practice, M/M/1 systems are useful because many complex systems can be abstracted as composition of simple M/M/1 systems. M/M/1 systems also have an accurate mathematical solution with respect to the arrival rate and the service rate, which makes them well suited for validating simulation results.

An M/M/1 system built in COST is composed of three components, namely, source, FCFS server, and sink, as shown in Figure 1. Packets are generated by source, queued and served by FCFS server, and dispatched from sink.

Figure 1: An M/M/1 System

4.1 Data Type

A new data type, *Packet*, is defined to represent the packets that flow through the M/M/1 system. To measure the time spent in the *FCFS* component for each packet, a field *arrival_time* records the arrival time of a packet at the *FCFS* component.

```
struct Packet {
 double arrival_time;
};
```

4.2 Source

The source component creates packets at a rate specified by a given interval. It contains an output port of type *Packet* and a timer for scheduling the time to deliver the next packet. It is derived from the class *typeii*, the base class of all COST components.

```
class Source : public typeii {
public:
 double interval;
 outport < Packet > out;
 timer wait;

 void Setup(const char*);
 void Start();

private:
 bool Create(simtime_t, int index);
};
```

All COST components must provide a *Setup()* function in which the *Setup()* function of every port and timer must be called. The *Setup()* function of the base class *typeii* must be invoked first.

```
void Source::Setup(const char* name) {
 typeii::Setup(name);
 out.Setup(this,"out");
 wait.Setup(this, &Source::Create,"timer");
}
```

The *Start()* function, invoked the moment the simulation gets started, i.e., at the simulation time zero, is where a component can perform initialization of variables and schedule initial events using the *SetTimer()* method declared in the *typeii* class. *Exponential()* is another method declared by *typeii* to create a Poisson distribution.

```
void Source::Start() {
 m_seq_number=0;
 SetTimer(wait, Exponential(interval));
}
```

The *Create()* function is bound to the timer *wait*, so it is invoked every time the timer becomes activated. Its tasks include scheduling the event representing the next packet to be generated and delivering the current packet to the output port. Finally, it returns a true value. This is required for all member functions that are bound to input ports or timers. A true value indicates that the function has finished successfully.

```
bool Source::Create(simtime_t time,int) {
 SetTimer(wait, time+Exponential(interval) );
 Packet packet;
 packet.arrival_time = time;
 out.Write(packet,time);
 return true;
}
```

4.3 FCFS Server

The *FCFS* component is declared as a template class with a template parameter that has the type of packets that the FCFS server can hold. By instantiating it with different packet types, the FCFS component is capable of holding any packets. It could have been designed particularly for packets of type *Packet*, but that would prevent it from being used in a different simulation for any types other than *Packet*. This exemplifies the great benefit of using C++ template.

The FCFS component contains an input port and an output port, to receive and sent packets, as well as a timer to simulate the service of packets. A public member variable *service_time* specifies the average service time each packet will receive. There are three private member variables: *m_busy* reflects the status of the server; *m_queue* stores the packets waiting to be serviced; *in_service* is the packet that is currently being serviced.

```
template < class DATATYPE >
class FCFS : public typeii {
public:
 void Setup(const char*);
 void Start(){m_busy=false;}

 double service_time;
 inport < DATATYPE > in;
 ouport < DATATYPE > out;
 timer wait;
private:
 bool m_busy;
 std::deque<DATATYPE> m_queue;
 DATATYPE in_service;

 bool Arrive(const DATATYPE& packet,
   simtime_t, int index);
 bool Depart(simtime_t, int index);
};
```

Again, the *Setup()* function sets up every port and timer.

```
template < class DATATYPE >
void FCFS <DATATYPE>
 ::Setup(const char * name) {
 typeii::Setup(name);
 in.Setup(this,&FCFS<DATATYPE>::Arrive,"in");
 out.Setup(this,"out");
 wait.Setup(this,&FCFS<DATATYPE>::Depart,
   "next");
}
```

The *Arrive()* function is called when a packet arrives. Notice that packets are passed by reference to avoid variable copying overhead. The *const* keyword prevents the packet from being modified accidentally in the function.

The value of *m_busy* denotes whether or not the server is busy serving another packet. If it is not, the arriving packet is put into service and a service time is generated randomly. If it is, this packet is simply put into the queue.

```
template < class DATATYPE >
bool FCFS<DATATYPE>::Arrive(
 const DATATYPE& packet, simtime_t time,int){
 if (!m_busy){
  in_service=packet;
  SetTimer(wait,time +
   Exponential(service_time));
  m_busy=true;
 }
 else
  m_queue.push_back(packet);
 return true;
}
```

The *Depart()* function is called when the timer *wait* becomes activated. It outputs the current packet in service, and then checks if there are any other packets waiting in the queue.

```
template < class DATATYPE >
bool FCFS <DATATYPE> :: Depart(
 const trigger&, simtime_t time, int) {
 out.Write(in_service,time);
```

```
if (m_queue.size()>0){
 in_service=m_queue.front();
 m_queue.pop_front();
 SetTimer(wait,time +
  Exponential(service_time));
}
else
 m_busy=false;
return true;
}
```

4.4 Sink

In the Sink component, we collect the time that each packet spent in the FCFS server. It only has one input port and no timer. The two private member variables are used to record the cumulative delay time and the number of packets received, respectively. *Start()* is called when the simulation begins, and *Stop(),* in which we print out the result, is called when the simulation reaches the preset ending time.

```
class Sink : public typeii {
private:
 double m_total;
 int m_number;
public:
 inport< Packet > in;

 void Start(){
  m_total=0.0;
  m_number=0;
 }
 void Setup(const char* name) {
  typeii::Setup(name);
  in.Setup(this,&Sink::Arrive,"in");
 }
 void Stop(){
  printf("Average delay is: "
   "%f (%d packets) \n",
   m_total/m_number, m_number);
 }
private:
 bool Arrive(const Packet& packet,
  simtime_t time,int ) {
  m_total+=time-m_packet.arrival_time;
  m_number++;
  return true;
 }
};
```

4.5 Constructing the Simulation

The simulation class is derived from the *CostSystem* class. Components are instantiated as private member variables. Two public member variables are two simulation parameters that determine the arrival rate and the service rate.

```
class MM1 : public CostSystem {
public:
 void Setup(const char*);
 double interval;
 double service_time;
private:
 Source source;
 FCFS <Packet> server;
```

```
 Sink sink;
};
```

The simulation has a *Setup()* function too. It first maps component parameters to simulation parameters, and then invokes the *Setup()* function of every component. After that, it connects pairs of input and output ports. Finally, the *Setup()* function of the base class is invoked.

```
void MM1::Setup(const char*name) {
 source.interval=interval;
 server.service_time=service_time;
 source.Setup("source");
 server.Setup("server");
 sink.Setup("sink");
 Connect(source.out,server.in);
 Connect(server.out,sink.in);
 CostSystem::Setup(name);
}
```

4.6 Running the Simulation

To run the M/M/1 simulation, first we need to instantiate an M/M/1 simulation object, and then choose the parameters. *StopTime* is a default parameter indicating the ending time of the simulation. The *Setup()* function must be invoked prior to the simulation.

```
int main(int argc, char* argv[]) {
 MM1 mm1;
 mm1.interval=1;
 mm1.service_time=0.5;
 mm1.StopTime=1000000.0;
 mm1.Setup("mm1");
 mm1.Run();
 return 0;
}
```

4.7 Reusability in COST

COST has been used for other, far more complex, simulations , like queuing networks, computer networks and PCS simulations. These examples can be found at <http://www.cs.rpi.edu/~cheng3/cost>. It is targeted at the simulation modelers who have beginning or intermediate knowledge of the C++ language. Once they understand the basic component-port model and its support classes, it is fairly easy for them to write models with COST, and, more importantly, to take the component-based approach to model the system to be simulated.

Although some simulators, like CSIM (Schwetman 1986), may simulate the M/M/1 system in perhaps tens of lines of code, COST does not necessarily imply longer code. First, we can see that a large portion of the COST code is straightforward and suitable for code generation. Second, COST components are highly reusable. For instance, the FCFS component can process any types of packets. Even the Source and the Sink components can be modified with few changes into template classes to take any type of packets

with a field *arrival_time*. Once a component repository with a wide range of models is developed, the modeler will be able to construct a simulation just by connecting components obtained from the component.

5   SUMMARY

COST is a discrete event simulator written in C++ that embodies a component-oriented modeling style. At the heart of COST is a component-port model, which is distinguished from many developed component models by the notion of output ports. Our simulation component classification allows us to extend such a component-port model to make it well suited for discrete event simulation by introducing the implicit timestamp mechanism and timers.

The most distinct feature of COST is the component reusability. Components developed for one simulation can be effortlessly reused in other simulations. With an extensive set of library components, writing simulation in COST could be as simple as dragging a few components from the library and connecting them, as some commercial simulators do. The extra advantage of COST is that building components from scratch is simple.

The only inefficiency of COST simulations comes from the message exchange between components, which may involve several layers of function calls and a few virtual function table lookups. However, this is rather the deficiency of the C++ language, not of the underlying component-port model, because theoretically such overhead can be eliminated during the configuration phase. Had we had a truly component-oriented language, COST would have achieved perfect efficiency.

ACKNOWLEDGMENT

REFERENCES

Austern, M. H., 1999. *Generic Programming and the STL.* Addison-Wesley.

Chen, G., and Szymanski, B. K., 2001. Component-Based Simulation. *Proc. 2001 European Simulation Multi-Conference*, pp. 68-75. SCS Press.

Ferenci, S., Perumalla, K. S., and Fujimoto, R. M., 2000. An Approach for Federating Parallel Simulators. *Proc. 4th Workshop on Parallel and Distributed Simulation*, pp. 63-70.

Fishwick, P. A., 1992. SIMPACK: Getting Started with Simulation Programming in C and C++. *Proc. 1992 Winter Simulation Conference*, ed. J. J. Swain, D. Goldsman, R. C. Crain, and J. R. Wilson, pp. 154-162.

Gomes, F., Franks, S., Unger, B., Xiao, Z., Cleary, J., and Covington, A., 1995. SIMKIT: A High Performance Logical Process Simulation Class Library in C++. *Proc. 1995 Winter Simulation Conference*, ed. C. Alexopoulos, K. Kang, W. R. Lilegdon, and D. Goldsman, pp. 706-713.

Law, A. M., and Kelton, W. D., 1982. *Simulation Modeling and Analysis.* McGraw-Hill.

Musser, D. R., and Saini, A., 1996. *STL Tutorial and Reference Guide.* Addison-Wesley.

Schwetman, H., 1986. CSIM: A C-Based, Process-Oriented Simulation Language. *Proc. 1986 Winter Simulation Conference*, ed. J.Wilson, J. Henriksen, and S. Roberts, pp. 387-396.

Shanmugan, K. S., and LaRue, W., 1992. A Block-Oriented Network Simulator (BONeS). *Simulation*, 58:2, pp. 83-94.

AUTHOR BIOGRAPHIES

GILBERT CHEN is a graduate student at the Department of Computer Science, Rensselaer Polytechnic Institute. He received an MS and a BS in electronic engineering from Tsinghua University, P.R. China. His research interests include parallel discrete event simulation and component-based software development.

BOLESLAW K. SZYMANSKI is a Professor at the Department of Computer Science, Rensselaer Polytechnic Institute. He received his Ph.D. in Computer Science from National Academy of Sciences in Warsaw, Poland, in 1976. Dr. Szymanski is an IEEE Fellow and a member of the IEEE Computer Society, and the ACM for which he was a National Lecturer.