Proceedings of the 1996 Winter Simulation Conference
ed. J. M. Charnes, D. J. Morrice, D. T. Brunner, and J. J. Swain

A CORBA FACILITY FOR NETWORK SIMULATION

Chien-Chung Shen

Bellcore
331 Newman Springs Road
Red Bank, New Jersey 07701, U.S.A.

ABSTRACT

Contemporary software development environments for discrete-event simulation have adopted either a language-based approach or a library-based approach. Although these approaches have advantages such as type checking and optimized code generation provided by the former and familiar programming environments facilitated by the latter, they suffer from the inherent limitations of model portability and interoperability, which may result in inflexible models and higher model development cost. This paper proposes a CORBA discrete-event simulation facility to facilitate portable and interoperable simulation models. The proposed facility is defined by a CORBA IDL interface which defines operations for object definition, inter-object communication and event scheduling. Based on the given IDL interface definition, different vendors could supply different products (at different costs) by using different simulation algorithms, different programming language, or on different operating system or hardware platforms. With respect to the simulation models, they see a consistent interface across all products. The paper presents the simulation facility IDL interface, describes its prototype implementation in C++, and illustrates its usage by a bounded-buffer producer-consumer example.

1 INTRODUCTION

Contemporary software development environments for discrete-event simulation have adopted either a language-based approach or a library-based approach (Martin and Bagrodia 1995). In either case, programmers are provided with a set of model definition primitives together with a set of parallel programming primitives for object definition and inter-object communication and synchronization. These primitives are provided either as language extensions or as functions implemented as library routines. Although these approaches have advantages such as type checking and optimized code generation provided by the former and familiar programming environments facilitated by the latter, they suffer from the following two inherent limitations.

- **Portability.** Simulation models developed using one simulation language might not be easily ported to another simulation language. The programmers will be required to learn new language constructs and perhaps an entirely new set of program development tools.
- **Interoperability.** Components of a simulation model are required to be programmed in the same host language dictated by the simulation language or library routines used. It is, so far, infeasible that a simulation model consists of interoperable components written in different programming languages, or running on different operating systems or hardware platforms.

These limitations may potentially lock simulation application development into a particular (vendor) environment and result in inflexible models and higher model development cost.

However, these restrictions are not inherent in simulation application development alone. In fact, one of the most difficult tasks challenging information industry today is to enable application interworking and construction in a distributed, heterogeneous, and multi-vendor environment.

Confronted with the challenge, the Object Management Group (OMG) was founded in 1989 to develop a set of standards, using *object technology*, to facilitate distributed computing, which guarantee application interoperability and portability. Among them, the *Common Object Request Broker Architecture (CORBA)* (OMG 1993) has emerged as a *de facto* standard for distributed object computing.

At its core, CORBA defines the facilities required to allows a client object to *transparently* invoke the

services offered by CORBA-compliant objects (or CORBA objects for short) running on any machine in a heterogeneous distributed environment. The remote CORBA object are available across different operating systems (UNIX, Windows, OS/2, MVS), and different programming languages, such as C++, C, Ada, and Java, with many others to follow.

Each CORBA object has an *interface* that defines the services it offers to its clients, and this interface is defined in an Interface Definition Language (IDL) specified by the CORBA standard. The CORBA IDL is not a programming language and it does not replace the use of programming languages. Instead, the IDL's only role is to define interfaces which consist of operations available to the clients of the interfaces.

The advantage of using CORBA IDL is that it allows a CORBA object to define its interface (services) in a *declarative* fashion, which is independent of the programming language used to implement the object itself, or the programming language used to implement the clients of the object. In particular, the language used to implement CORBA objects need not be the same as that used by clients, and of course, the clients that invoke a given CORBA object need not all be implemented in the same programming language. For example, a client object written in Ada need not be aware that a CORBA object it is invoking is implemented in C++. In this case, the IDL definition of the object interface is translated, through an appropriate IDL compiler, automatically into Ada for the benefit of the client, and into C++ for the benefit of the implementer of the interface.

In this paper, we propose a CORBA discrete-event simulation facility using the message-based approach (Bagrodia, Chandy, and Misra 1987). The facility exports an interface, defined in CORBA IDL, consisting of operations for object definition, inter-object communication and event scheduling, and implements the interface using a discrete-event simulation algorithm. Simulation models utilizing the facility shall first *bind* to the facility (as an object) and then invoke the exported operations to obtain runtime support for executing simulation models.

Based on the proposed interface definition, different vendors are encouraged to supply different products (at different costs) by using different simulation algorithms, different programming language, or on different operating system or hardware platforms. However, with respect to the simulation models, they see a consistent interface across all products.

The remainder of the paper is organized as follows. We first present an overview of CORBA in the next section. The concept of message-based discrete-event simulation is introduced in Section 3. Based on that, we define a discrete-event simulation facility and present its CORBA IDL interface definition in Section 4. Section 5 describes its prototype implementation. Application of the facility using a bounded-buffer producer-consumer example is described in Section 6, and Section 7 is the conclusion.

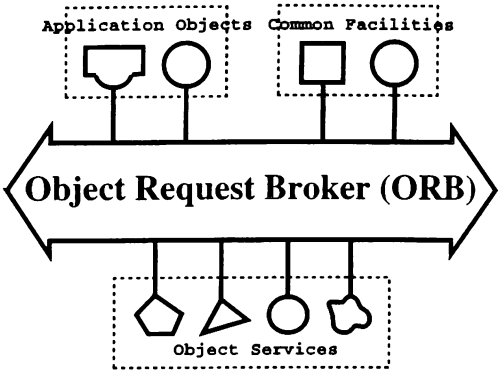

Figure 1: Object Management Architecture (OMA)

2 DISTRIBUTED OBJECT COMPUTING WITH CORBA

To enable distributed application interworking and construction, OMG defines the *Object Management Architecture (OMA)* Reference Model (OMG 1992) which identifies and characterizes the components, interfaces and protocols necessary to guarantee interoperability and portability of distributed applications in heterogeneous environments.

2.1 The OMA Reference Model

The OMA defines a reference model (Figure 1) for distributed object computing. Within the reference model, the most important component is the Object Request Broker (ORB). An ORB provides the basic mechanism for transparently making requests to and receiving responses from objects located locally or remotely without the client needing to be aware of the mechanisms used to communicate with the objects. As such, the ORB forms the foundation for building applications constructed from distributed objects and for interoperability between applications in distributed heterogeneous environments.

Every entity in the reference model is modeled as an object. These objects communicate with each other via the ORB. According to their functionality, objects are categorized into three groups:

- **Object Services** comprise a collection of services (interfaces and objects) that provide basic functions for using and implementing objects.

Examples are naming and event services.

- **Common Facilities** provide a collection of commonly-found services useful in many applications. In contrast to Object Services which are general purpose and application-domain independent, Common Facilities are services which are often application-domain specific and typically provide functionality directly to end-users. The simulation facility is an example.
- **Application Objects** are objects specific to particular end-user applications. Examples are simulation and banking applications.

2.2 The CORBA Architecture

The CORBA specifies a concrete description of the interfaces and services that must be provided by compliant Object Request Brokers. CORBA is composed of the following five major components.

- **ORB Core.** The ORB Core provides the basic communication capability between objects. It supports two different ways in which clients can make requests to objects: *static invocation* via interface-specific stubs and skeletons compiled from IDL interface definitions, and *dynamic invocations* via the Dynamic Invocation Interface. No matter which of these methods is used by a client to make a request, the ORB Core locates the desired object, activates it if it is not already executing, and delivers the request to it. The object performs the requested service and returns any result back to the ORB Core which then returns it to the client.
- **Interface Definition Language (IDL).** In CORBA, object services are defined as *interfaces* in IDL, a language-independent declarative language. IDL supports (multiple) inheritance in which derived interfaces inherit the operations and types defined in their base interfaces.
- **Interface Repository (IR).** The IR provides persistent storage for IDL interface definitions. Its primary function is to provide the interface information necessary to issue requests using Dynamic Invocation Interface.
- **Dynamic Invocation Interface (DII).** Some applications, such as browsers, require the capability of sending requests to objects without having compile-time knowledge of their interface definitions. The DII allows run-time discovery of interfaces from an IR, dynamic creation and invocation of requests to objects.

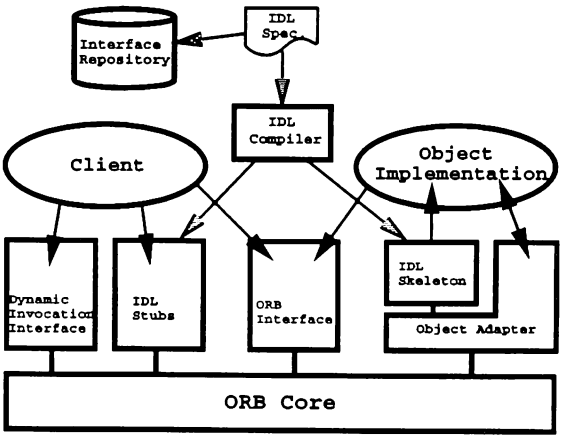

Figure 2: The CORBA Components

- **Object Adaptor (OA).** An object adaptor provides the means by which various types of object implementations can utilize ORB services, such as object method invocation, activation and deactivation of objects and implementations.

Figure 2 shows how the various CORBA components working together to facilitate distributed object computing. We assume that a client wants to invoke a service supported by an object. An IDL definition file is created to describe the interface (services) the object provides, which is stored in the IR as well as compiled into an IDL Stub and an IDL Skeleton. The client can initiate a request by calling the IDL Stub. Alternatively, the client looks up the IR and dynamically creates and invokes a request using DII. In either case, the request is directed to the ORB Core. The ORB Core locates the object implementation and then delivers the request to the OA managing that object. The OA feeds the request into the IDL Skeleton where it is then passed to the object implementation. Any return values are passed back through the IDL Skeleton and OA to the ORB Core. Then, depending on the original call, the ORB Core returns the value either through the IDL Stub or the DII to the client.

3 MESSAGE-BASED DISCRETE-EVENT SIMULATION

In this section, we review the basic concept of message-based discrete-event simulation which provides a more natural paradigm for simulating distributed systems, and therefore better serves as the foundation for the definition of a simulation facility.

In message-based simulation, each physical entity is abstracted by an *logical object* (*lo*), and interactions among the entities, called *events*, are represented by message communications among the corresponding

clock = 0;
Initialize *event-list*;
while (*execution not terminated*) **do**
{ fetch next tuple (m_i, s_i, d_i, t_i) from *event-list*;
if (m_i is a timeout message) **then** *clock* = t_i;
deliver m_i to d_i for simulation;
}

Figure 3: Message-Based Simulation Algorithm

lo. (Symbol *lo* represents both singular and plural forms.) A message-based simulation algorithm uses two data structures (Misra 1986): a *simulation clock* and an *event-list* (Figure 3). The simulation clock gives the time up to which the physical system has been simulated. The event-list is a partial order of tuples; a tuple is represented by (m, s, d, t), where m represents a message, s and d are the source and destination *lo* for m, and t is a timestamp. The partial order is typically based on the timestamp and ensures that events are simulated in the order of their dependencies. At every step of the simulation, the algorithm selects the tuple with the *smallest* timestamp, say (m_i, s_i, d_i, t_i), removes it from the event-list, and delivers m_i to d_i. Multiple tuples with the same timestamp may be handled in an arbitrary order, or be ordered deterministically using transparent sequence numbers to reflect their dependencies. The simulation of m_i by *lo* d_i may generate additional messages which are added to the event-list.

During the execution of a simulation program, the simulation clock advances in a *monotonic nondecreasing* manner through the timestamps associated with each tuple. Note that the simulation clock is completely decoupled from the physical processor clock. The physical time required to simulate a message does not have any effect on the simulation clock. How is the timestamp assigned to a message? When a message is generated, it is timestamped with the current value of the simulation clock — with one exception. We define a special message called a *timeout* message. The timeout message is scheduled by an *lo* for delivery to *itself* at a *future* time and is typically used to simulate the time of a simulation step that would be required by the physical entity to execute the corresponding operational step. An *operational step* refers to the statements executed by a physical entity to process a message received by it, and a *simulation step* models the activities that would be executed by the corresponding operational step. For example, consider a file-handler entity. On receiving a *read* request for the file, a physical (operational) file-handler will read the appropriate record from the file and return it to the requesting entity. If the file-handler is abstracted by a logical object, on receiving a *read* request, the *lo* estimates t, the time required for the corresponding physical entity to read the file and schedules a timeout message to itself t time units later. As the timestamp on all messages other than the timeout message refers to the current value of the simulation clock, the simulation time advances only when a timeout message is delivered to an *lo*.

4 SIMULATION FACILITY

In light of the above description, we define the `DESFacility` interface (Figure 4), in CORBA IDL, for message-based discrete-event simulation, which conists of operations for object definition, inter-object communication and event scheduling.

```
typedef string  ObjName;
typedef string  MsgType;
typedef any     MsgContents;
typedef long    Time;
typedef boolean BGuard;

struct WCItem {
  MsgType     msg_type;
  BGuard      bguard;
  Time        duration;
};
typedef sequence<WCItem> WaitCondition;

struct Message {
  ObjName     source;
  ObjName     sink;
  MsgType     msg_type;
  MsgContents contents;
};

interface DESFacility {
  void role(in ObjName myself);
  void enroll(in ObjName myself);
  void resign(in ObjName myself);
  void send(in Message msg);
  void receive(out Message msg,
               in  ObjName myself);
  void waituntil(out Message       msg,
                 in  ObjName       myself,
                 in  WaitCondition wc);
  void hold(in Time tm, in ObjName myself);
  Time now();
};
```

Figure 4: Interface Definition (DESFacility.idl)

Object Definition. Operations `role`, `enroll`, and `resign` are used to define participating simulation objects, and to demarcate their beginning and ending of execution.

Inter-Object Communication. Objects communicate with each other using buffered message-passing (asynchronous communication), where every object has a unique message-buffer. An object sends a message to another by invoking a `send` operation. Each message, with structure defined by `Message`, contains sender, receiver, message type, and message contents information. A message is deposited in the message-buffer of its destination object on the invocation of a `send` operation, and carries a timestamp which corresponds to the simulation time at which the corresponding `send` operation is invoked. An object accepts messages from its message buffer by invoking a `waituntil` operation. The `waituntil` operation takes a sequence of wait-condition items as an input argument to determine which message will be accepted by the object. Each wait-condition item specifies a message-type, say m_i and a boolean value, say b_i, which is said to be *enabled* if the message buffer contains a message of type m_i and b_i is true, and m_i is called an *enabling* message. The enabling message with the earliest timestamp is removed and delivered to the object. If all wait-condition items are disabled, the object is suspended for a *maximum* duration of *simulation* time equal to `duration` of a wait-condition item with `msg_type` equal to `timeout`. If omitted, a default wait-condition item with `msg_type` equal to `timeout` is set with an arbitrarily large `duration` value. If no enabling message is received in the `duration` interval, the object is sent a `timeout` message. An object must accept a `timeout` message that is sent to it. A separate `receive` operation is also provided to accept the message with the earliest timestamp from the message-buffer, regardless of its message type. It will block the invoking object until a message is available.

Event Scheduling. The `hold` operation enables the invoking object to schedule a timeout message for delivery to itself `tm` time units from now. It is used to specify the simulation time used by a simulation step. The `now` operation lets the invoking object to read the current value of the simulation clock.

5 PROTOTYPE IMPLEMENTATION

In this section, we describe a C++ implementation of the simulation facility in a multi-threaded version of the commercial ORB Orbix (IONA 1995). (Orbix is a Registered Trademark of IONA Technologies Ltd.)

We adopt the *wait-until* simulation algorithm

```
clock = 0;
Initialize event-list;
while (execution not terminated) do
{  fetch next tuple (m_i, s_i, d_i, t_i) from event-list;
   if (m_i is not accepted by d_i) then
      store m_i in temp_queue;
   else
   {  if (m_i is a timeout message) then clock = t_i;
      deliver m_i to d_i for simulation;
      merge temp_queue with event-list;
   }
}
```

Figure 5: Wait-Until Simulation Algorithm

(Franta 1977), as shown in Figure 5, to implement the simulation engine, in which an object may delay acceptance of a message based on its state such that messages are not necessarily delivered in the partial order specified by the event-list. Each *lo* may specify a wait-condition which restricts the types of message that it is willing to receive; a message is delivered to the destination *lo* only if it satisfies its wait-condition. The simulation clock is advanced through the timestamps associated with the timeout messages in the event-list.

In the C++ implementation, the `DESFacility` interface is implemented as the `DESFacility_i` C++ object (Figure 6) exporting the defined operations. Upon receiving an invocation for operation `send`, the `DESFacility_i` object inserts the message into the event list according to increasing timestamp order. On receiving an invocation for operations `waituntil` and `receive`, a thread is created implicitly to check if there is any enabling message. It will block the invoking object in case there is no enabling message. Finally, upon receiving an invocation for operation `hold`, a thread is created implicitly to first deposit a timeout message in the event list. It then block until the scheduled timeout message becomes deliverable.

As simulation facility is defined in CORBA IDL, it does not dictate its implementation. The simulation facility could very well be implemented using a parallel simulation algorithm to take advantage of parallel processing environment. With respect to its clients, the implementation is totally transparent, except for potential performance improvement.

6 AN EXAMPLE

As an example, we consider the simulation of a *bounded-buffer producer-consumer* system (Bagrodia and Shen 1991). The consumer object repeatedly re-

```
#include "DESFacility.hh"

struct Event {
  Message       msg;
  Time          tstamp;
  struct Event  *lst;
  struct Event  *nxt;
};

class DESFacility_i :
      public virtual DESFacilityBOAImpl
{
private:
  void enqueue(const struct Message& msg,
               Time tstamp);
public:
  Time          sim_clock;  // sim clock
  struct Event  eventList;  // event list
  DESFacility_i(char *marker); // constructor
  virtual ~DESFacility_i();    // destructor
  virtual void role(const ObjName myself);
  virtual void enroll(const ObjName myself);
  virtual void resign(const ObjName myself);
  virtual void send(const Message& msg);
  virtual void receive(Message& msg,
                       const ObjName myself);
  virtual void waituntil(Message& msg,
                      const ObjName myself,
                      const WaitCondition& wc)
  virtual void hold(Time tm,
                    const ObjName myself);
  virtual Time now();
};
```

Figure 6: DESFacility C++ Implementation Class

quests a data item from the buffer using a **request** message, waits to receive a **data** message, and invokes a **hold** operation to simulate data consumption. The producer object waits to receive a **free** message from the buffer which indicates that the buffer has a free slot, invokes a **hold** operation to simulate the generation of a data item, and sends the item to the buffer via a **data** message. The buffer object repeatedly invokes a **waituntil** operation that accepts a **request** message only when it is not empty and a **data** if it is not full.

Figures 7, 8, and 9 list the C++ source code for the producer, consumer, and buffer object, respectively, and Figure 10 depicts their interaction with the **DESFacility** object to obtain the discrete-event simulation service.

```
main (int argc, char **argv)
{
  DESFacility  *p;
  Message      msg_s, msg_r;
  int i, n, now, t_wait;
  p = DESFacility::_bind("sim:simSrv", "");
  p->enroll("producer");
  now = t_wait = 0;  n = Q_LENGTH;
  msg_s.source   = new char[20];
  strcpy(msg_s.source, "producer");
  msg_s.sink     = new char[20];
  strcpy(msg_s.sink,   "buffer");
  msg_s.msg_type = new char[20];
  strcpy(msg_s.msg_type, "data");
  for (i = 0; i < MAX_ITEM; i++) {
    if (n == 0) {
      now = p->now();
      p->receive(msg_r, "producer");
      n++;  t_wait += p->now() - now;
    }
    p->hold(rand()%MAXPRODUCE, "producer");
    n--;  p->send(msg_s);
  }
  now = p->now();
  printf("Producer utilization == %f\n",
         ((float)(now - t_wait)/(float) now));
  p->resign("producer");  p->_release();
}
```

Figure 7: The Producer Code

7 CONCLUSION

At the time when the computing industry is demanding portable and interoperable distributed applications, of which simulation is a notable example, CORBA has emerged as a *de facto* standard for distributed object computing. A CORBA-compliant object request broker serves as a *software bus* to facilitate communication among distributed objects, independently of their languages, operating systems, or hardware platforms. In addition, the OMA has depicted a reference model where application-specific services are provided as Common Facilities.

The paper describes a CORBA discrete-event simulation facility. The facility exports an interface, defined in CORBA IDL, consisting of model definition operations, and implements the interface using a discrete-event simulation algorithm.

The main thrust of the paper is not to propose a new simulation language or a more efficient parallel simulation algorithm, but to introduce a new paradigm for facilitating discrete-event simulation us-

```
main (int argc, char **argv)
{
  DESFacility *c;
  Message     msg_s, msg_r;
  int i, t, t_used, now;
  c = DESFacility::_bind("sim:simSrv", "");
  c->enroll("consumer");
  now = t_used = 0;
  msg_s.source   = new char[20];
  strcpy(msg_s.source, "consumer");
  msg_s.sink     = new char[20];
  strcpy(msg_s.source, "buffer");
  msg_s.msg_type = new char[20];
  strcpy(msg_s.source, "request");
  for (i = 0; i < MAX_ITEM; i++) {
    c->send(msg_s);
    c->receive(msg_r, "consumer");
    t_used += (t = rand() % MAXCONSUME);
    c->hold(t, "consumer");
  }
  now = c->now();
  printf("Consumer utilization == %f\n",
         ((float)(t_used) / (float)now));
  c->resign("consumer");  c->_release();
}
```

Figure 8: The Consumer Code

ing the newly emerged CORBA technology and to motivate more research activities and product development in this direction.

ACKNOWLEDGMENTS

This research was funded by the Army Research Laboratory under Cooperative Agreement No. DAAL01-96-2-0002."

REFERENCES

Bagrodia, R. L., K. M. Chandy, and J. Misra. 1987. A message-based approach to discrete-event simulation. *IEEE Transactions on Software Engineering* 13(6):654–665.

Bagrodia, R. L. and C.-C. Shen. 1991. MIDAS: Integrated design and simulation of distributed systems. *IEEE Transactions on Software Engineering* 17(10):1042–1058.

Franta, W. R. 1977. *The process view of simulation.* New York: Elsevier North-Holland Inc.

IONA. 1995. *Orbix: Programmer's Guide.* IONA Technologies Ltd.

Martin, J. M. and R. L. Bagrodia. 1995. COMPOSE: An object-oriented environment for parallel discrete-event simulation. In *Proceedings of the 1995 Winter Simulation Conference.* 163–166, Washington, DC.

Misra, J. 1986. Distributed discrete-event simulation. *Computing Survey* 18(1):39–65.

OMG. 1992. *Object Management Architecture Guide.* Object Management Group and X/Open.

OMG. 1993. *The Common Object Request Broker: Architecture and Specification.* Object Management Group and X/Open.

AUTHOR BIOGRAPHY

CHIEN-CHUNG SHEN is a Member of Technical Staff in Bellcore Applied Research. He received B.S. and M.S. degrees in computer science from National Chiao Tung University, Taiwan, in 1982 and 1984, respectively, and a Ph.D. degree in computer science from UCLA in 1992. His research interests include distributed network control and management for SONET, ATM, WDM and wireless networks, distributed object computing, and network simulation.

```
main (int argc, char **argv)
{
  DESFacility *b;
  Message     msg_s_p, msg_s_c, msg_r;
  WaitCondition wc;
  int q, i, v, tstart, tend;
  b = DESFacility::_bind("sim:simSrv", "");
  b->enroll("buffer");  tstart = b->now();
  msg_s_p.source   = new char[20];
  strcpy(msg_s_p.source, "buffer");
  msg_s_p.sink     = new char[20];
  strcpy(msg_s_p.sink, "producer");
  msg_s_p.msg_type = new char[20];
  strcpy(msg_s_p.msg_type, "free");
  msg_s_c.source   = new char[20];
  strcpy(msg_s_c.source, "buffer");
  msg_s_c.sink     = new char[20];
  strcpy(msg_s_c.sink, "consumer");
  msg_s_c.msg_type = new char[20];
  strcpy(msg_s_c.msg_type, "data");
  wc._length = 2;  wc._maximum = 2;
  wc._buffer = new WCItem[2];
  wc._buffer[0].msg_type = new char[20];
  strcpy(wc._buffer[0].msg_type, "data");
  wc._buffer[1].msg_type = new char[20];
  strcpy(wc._buffer[1].msg_type, "request");
  for (i = q = v = 0;;) {
    wc._buffer[0].bguard = (q < Q_LENGTH);
    wc._buffer[1].bguard = (q > 0);
    b->waituntil(msg_r, "buffer", wc);
    if (strcmp(msg_r.msg_type, "data") == 0) {
      tend = b->now();  v += q*(tend-tstart);
      tstart = tend;  q++;  break;
    } else {          // "request"
      b->send(msg_s_c);  tend = b->now();
      v += q*(tend-tstart); tstart=tend; q--;
      if ((++i) < MAX_ITEM)
        b->send(msg_s_p);
      else
        break;
    }
  }
  printf("The avg # of items in buf: %f\n",
         (float) v / (float) b->now());
  b->resign("buffer");  b->_release();
}
```

Figure 9: The Buffer Code

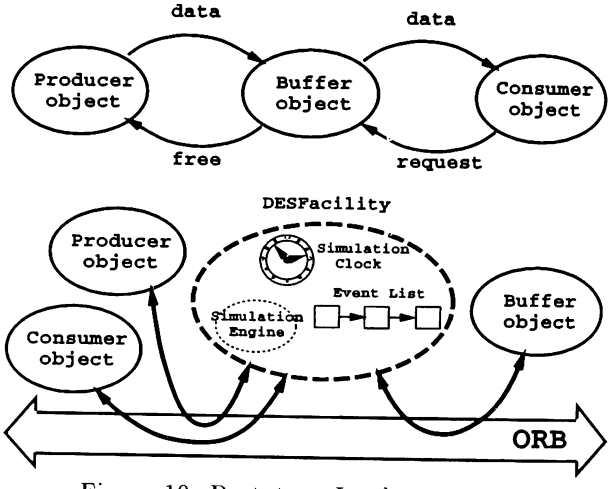

Figure 10: Prototype Implementation