Proceedings of the 1991 Winter Simulation Conference
Barry L. Nelson, W. David Kelton, Gordon M. Clark (eds.)

ITERATIVE DESIGN OF EFFICIENT SIMULATIONS USING MAISIE*

Rajive L. Bagrodia

Computer Science Department
University of California at Los Angeles
Los Angeles, CA 90024

ABSTRACT

Maisie is a C-based simulation language that encourages iterative design of efficient simulations using step-wise refinements. Maisie is among the few languages that cleanly separates the simulation program from the underlying algorithm (sequential or parallel) that is used to execute the program. It is thus possible to design a sequential simulation and, if needed, to subsequently port it to a parallel machine with few modifications. This paper gives a brief introduction to modeling and simulation using Maisie. It develops a complete Maisie model of a queuing network and illustrates both sequential and parallel implementations of the model.

1 INTRODUCTION

Simulation is an inherently iterative activity: the level of detail desired in the model of a physical system is rarely known a priori. A prototype model serves to identify the critical parts of the system that need further elaboration and detailed analysis. This paper describes a simulation language called Maisie that encourages a programmer to develop modular simulations using step-wise refinement, where the refinements progressively transform a prototype to an efficient (sequential or parallel) implementation. The design of a Maisie simulation is separated into two parts:

- design of a prototype simulation which models the physical system at the appropriate level of detail
- efficient implementation of the simulation on a sequential or parallel architecture.

*This research was partially supported by NSF (CCR 88 10376) and by Hughes Aircraft Co and State of California MICRO project

The purpose of the initial prototype is to ensure that the simulation program is an appropriate model of the physical system. In the initial stage, the emphasis is on rapid model design, rather than its efficient execution. Maisie constructs allow events and their enabling conditions to be specified at a high level of abstraction. Further, many simulations may be described graphically using an interactive icon-based model definition facility (Golubchik et al, 1991). This allows an analyst to explore a variety of alternative representations with minimal effort. After defining an appropriate model, it may possibly be refined to improve its efficiency. Simple monitoring facilities are (transparently) attached to the program to allow the analyst to identify the set of most frequently executed events. If desired, the enabling conditions and the actions associated with these events may be elaborated in terms of other Maisie constructs to improve efficiency. In the initial stage, the program is executed using a sequential simulation algorithm. Once the sequential model has been validated the program is progressively refined such that the enabling conditions are expressed using more efficient implementations. If the completion time of the sequential implementation is not acceptable, parallel implementations may be explored.

Maisie maintains a clear separation between the simulation program and the specific algorithm that is used to execute the program on a sequential or parallel architecture. With minor modifications, Maisie programs may be executed using a sequential algorithm or a variety of parallel algorithms. To execute the program on a parallel architecture, the initial refinements to the sequential program simply allocate Maisie processes among the available processors. In particular, at this stage the analyst need not be concerned with the specific simulation algorithm that is used to execute the program on the parallel architecture. A parallel Maisie program may, in general, be executed using a variety of simulation algorithms in-

cluding conservative algorithms (Misra 1986) or optimistic algorithms (Jefferson 1985, Chandy and Sherman, 1989). The final refinements to the program are dictated by the specifics of a particular simulation algorithm that is to be used. If an optimistic algorithm is used, these refinements can be targeted to reduce either state saving or recomputation overheads for the program. In contrast, if a conservative algorithm is to be used, the optimizations could reduce the synchronization overheads. The goal at this stage is to exploit the specifics of the application and the simulation algorithm to generate an efficient implementation. Note that the availability of an equivalent sequential implementation permits consistent comparisons of the relative efficiency of the sequential and parallel implementations of a given application.

In the remainder of this paper, we give a brief description of the language and use a simple queuing network to illustrate its use in designing efficient simulations.

2 MAISIE SIMULATION LANGUAGE

Maisie (Bagrodia and Liao 1990a, 1990b) enhances C with a few primitives to model objects and their interactions. Maisie uses an entity-type to model objects of a given type. An entity-instance, henceforth referred to simply as an entity, represents a specific object and may be created and destroyed dynamically. An entity is created by the execution of a **new** statement and is automatically assigned a unique identifier on creation. An entity can reference its own identifier using the keyword **self**. Maisie also defines a type called **e_name** which is used to store entity-identifiers. Figure 1 presents an entity called *driver* that contains a new statement (line 4). Execution of this statement creates an instance of entity *manager* and stores its identifier in variable *r*1. Every Maisie program must have a *driver* entity. This entity initiates execution of the simulation program and serves essentially the same purpose as the main function in C.

Entities communicate with each other using buffered message-passing. Maisie defines a type called **message**, which is used to define the types of messages that may be received by an entity. Definition of a message-type is similar to a struct; in figure 1, the *manager* entity type defines a message-type called *req* with two parameters (or fields) called *count* and *hisid* respectively (line 14). An entity sends a message to another by executing an **invoke** statement. Every entity is associated with a unique message-buffer. Execution of an invoke statement deposits a message in the message-buffer of the named entity. For instance, the invoke statement in line 6 will deposit a *req* message in the message-buffer of entity *r*1. A message is deposited in the destination buffer at the same simulation time as it is sent. If required, an appropriate hold statements (described subsequently) may be executed to model message transmission times or a separate entity may be defined to simulate the transmission medium.

An entity accepts messages from its message-buffer by executing a **wait** statement. The wait statement has two components: an optional wait-time (t_c) and a required resume-block. If t_c is omitted, it is set to an arbitrarily large value. The resume-block is a set of resume statements, each of which has the following form:

mtyp(m_i) [**st** b_i] *statement*$_i$;

where m_i is a message-type, b_i an optional boolean expression referred to as a *guard*, and *statement*$_i$ is any C or Maisie statement. The guard is a side-effect free boolean expression that may refer to local variables or message parameters. If omitted, the guard is assumed to be the constant *true*. The message-type and guard are together referred to as a *resume condition*. A resume condition with message-type m_i and guard b_i is said to be *enabled* if the message buffer contains a message of type m_i, which if delivered to the entity would cause b_i to evaluate to *true*; the corresponding message is called an *enabling message*. A resume condition that is not enabled is said to be *disabled*.

With the wait-time omitted, the wait statement is essentially a selective receive command that allows an entity to accept a particular message only when it is ready to process the message. For instance, the wait statement in line 17 of the *manager* entity consists of two resume statements. The resume condition (line 18) in the first statement ensures that the entity accepts a *req* message only if the requested number of units are currently available (the requests are serviced in first-fit manner). The statement associated with this resume condition (line 19-20) simply allocates the desired resources to the requesting entity. Keyword **msg** is used to refer to the last message that was removed from its buffer and delivered to the entity. The second resume statement (line 20) accepts a *free* message and the associated action adds the returned units to the available resource pool. In general the resume condition in a wait statement may include multiple message-types, each with its own boolean expression. This allows many complex enabling conditions to be expressed directly, without requiring the programmer to describe the buffering explicitly.

If two or more resume conditions in a wait statement are enabled, the timestamps on the correspond-

ing enabling messages are compared and the message with the earliest timestamp is removed and delivered to the entity. If all resume conditions in the wait statement are disabled, a timeout message is scheduled for the entity t_c time units in the future. The timeout message is canceled if the entity receives an enabling message *prior* to expiration of t_c; otherwise, the timeout message is sent to the entity on expiration of interval t_c.

A **hold** statement is provided to unconditionally delay an entity for a specified simulation time. The hold statement in line 8 will suspend the *driver* entity for t units in simulation time.

```
1  entity driver{}
2  { e_name r1;
3    message done;
4    r1 = new manager{};
5    ...
6    invoke r1 with req{self,4 };
7    wait until mtyp(done)
8      hold(t);
9    ...
10 }

11 entity manager{}
12 {
13   int units = MAX_UNITS;
14   message req {e_name hisid; int count; };
15   message free {int count; };
16   for (;;)
17     wait until
18     { mtyp(req) st (units >= msg.req.count)
19         { units = units-msg.req.count;
20           invoke msg.req.hisid with done;}
21       or  mtyp(free)
22         units = units+msg.free.count;}
23 }
```

Figure 1: A Resource Manager

3 EXAMPLE

In this section we develop a Maisie model for a simple queuing network and subsequently refine it for parallel execution. We also provide completion time measurements for the sequential and parallel implementations of the network.

Consider a closed queueing network (henceforth referred to as CQNF) that consists of N fully connected switches. Each switch is a tandem queue of Q fifo servers. A job that arrives at a queue is served sequentially by the Q servers and is thereafter routed to one of the N neighboring switches (including itself) with equal probability. The service time of a job at

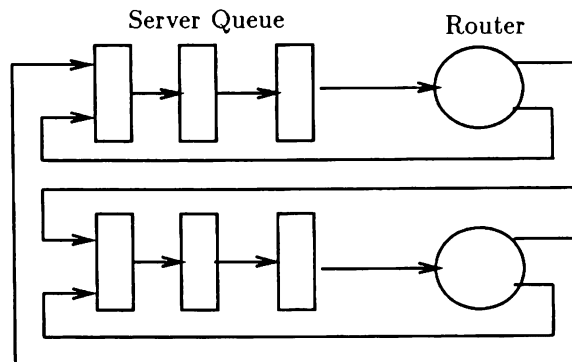

Figure 2: Model of a 2 Switch CQNF network

a server is generated from a negative exponential distribution, where all servers are assumed to have an identical mean service time. Each switch is initially assigned J jobs that make a predetermined number of trips through the network.

The Maisie model of this network consists of two primary entities: a *queue* entity to model the tandem servers in a queue and a *router* entity that routes a job after it has completed service at a queue. Figure 2 displays the model of a network with N=2 and Q=3. Each job in the network may be modeled as a separate entity or be abstracted by a sequence of messages. We adopt the latter approach. The complete Maisie program for this example is in figure 3. The driver entity is responsible for creating the *queue* and *router* entities. As the *queue* and *router* entities communicate with each other, each must have the entity-identifier for the other. Rather than use global variables for this purpose, the appropriate id is passed to the entity as either an entity parameter (as when creating the *router* entities in line 8) or in a separate message (as for the *queue* entities in line 11). The driver entity also instantiates a statistics collection entity (*basic_stats*) from the Maisie library (line 4). This entity is used to compute the average system time spent by a job in a queue.

A Maisie program can terminate in one of two ways: when the simulation clock exceeds some specified value or when the event-list is empty. In either case, when the simulation has terminated, a special message called *simdone* is sent to each entity. The driver entity is programmed such that on receipt of this message, it will send a *dump* message to the statistics collection agency which causes it to print its report.

We first consider the *queue* entity (lines 30-52) that simulates service of incoming jobs at each of its servers. Array *lastj* tracks the time at which the last job serviced at the queue departed from each server. The service time for a job at the ith server is generated

from an exponential distribution and used to update *lastj*[*i*] (line 45). When the job has been serviced at each server, its trip count and service completion time are incremented and it is forwarded to its *router* entity(line 49). Also, the total time spent by the job in the queue is sent to the statistics collection entity (line 50).

The jobs initially allocated to each switch of the physical network are allocated to the corresponding *router*. On being created, a *router* entity distributes these jobs among the various *queue* entities (line 21). Subsequently, for each incoming job, if the incoming job has not completed its required number of trips, the *router* entity generates a future message that simulates arrival of the job at the next switch in the network: it delays the job appropriately by executing a hold statement in line 26 (function sclock() returns the current value of the simulation clock), and then forwards the job to one of the N *queue* entities (line 27). Note that if the incoming job has completed the required number of trips, no additional messages are generated or scheduled. This implies that the event-list becomes empty when each job in the system has completed the required number of trips.

The preceding Maisie program was executed on a Sun Sparcstation for a network of 16 switches. The configuration information together with the computed statistics are shown in figure 4.

The program was subsequently refined for a parallel implementation where each *queue* and its corresponding *router* entity execute on a separate processor. Except for the driver entity, the remainder of the program remains unchanged. The driver entity must be changed to specify remote creation of entities. As seen from figure 5, the only change in the entity is to extend each new statement with the **at** clause (lines 6 and 8) to indicate the processor number on which the corresponding entity is to be created and executed. To execute the program on a parallel architecture, the mayc command must specify the architecture and number of nodes as follows:

```
% mayc cqnf.may -arch s2010 -nodes 16
```

The preceding command specifies that the Maisie program in file *cqnf.may* be executed on 16 nodes of the Symult S2010 multicomputer. The program is executed transparently using an optimistic simulation algorithm. Figure 6 shows the speedup that was obtained with the parallel version as the number of switches in the network is increased from 1 to 8. The sequential version used for the comparison was executed on a single node of the same machine using a sequential simulation algorithm.

```
#include "cmay.h"
#define N 3
extern entity basic_stats{};

1  entity driver{}
2  {  e_name rtr[N],q[N],stat1;
3     int i;
4     stat1= new basic_stats{"Average System Time"};
5     for (i=0;i<N;i++)
6       q[i] = new queue{5,1000,stat1};
7     for (i=0;i<N;i++)
8       rtr[i] = new router{10,10,q};
9     for (i=0;i<N;i++)
11      invoke q[i] with idmsg{rtr[i]};
12    wait until mtyp(simdone)
13       invoke stat1 with dump;
14 }

15 entity router{njobs,mtrips,qids}
16  int njobs, mtrips;
17  e_name qids[N];
18 { message job{int stime; int count;} j1;
19   int i;
20   for (i=0;i<njobs;i++)
21      invoke qids[i%N] with job{0,0};
22   for (;;)
23    wait until mtyp(job) {
24     j1=msg.job;
25     if (j1.count < mtrips) {
26      hold (j1.stime-sclock());
27      invoke qids[iurand(0,N-1)] with job=j1;}
28   }
29 }

30 entity queue{nsrvr, mtime, statid}
31  int nsrvr,mtime; e_name statid;
32 { int i,t1,lastj[nsrvr];
33   e_name rid;
34   message job{int stime; int count;} j1;
35   message idmsg{ e_name id;};
36
37   wait until mtyp(idmsg) rid= msg.idmsg.id;
38   for (i=0;i<nsrvr;i++)
39    lastj[i]=0;
40   for (;;)
41    wait until mtyp(job) {
42     j1=msg.job;
43     t1=j1.stime;
44     for (i=0;i<nsrvr;i++) {
45      lastj[i]=MAX(t1,lastj[i]) + expon(mtime);
46      t1=lastj[i];
47     }
49     invoke rid with job{t1,j1.count+1};
50     invoke statid with value{(t1-j1.stime)};
51     }
52 }
```

Figure 3: Maisie model of CQNF

CQNF Configuration simulated:
No. of switches = 16
Initial no. of jobs/switch = 16
No. of servers/queue = 10
No. of trips/job = 10

Statistics Collected: Average System Time
Total number of values 2560
Mean value 24988.93
Maximum value 52407.00
Minimum value 7160.00

Figure 4: Sample Run of CQNF

```
1  entity driver{}
2  { e_name rtr[N],q[N],stat1;
3     int i;
4     stat1= new basic_stats{"System Time"};
5     for (i=0;i<N;i++)
6       q[i] = new queue{5,1000,stat1} at i;
7     for (i=0;i<N;i++)
8       rtr[i] = new router{10,10,q} at i;
9     for (i=0;i<N;i++)
10       invoke q[i] with idmsg{rtr[i]};
11    wait until mtyp(simdone)
12        invoke stat1 with dump;
13 }
```

Figure 5: Parallel CQNF: Driver Entity

4 SUMMARY

The Maisie simulation language was designed by enhancing C with a few primitives to create entities and to model events. An important construct of Maisie is the wait statement which allows a programmer to directly specify an event and its enabling condition. Appropriate use of the wait statement leads to succinct programs and reduces program development time. Maisie models may also be constructed visually using an interactive front-end. The sequential implementation provides an interactive trace facility to facilitate program debugging. An event monitoring facility is also provided to allow an analyst to identify the compute-intensive portions of the model. This paper provided a quick introduction to common language features. Readers are referred to the Reference manual for a complete description.

Maisie programs may be executed on any machine (including laptop computers) that supports C. With minor modifications, a Maisie program may also be executed on parallel multicomputer architectures running UNIX-like operating systems as also on networks of (heterogeneous) workstations running UNIX. Performance studies on the efficiency of parallel Maisie implementations in the simulation of deterministic and stochastic systems may be found in Bagrodia and Liao (1990c) and Bagrodia, Chandy and Liao (1991).

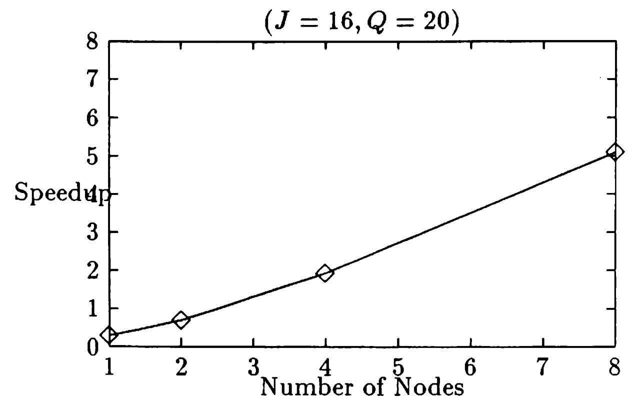

Figure 6: Speedup for CQNF

REFERENCES

R. Bagrodia and Wen-toh Liao. 1990a. *Maisie User Manual.* Computer Science Department, UCLA, Los Angeles, CA 90024.

R.L. Bagrodia and Wen-toh Liao. 1990b. Maisie: A language and optimizing environment for distributed simulation. In *1990 Simulation Multiconference: Distributed Simulation*, Eds. D.Nicols and R.Fujimoto, San Diego, California.

R.L. Bagrodia and Wen-toh Liao. 1990c. Parallel simulation of the sharks world problem. In *1990 Winter Simulation Conference*, Eds. O.Balci, R.Sandowski and R.Nance, New Orleans.

R. Bagrodia, K.M. Chandy, and Wen-toh Liao. 1991. An Experimental Study on the Performance of the Space-Time Algorithm *Preprint* Computer Science Department, UCLA, Los Angeles, CA.

K.M. Chandy and R. Sherman. 1989. Space-time and simulation. In *1989 Simulation Multiconference: Distributed Simulation*, Eds. B.Unger and R.Fujimoto, Miami, Florida.

L.Golubchik, G.Rozenblat,W.Cheng and R.Muntz. 1991. The Tangram Modeling Environment in *Modeling Techniques and Tools for Computer Performance Evaluation*, 421-435. Torino, Italy.

D. Jefferson. 1985. Virtual time. *ACM TOPLAS*, 7(3):404–425.

J. Misra. 1986 Distributed discrete-event simulation. *Computing Surveys*, 18(1).

AUTHOR BIOGRAPHY

Rajive L. Bagrodia is an Assistant Professor in the Computer Science Department at UCLA. His research interests include parallel languages, distributed simulation, distributed algorithms and software design methodologies. He was selected as a 1991 Presidential Young Investigator by NSF.