Computer Science Department Technical Report
University of California
Los Angeles, CA 90024-1596

A LANGUATE FOR ITERATIVE DESIGN OF EFFICIENT SIMULATIONS

R. Bagrodia
W. Liao

March 1992
CSD-920005

A Language for Iterative Design of Efficient Simulations*

Rajive L. Bagrodia
Wen-Toh Liao

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 refinement. It is among the few languages that separate the model from the underlying simulation algorithm (sequential or parallel) that is used to execute the model. With few modifications, a Maisie program may be executed using a sequential simulation algorithm, a parallel conservative algorithm or a parallel optimistic algorithm. The Maisie design also allows the runtime system to transparently reduce recomputation and state saving overheads for optimistic simulations as also synchronization overheads for conservative implementations. This paper presents the Maisie simulation language and describes how efficient parallel simulations may be developed as refinements of sequential models.

1 Introduction

Distributed (or parallel) simulation refers to the execution of a simulation program on parallel computers. A number of algorithms[Mis86, CS89a, CS89b, Rey82, Jef85, GL90] have been suggested for distributed simulation and many experimental studies have been conducted to evaluate the speedups that may be obtained from these algorithms and their variants. Experience with parallel simulators suggests that the reduction in the completion time of a simulation depends significantly on the application as well as the specific algorithm used to execute the model on a parallel architecture. For some applications, multiple independent replications of a sequential model may be more efficient than parallel execution of the model. In the absence of a priori knowledge about the suitability of a specific simulation algorithm to a given application, it is desirable to develop languages that separate the model from the underlying algorithm. Such notations would allow the analyst to develop a model and subsequently select the most suitable algorithm — sequential, conservative or optimistic — for its execution.

This paper describes a simulation language called Maisie. The Maisie constructs encourage 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.

*This research was partially supported under NSF PYI award ASC 9157610 and ONR grant N00014-91-J-1605

- efficient implementation of the simulation on a sequential or parallel architecture.

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. Many queuing network models may be described graphically using an interactive icon-based model definition facility. After defining and validating the initial 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 events whose implementation efficiency may be improved. If necessary, the enabling conditions for these events may be elaborated in terms of other Maisie constructs to improve efficiency. At the initial stage, the program is executed using a sequential simulation algorithm. If the completion time of the sequential program is not acceptable, parallel implementations may be explored.

Maisie maintains a clear separation between the simulation program and the specific algorithm used to execute the program on a sequential or parallel architecture. 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 including null-message based techniques, conditional event based algorithms, or the optimistic space-time algorithm. 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 state saving and 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 model.

In addition to the user-directed optimizations, the Maisie design also permits transparent optimizations of rollback overheads for optimistic simulations. In a subsequent section, we introduce the concept of an *artificial rollback* and describe how these can be detected transparently by the Maisie runtime system to reduce overheads for optimistic algorithms. We also indicate how the Maisie constructs allows *lookahead* characteristics of some applications to be extracted transparently from the program to reduce the synchronization overheads for conservative algorithms.

The rest of the paper is organized as follows: Section 2 discusses related work. Section 3 describes the primary constructs of the Maisie simulation language. Section 4 indicates how Maisie programs may be executed transparently using three different parallel simulation algorithms. Section 5 is devoted to reducing the overheads with optimistic implementations and section 6 discuss optimizations for conservative implementations. Section 7 addresses implementation issues and section 8 is the conclusion.

2 Related Work

A large number of sequential simulation languages have been designed including Simula, GASP, GPSS, Simscript, MAY, CSIM and many others. In contrast, parallel simulation languages (PSL) is a relatively new area of research. PSLs typically adopt one of two approaches: (a) enhance sequential simulation languages with primitives or library functions to specify parallel execution; examples include Yaddes[Pre89], Maisie[BL90a], Modsim[WM88] and Sim++[BLU90] among others, and (b) add simulation constructs to existing parallel languages as typified by Ada-based simulation environments like SCE[GMRR89].

The goal of Maisie was to design a simulation language that could be used to develop efficient sequential and parallel simulations. It is among the few languages that support both conservative and optimistic algorithms for its parallel implementations. The specific simulation constructs provided by Maisie are similar to those provided by other process-oriented simulation languages. For example, Sim++, a C++ based language that supports sequential simulations and Time-Warp simulations, has comparable constructs. However, Maisie extends these constructs to allow the enabling condition for an event to be described in a more abstract manner than possible in most contemporary simulation languages. To the best of our knowledge, Maisie is the first language that provides user-transparent or programmer-specified facilities to reduce the synchronization overheads for parallel implementations using either conservative or optimistic simulation techniques.

Other languages/systems that support multiple parallel simulation algorithms include OLPS [Abr88], Yaddes[Pre89], SPECTRUM[Rey88], and SPEEDES[Ste91]. Yaddes is a specification language for event-driven simulation that resembles Yacc and Lex. A Yaddes program is translated into a C program which is later compiled and linked with the runtime support library. Different simulation environments are provided by specifying the appropriate runtime library at link time. The system supports sequential, conservative, and optimistic simulations. Yaddes requires that the configuration of logical processes in the model and its mapping on a parallel architecture be specified completely at compile time.

OLPS provides a C++ object library to support sequential, conservative and optimistic simulation. It uses YACC grammar to specify the simulated physical system and, like Yaddes, also requires that the configuration of logical processes be specified statically. OLPS is not algorithm-independent because the set of objects specified by the program depends on the specific algorithm used to execute the program. Also, OLPS uses heavyweight UNIX processes that are created at runtime, which may potentially increase its runtime overhead.

SPEEDES is a C++ based simulation environment which supports sequential algorithm, time-driven algorithm, Time-Warp algorithm, and the SPEEDES algorithm (a combination of the time-bucket and the Time-Warp algorithm.) Using the unique SPEEDES algorithm, this system is capable of supporting interactive simulation in a parallel environment.

Reynolds[Rey88] described nine design variables (partitioning, adaptability, aggressiveness, accuracy, risk, knowledge embedding, knowledge dissemination, knowledge acquisition, and synchrony) for designing a distributed simulation algorithm. SPECTRUM is the testbed for exploiting all these possible parallel discrete event simulation algorithms. A library of applications and supporting routines is provided for parallel simulation protocol design experimentation. Supported algorithms include SRADS, null-message, and SRADS with local rollback.

3 Simulation Language

Maisie is a C-based derivative of MAY[BCM87] and has been influenced in varying degrees by distributed programming languages like CSP and SR[And81] among others. The discussion in this section emphasizes semantic issues. The syntax is presented via illustrations. Readers are referred to Appendix A for a precise description of the syntax of Maisie statements.

```
1   entity manager{max_printers}
2   int max_printers;
3   {
4     int units = max_printers;
5     message preq{ ename hisid; } ;
6     message releas;
7     for (;;)
8       wait until
9       { mtype(preq) st (units>0)
10          { units--;
11            invoke msg.preq.hisid with done; }
12      or mtype(releas)
13          units++; }
14  }
```

Figure 1: A Resource Manager: Single Resource

Maisie adopts the process-interaction approach to discrete event simulation[Nan81]. An object (also referred to as a PP for physical process) or set of objects in the physical system is represented by a logical process or LP. Interactions among PPs (events) are modeled by message exchanges among the corresponding LPs. We first describe the process representation and communication primitives of Maisie and subsequently indicate how they are used to describe simulation events. A resource manager is used as a running example to illustrate Maisie constructs. The manager manages two types of resources: channels and printers. The initial version of the resource manager only handles printer requests. It is subsequently extended to process channel requests.

3.1 Entities

A Maisie program is a collection of C functions and entity definitions. An entity definition (or an entity type) describes a class of objects. The definition of an entity is similar to the definition of a C function: the heading contains the entity name and a list of typed parameters and the body is a compound statement. Figure 1 describes a *manager* entity type that models the resource manager. The heading in lines 1 and 2 indicates that the entity type is called *manager* and has one integer parameter *max_printers*; the parameter refers to the number of printers initially available with the manager.

An entity instance, henceforth referred to simply as an entity, represents a specific object in the physical system. Maisie supports dynamic and recursive entity creation. An entity is created by the execution of a **new** statement which returns a unique identifier of type **ename**. This is a new type introduced by Maisie; variables of this type are used only to refer to entities. An entity can refer to its own identifier using the keyword **self**. For instance, the following statement creates a new instance of the *manager* entity type and saves the unique identifier assigned to the entity in variable $m1$ (which must be defined to be of type **ename**.)

```
m1=new manager{10};
```

By default, a new entity executes on the same processor as its creator entity. The new statement may optionally include an **at** clause to specify a different processor for execution of the new entity, as illustrated by the following example:

m1=**new** *manager*{10} **at** *pno*;

The value of *pno* may be an arbitrary non-negative integer; it is mapped to the node *pno modulo* N on the parallel architecture, where N is the number of nodes used in an execution of the Maisie program (N is specified as a command line argument). The **at** clause is ignored for sequential implementation of the program.

A Maisie entity may terminate itself in one of two ways: by executing a C return statement (if the return statement includes an expression, it is ignored) or by 'falling off the end' of the entity body.

3.2 Messages

Entities communicate with each other using buffered message passing. Every entity is associated with a unique message buffer. Asynchronous send and receive primitives are provided to respectively deposit and remove messages from the buffer.

Maisie uses typed messages. Every entity must define the types of messages that may be received by it. A message type consists of a name and a (possibly empty) parameter list. Message definition is syntactically similar to the declaration of C *structs*. Message parameters may be viewed as fields defined within a struct and are referenced using the '.' operator used to reference fields within a C *struct*. Two message types are defined for the *manager* entity type(lines 5-6): *preq* to request a printer, and *releas* to return the printer to the manager. The *preq* message contains one field called *hisid* of type **ename.**

Sending Messages An entity sends a message to another by executing an **invoke** statement. The invoke statement performs an asynchronous send: the sending entity fetches dynamic memory in which to copy the message parameters, delivers the message to the underlying communication network and resumes execution. Each message is timestamped by the runtime system with the current simulation time. A message is delivered to the destination buffer at the same simulation time at which it is sent, although it may not be accepted by the receiver immediately (as described below). The following example demonstrates two ways of sending a *preq* message to entity *m1*. The first statement specifies the message parameters explicitly; the second specifies that the message be copied from variable *oldreq* which must be declared to be of type *message preq.*

invoke *m*1 **with** *preq*{ **self** };
invoke *m*1 **with** *preq* = oldreq;

Receiving Messages An entity accepts messages from its message buffer by executing a **wait** statement. The **wait** statement has two components: an integer value called wait-time (t_c) and a Maisie compound statement called a resume block. A resume block contains a set of resume statements. The wait statement has the following form:

wait t_c **until**
{ declarations;
r_1;

or r_2;

$\vdots$

or r_n; }

Each r_i is a resume statement. A resume statement is similar to a guarded command as described below. Unlike other languages, the enabling condition in a Maisie resume statement can be a complex condition that involves multiple messages.

The most commonly used version of the resume statement references a single message type and has the following form:

[*mvar*=] **mtype**(m_t) [**max** v_i] [**st** b_i]
statement;

where m_t is a message type, *mvar* is a variable of type m_t, b_i is a boolean expression called a *guard*, v_i is a message parameter called a *ranker* and *statement* is any Maisie statement. The guard is a side-effect free expression that may reference entity variables and message parameters. If omitted, it is assumed to be the boolean constant *true*. The guard is said to be *local*, if it can be evaluated using only entity variables. The message type, guard and ranker are together referred to as a resume condition. A resume condition with message type m_t and guard b_i is said to be *enabled* if the message buffer contains a message of type m_t and b_i evaluates to *true* (b_i is evaluated only if the buffer contains a message of type m_t); the corresponding message is referred to as an *enabling* message. For instance, the resume statement in line 9 of entity *manager* is enabled only if the buffer contains a *preq* message and the local guard ($units > 0$) is true.

If the message buffer contains exactly one enabling message, the message is removed from the buffer and delivered to the entity in variable *mvar*, which then resumes its execution. The variable *mvar* is often omitted, in which case the enabling message is returned in a system defined variable **msg** whose type is the union of all message types declared in the entity. If the buffer contains more than one enabling message of a given type, the ranker is used to select a unique enabling message: if keyword **max** (**min**) is used, the enabling message with the largest (smallest) value for parameter v_i is delivered to the entity. If the ranker is omitted, the messages are ranked in increasing order of their timestamps. If two or more resume conditions are *enabled*, the timestamps on the selected enabling message of each type are compared and the message with the earliest timestamp is selected for delivery. The selected message is removed from the buffer and delivered to the entity either in the corresponding variable *mvar* or, if *mvar* is not specified, in variable **msg.** Note that a *mvar* specified in a resume condition is modified only if a corresponding enabling message is selected for delivery to the entity.

If no resume condition is enabled, the entity is suspended for a *maximum* duration equal to its wait-time t_c; if omitted, t_c is set to an arbitrarily large value. A suspended entity resumes execution *prior* to expiration of t_c, if it receives an *enabling* message. If no enabling message is received in the interval t_c, the entity is sent a special message called a **timeout** message. Timeouts are discussed further in the next subsection. Note that a non-blocking form of receive may be implemented by specifying t_c=0.

Once again, consider the *manager* entity type of Figure 1: the wait statement in lines 8-13 contains two resume statements. The first resume statement (line 9) specifies *preq* as the message type and was discussed earlier. The resume condition in the second statement (line 12) does not include a guard. This condition is enabled whenever a *releas* message is available in the buffer. As neither resume condition specifies a message variable, the enabling message is returned in variable

```
1      entity manager{max_printers}
         ⋮
5        message preq{ ename hisid; int count; } ;
         ⋮
8          wait until
9          { mtype(preq) st (msg.preq.count<=units)
             ⋮
```

Figure 2: Modified Resource Manager: Single Resource

msg. As seen in line 11, on receiving the *preq* message, the manager sends a *done* signal to the requesting job to indicate that the resource has been granted.

In the preceding example, the resume condition included only local guards. In general, the guard in a resume condition can reference message parameters as also refer to the enabling message of other resume conditions. Consider a modification to the resource manager, where an incoming request may ask for one or more printer units and the manager services requests using the *first-fit* discipline. The modifications are shown in Figure 2. The message type *preq* is modified to include an additional parameter *count* (line 5); the resume condition for the message is also modified to ensure that a specific message is accepted only if the requested number of units are available with the manager. The resume condition uses **msg**.*preq* to refer to an arbitrary message of type *preq* in the entity's buffer.

The modified manager of Figure 2 allows starvation in that frequently occurring requests for smaller number of units may forever block a large request from service at the manager. Starvation may be avoided by servicing incoming requests in first-in-first-out rather than the first-fit discipline. Thus the resume condition should be enabled only if the message at the head of the queue is an enabling message. A message at a specific position in the queue may be accessed by using function **qelem**(*pos*,m_t), where *pos* is a positive integer greater than 0 and m_t is a message type. The function returns the message at position *pos* in the queue of messages of type m_t. Program execution is aborted, if no such message exists in the buffer. A special form of this function called **qhead**(m_t) is defined to return the first message of type m_t in the entity's message buffer (where *first* is defined with respect to the ranker). The following resume condition uses function **qhead**() to specify *fifo* service of incoming requests:

mtype(*preq*) **st** ((**qhead**(*preq*)).*count* <=*units*)

Note that the guard in the preceding condition is evaluated only if the buffer has a message of type *preq*.

Finally, the manager may be modified to service request messages in the shortest-job-next discipline (where messages are ordered by the size of their request). The following resume condition accomplishes this simply by specifying that the messages be ordered on the basis of parameter *count*:

mtype(*preq*) **min** *count* **st** ((**qhead**(*preq*)).*count*<=*units*)

The compiler also defines a function **qsize**(m_t) which returns the number of messages of type m_t in the entity's message buffer. A special form of this function **qempty**(m_t) is defined, which returns *true* if the buffer does not contain m_t messages, and returns *false* otherwise. Assume that the manager is modified such that a request message is accepted only if no *releas* messages are available in the buffer. A simple way to do this is to strengthen the guard for the *preq* message, such that a *preq* message is accepted only if **qempty**(*releas*) returns true. The modified resume condition is as follows:

```
wait until
{ mtype(preq) st (qempty(releas) && msg.preq.count<=units)
    ⋮
}
```

Inclusion of guards in the resume conditions simplifies the entity definition as code to accept and buffer messages that cannot be processed immediately need not be included in the entity definition. The guard also facilitates rollback optimizations as discussed in section 5.

3.3 Events

A discrete event simulation is a sequence of events, where an event is any activity that changes the global state of the system. Each event in the model simulates some activity of interest in the physical system and may involve one or more objects. In developing a model of the resource manager, events include 'a job requesting a printer' or 'a job using the printer for t time units'. Events in a Maisie model are simulated by messages. For instance, the first event is modeled by a job entity sending a *preq* message to the *manager* entity; the second event is modeled by a job entity executing a wait statement with wait-time t such that a timeout message is received by the entity after t time units have elapsed.

Each event in a simulation is either definite or conditional. Assume that an entity schedules a future (timeout) event for time t_e at simulation time t_s, where $t_s \leq t_e$. The event is said to be *definite* or unconditional if its occurrence is independent of any other event in the system in the interval $[t_s,t_e]$ otherwise it is said to be *conditional*. Both definite and conditional future events may be scheduled by executing an appropriate wait statement.

Consider an entity that models a priority preemptible server. The entity receives two types of requests, *low* and *high*, where the arrival of a *high* message can interrupt the processing of a *low* message. On receipt of a message, the server schedules a future event to simulate completion of service for the request. If the incoming message is of type *high*, the completion event is scheduled as a definite event; for a *low* message, only a conditional completion event may be scheduled. Assume that each message needs 10 units of service. Service of a *high* message is simulated by the following wait statement which schedules a definite timeout message and sends a *done* message to the requesting entity to indicate completion of service.

```
wait 10 until mtype(timeout)
   invoke jobid with done;
```

Service of a *low* message is simulated by the following wait statement which schedules a *conditional* timeout message. This message is rescheduled if a *high* message is received by the entity in the interim.

```
rtime=10;
wait rtime until
```

{ **mtype**(*high*)
    preempt and serve high priority message;
or mtype(timeout)
    invoke *jobid* **with** *done*;
⋮
}

Maisie also provides a **hold** statement which may be used to unconditionally suspend an entity for a specified interval. This statement has the following form:

hold(t_c);

Execution of a hold statement suspends the entity and resumes its execution only after t_c units of simulation time have elapsed. Thus the wait statement used to simulate service of a *high* message may also be written as follows:

hold(10);
invoke *jobid* **with** *done*;

3.4 Compound Resume Conditions

We now consider resume statements at their most general level, where the resume condition may include multiple message types. The general form of a resume statement is as follows:

$mvar_a$ = **mtype**(m_a) [**max** v_a] [**st** b_a]
and $mvar_b$ = **mtype**(m_b) [**max** v_b] [**st** b_b]
⋮
and $mvar_n$ = **mtype**(m_n) [**max** v_n] [**st** b_n]
    statement;

The preceding statement is enabled if the message buffer contains a (different) enabling message for each conjunct in the resume condition. If the statement is enabled, the corresponding set of enabling messages is removed from the buffer and delivered to the entity in the specified message variables. If the message variables are omitted, keyword **msg** will contain an arbitrary enabling message. The **and** operator in the compound resume condition is a short circuit operator; the various conjuncts are evaluated in a left-to-right order and a conjunct is evaluated only if the message buffer contains an enabling message for each of the preceding conjuncts in the resume condition. The sequence of enabling messages for a compound resume condition is referred to as the *enabling sequence*. The largest timestamp of all messages in this sequence is referred to as the timestamp of the enabling sequence. If a compound resume condition is enabled together with other resume conditions in a wait statement, the timestamp of the enabling sequence is used in selecting a unique message (sequence) for delivery to the entity. In a compound condition of the form $mvar_a$=*r1* **and** $mvar_b$=*r2*, the guard in resume condition *r2* may reference message variable $mvar_a$. However, the value of variable $mvar_a$ is modified only if the corresponding enabling sequence is delivered to the entity; otherwise the value of message variables is left unchanged.

We illustrate the use of compound resume statements by modifying the *manager* entity type to also include the channel resource. Assume that requests for a channel are satisfied only in pairs that match a sending process with a receiving process. The sender process requests a channel using

```
1   entity manager{max_printers}
2   int max_printers;
3   {
4     int units = max_printers;
5     int cno = 0;
6     message chnls{ename hisid; } csend;
7     message chnlr{ename hisid; } crecv;
8     message preq{ename hisid; };
9     message releas;
10    for (;;)
11      wait until
12      { mtype(preq) st (units>0)
13          { units--;
14            invoke msg.preq.hisid with done;}
15      or  mtype(releas)
16          units++;
17      or csend= mtype(chnls) and crecv= mtype(chnlr)
18          { invoke csend.hisid with alloc{cno};
19            invoke crecv.hisid with alloc{cno};
20            cno++; }
21      }
22  }
```

Figure 3: Resource Manager: Multiple Resources

a *chnls* message and the receiver process uses a *chnlr* message. Initially assume that the manager has an infinite number of channels and a *chnls* message can be matched with an arbitrary *chnlr* message. The modified entity type is described in Figure 3 where the two new message types have been introduced in lines 6 and 7. The wait statement is augmented to include a resume statement (lines 17–20) to handle channel allocation: the statement is enabled only when its message buffer contains a *chnls* and a *chnlr* message; the associated actions simply informs the pair of requesting processes of the channel number that has been allocated for their communication[1].

Consider a modified specification where each pair of processes request a specific channel identified by a unique id, and the request is granted only when the manager has received matching requests and the desired channel number is available with the manager. The code fragment in Figure 4 indicates the primary modifications: message types *chnls* and *chnlr* are modified to include an integer parameter which refers to a channel number. Also, the modified resume condition ensures that the entity accepts a pair of matching requests only if the requested channel is available. Because the **and** operator is a short-circuit operator, the specified condition gives priority to the *chnls* message; that is, if the buffer contains many matched pairs, the pair with the earliest *chnls* message will be removed first. In case no pair of enabling messages is identified, the value of variables *csend* and *crecv* remains unchanged.

[1] We temporarily ignore the problem of overflow for the channel number

```
entity manager{max_printers}
int max_printers;
{
  int units = max_printers;
  message chnls{ename hisid; int cno; } csend;
  message chnlr{ename hisid; int cno; } crecv;
  ⋮
    wait until
      ⋮
    or  csend= mtype(chnls) st (cfree[msg.chnls.cno])
      and crecv= mtype(chnlr) st (msg.chnlr.cno==csend.cno)
        cfree[csend.cno]=false;
        ⋮
}
```

Figure 4: Modified Channel Request

3.5 Refinements

The primary cost of executing a wait statement is the cost of identifying enabling messages from the message buffer of the entity. As discussed in section 7, in general, this cost increases with the complexity of the guards, with compound resume conditions being the most expensive and resume conditions with local guards being the most efficient. However any complex resume condition can be refined to a simpler form. In this section, we use the resource manager of Figure 4 to illustrate the refinement process.

The resource manager accepts request messages for the channel only if both messages have arrived and the requested channel is available. This makes the initial program concise and allows the enabling condition for each event to be expressed directly. If each message was instead buffered internally in the entity, the analyst would have to design the data structures to store the requests: if the requested channel numbers belonged to a small range and each channel number was requested by at most one pair of processes, an array implementation would be the most efficient. In contrast, if the range was large (any positive integer) and multiple pairs could simultaneously request the same channel number, a hash table may be more appropriate. The compound resume condition allows these considerations to be postponed until the analyst has a chance to collect more information If channel requests are relatively conflict-free and both processes tend to request their access at approximately the same time, then the overhead for the compound statement is small and it may not be necessary to refine the code any further[2].

For the first refinement, assume that both requests for the channel are generated at about the same (simulation) time, but there is heavy conflict for the channels. Thus the buffer may contain many matched requests that are not accepted because the channel is unavailable. And the guard may be evaluated many times for a given message, increasing the execution time for the model.

[2]As discussed in section 7, a simple monitoring facility is transparently attached to each program to track the 'cost' of executing specific wait statements in an entity

```
entity manager{max_printers}
int max_printers;
{
  int units = max_printers;
  message chnls{ename hisid; int cno; } csend;
  message chnlr{ename hisid; int cno; } crecv;
  ⋮
    wait until
      ⋮
    or  csend= mtype(chnls)
      and crecv= mtype(chnlr) st (crecv.cno==csend.cno)
        { if (cfree[csend.cno])
            inform requesting processes;
          else buffer matched request for channel cno;
          ⋮
}
```

Figure 5: Modified Channel Request : Refinement 1

This inefficiency may be reduced by modifying the resume condition such that a pair of matched requests is accepted and buffered internally if the requested channel is unavailable. The refined resume condition is shown in the code fragment in Figure 5, where the actions of the entity are specified using pseudo-code. The internal queue may be structured to optimize the search by using the channel number of available channels as the search key.

The next refinement is useful if the matched requests are *not* generated together. In this case an earlier unmatched message may be used to (unsuccessfully) find a matched pair many times and increase the overhead. The compound resume condition may be subdivided into two simple resume conditions, one for each message type. If a matching request is available, both requests are satisfied, otherwise request is buffered internally until a matching request has been received. Note that the data structures and code for the manager process becomes increasingly more complex; however the complexity in the model is introduced only on an *as needed* basis after the prototype has been validated.

3.6 Program Initiation and Termination

Every Maisie program must include an entity called **driver**. The runtime system begins execution by creating an instance of this entity and executing the first statement in its body.

A Maisie program terminates in one of two ways:

- The simulation clock exceeds the maximum simulation time specified by function **max-clock()**.
- All entities are suspended and no messages (including timeouts) are in transit.

When a termination condition is detected, the runtime system sends an **endsim** message to every entity in the system. An entity may either accept or ignore this message. In the former case, it may use the message to take appropriate actions before termination, including printing accumulated statistical data.

3.7 Example

In this section we develop a complete Maisie model for a simple queuing network. The model is refined for parallel execution in a subsequent section. 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 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. 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 6. 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 10). 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. When the simulation is completed, every entity including the statistics collection entity is sent an **endsim** message. On receiving this message, the entity prints 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 i^{th} 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.

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

entity driver{}
{  ename rtr[N],q[N],stat1;
   int i;
   stat1= new basic_stats{"Average System Time"};
   for (i=0;i<N;i++)
     q[i] = new queue{5,1000,stat1};
   for (i=0;i<N;i++)
     rtr[i] = new router{10,10,q};
   for (i=0;i<N;i++)
     invoke q[i] with idmsg{rtr[i]};
}

entity router{njobs,mtrips,qids}
 int njobs, mtrips;
 ename qids[N];
{ message job{int stime; int count;} j1;
  int i;
  for (i=0;i<njobs;i++)
     invoke qids[i%N] with job{0,0};
  for (;;)
   wait until mtype(job) {
    j1=msg.job;
    if (j1.count < mtrips) {
     hold (j1.stime-sclock());
     invoke qids[iurand(0,N-1)] with job=j1;}
  }
}

entity queue{nsrvr, mtime, statid}
 int nsrvr,mtime; ename statid;
{ int i,t1,lastj[nsrvr];
  ename rid;
  message job{int stime; int count;} j1;
  message idmsg{ename id;};

  wait until mtype(idmsg) rid= msg.idmsg.id;
  for (i=0;i<nsrvr;i++)
   lastj[i]=0;
  for (;;)
   wait until mtype(job) {
    j1=msg.job;
    t1=j1.stime;
    for (i=0;i<nsrvr;i++) {
     lastj[i]=MAX(t1,lastj[i]) + expon(mtime);
     t1=lastj[i];
    }
    invoke rid with job{t1,j1.count+1};
    invoke statid with value{(t1-j1.stime)};
    }
}
```


Figure 6: Maisie model of CQNF

4 Parallel Implementations

A sequential Maisie program can be refined to a parallel implementation simply by allocating processes among available processors and compiling and executing the programs in the parallel environment. Remote process creation was described in the preceding section and will not be discussed further. The runtime system for the parallel environment has two major responsibilities: providing interprocess communication (IPC) facilities and implementing the distributed simulation algorithm. The Maisie IPC facilities have been designed to operate in conjunction with existing IPC packages like the UNIX IPC or the Cosmic Environment[Sei85]. They can be easily modified to work on top of other distributed operating system kernels. The distributed simulation algorithm is implemented via a set of routines that are essentially transparent to the Maisie programmer. Entities mapped to a common processor are simulated sequentially and entities on different processors may be synchronized using either a *null* message algorithm, a conditional event algorithm, or the space-time simulation algorithm. This section discusses how a Maisie program can be executed transparently using any of the preceding algorithms. In the interest of brevity, we do not describe the respective algorithms, but simply indicate how the information required by each algorithm may be extracted from the Maisie program. Parallel implementations of Maisie using the optimistic space-time algorithms are operational and have been described in [BCL92, BL90b]. An implementation using the conservative conditional event algorithm is in progress[Jha91].

4.1 Null-Message Algorithm

A Maisie program can be executed using either *lazy* or *demand-driven* variations[SS89] for the *null*-message algorithm[Mis86]. In order to implement any *null*-message scheme, each LP must be aware of the set of its source and/or destination LPs. (In the absence of this information, *null* messages may have to be broadcast, making the implementation inefficient for simulation of a sparsely connected physical system). For each LP in the simulation, the runtime system implicitly maintains two variables, the *source-set* and the *dest-set* which respectively refer to its set of source and destination entities. We briefly indicate how the two sets are maintained for each entity (or LP) by the runtime system.

In order for a Maisie entity LP_a to send a message to LP_b, LP_a must have the identifier for LP_b. As all entity identifiers must be declared as type **ename** variables, the *dest-set* of an entity is assumed to comprise of all **ename** variables and is maintained transparently by the runtime system. To determine the *source-set* of an LP, it is sufficient to determine the set of entities that have access to its name. The name of an entity LP_a may be made available to LP_b in one of two ways: if LP_b creates LP_a, or if LP_b received the information in a message from another LP (including LP_a). In either case, the problem is to add LP_b to the *source-set* of LP_a. In the first case, this is done transparently by implicitly including LP_b in the initialization information used to create LP_a. In the second case, an entity that sends an **ename** LP_a to another LP must first execute a system call *addsrc*, which updates the *source-set* of LP_a. The runtime system can detect violations of the above rule and take appropriate action including abnormal termination of the simulation. On termination of an entity, the system automatically removes the name of the terminated entity from all *source-sets* and *dest-sets*.

The overhead associated with maintaining the *source-set* and *dest-set* information is negligible if the communication topology in the application is essentially static, as is the case for many simulations. Other than the single system call required to maintain the *source-set* information for an entity, the simulation algorithm is completely transparent to the Maisie programmer.

4.2 Conditional Event simulation

Chandy and Sherman[CS89a] have described a conservative simulation algorithm that does not rely on *null* messages to guarantee progress. Instead the algorithm distinguishes between definite and conditional events in a simulation. Generation of a message by an LP_c is a definite event, if it depends only on its current state (and the sequence of messages that have been received by the LP) and is not affected by any subsequent messages that may be received by it. An event that is not definite is conditional. If at some point in the computation, the next event of every entity is a conditional event, the simulation may deadlock. Rather than use *null* messages to avoid deadlocks, the algorithm suggests that cond_x, the timestamp on the earliest conditional event for LP_x be recorded consistently for all LPs. The minimum cond_x represents the earliest conditional event in the system, which may then be transformed into a definite event.

In order to execute a Maisie program using the conditional event algorithm, it must be possible to distinguish between definite and conditional events, as also to determine the cond_x for each entity. The resume conditions specified in the wait statement may be used to transparently distinguish definite events from conditional events. If the resume condition includes only the **timeout** message, the event is definite, otherwise it is treated as being conditional. In the former case, the simulation time of the entity is immediately incremented by the wait-time specified in the wait statement and the action associated with the receipt of the **timeout** message are executed to generate the appropriate messages as definite events. Conversely, if the resume condition indicates a conditional event, cond_x for the entity may be determined directly from the wait-time specified in the corresponding wait-statement. Once the definite events and the cond_x have been determined, the program may easily be executed using the conditional event algorithm. In section 6, we indicate additional mechanisms that may be used to distinguish definite events from conditional events.

4.3 Space-Time Simulation

In order to execute a Maisie program using the space-time algorithm, it must be possible to perform three primary tasks transparently: checkpointing, recomputation and determining the duration over which the simulation has converged. Functionally, checkpointing is transparent to the programmer; an entity changes its state on receipt of a message, and the old state is saved in a timestamped queue. Rollback is implemented automatically by tracking the timestamps on the messages delivered to each entity and the algorithm for detecting simulation convergence described in [CS89b] can easily be made transparent to the Maisie programmer. A detailed description of the transparent implementation has been provided in [BCL91].

4.4 Example

In this section we show how the Maisie model described in section 3.7 is refined for parallel execution, 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 7, 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 as follows:

```
entity driver{}
{  ename rtr[N],q[N],stat1;
   int i;
   stat1= new basic_stats{"System Time"};
   for (i=0;i<N;i++)
     q[i] = new queue{5,1000,stat1} at i;
   for (i=0;i<N;i++)
     rtr[i] = new router{10,10,q} at i;
   for (i=0;i<N;i++)
      invoke q[i] with idmsg{rtr[i]};
}
```

Figure 7: Parallel CQNF: Driver Entity

% mayc cqnf.may -arch s2010 -algo optimistic

The preceding command specifies that the Maisie program in file *cqnf.may* be compiled for execution on the Symult S2010 multicomputer using an optimistic simulation algorithm. Figure 8 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.

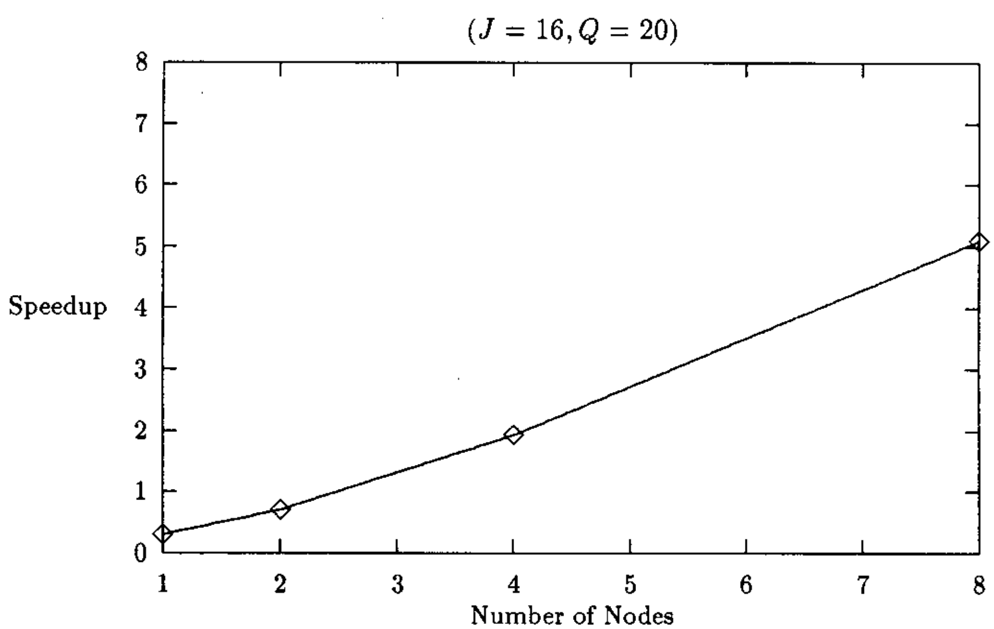

Figure 8: Speedup for CQNF

5 Optimizations for Optimistic Algorithms

This section discusses a variety of optimizations to reduce state saving and recomputation overheads for optimistic simulations. The discussion is divided into two sections with the first section addressing user-transparent optimizations and the latter user-specified optimizations that are specific to a given application.

Optimistic algorithms require recomputation whenever the runtime system detects that a message sequence *delivered* to an LP in the simulation is different from the message sequence delivered to the corresponding PP in the physical system (or its model). This may be either because the former contains a message that is not present in the latter (or vice-versa), or because the message sequence in the simulation is a permutation of the sequence in the physical system. In the first case, recomputation is generally unavoidable, as the incorrect message must be canceled. However, in the second case, rollback may often be avoided as explained subsequently. *In the remainder of the paper, we are only concerned with rollbacks that are caused by out-of-order message delivery.*

In implementing rollbacks, two decisions must be made: when should a rollback be initiated and what is the recomputation point (the state from which the system must be reexecuted). The number of events from the recomputation point to the current state of the system is referred to as the *rollback distance.* Existing implementations of optimistic simulations take a conservative approach: a rollback is required whenever an out-of-order message (m_w,t_w) is delivered to a simulation object, say LP_a, and the recomputation point is the last checkpointed state of LP_a immediately prior to t_w. However, as discussed in this section, the rollback distance of a recomputation may often be reduced significantly. We introduce the term *artificial rollback* to refer to a rollback whose rollback distance can be reduced while maintaining the correctness of the simulation. Detection of artificial rollbacks improves the execution efficiency of optimistic simulations by reducing recomputation and state saving overheads.

Let r_1 be a subsequence of the correct sequence of messages that is delivered to some entity LP_a. Let F_1 and s_1 respectively be the final state of the entity and the sequence of output messages generated by the entity as a result of receiving the messages in r_1. The state of an entity includes its local variables and its message buffer. Let r_2 be a permutation of r_1, $r_1 \neq r_2$, and F_2 and s_2 be the final state and the sequence of output messages generated due to delivery of r_2 to LP_a. An out-of-order message is henceforth referred to as a *warp* message. Any one of the following four relationships may hold among F_1, F_2, s_1 and s_2.

- $F_1 \neq F_2$ and $s_1 \neq s_2$
- $F_1 \neq F_2$ and $s_1 = s_2$
- $F_1 = F_2$ and $s_1 \neq s_2$
- $F_1 = F_2$ and $s_1 = s_2$

In a typical optimistic implementation like TWOS[JBWea87], delivery of sequence r_2 rather than r_1 would cause recomputation of LP_a and possibly other entities with which communication has occurred, *in each of the four cases.* However, it is clear that only the first case requires a propagating rollback. The recomputation can be considerably reduced in the second and third case, and completely eliminated in the last case. We note that the optimization suggested in this paper differ from those implied by the lazy message cancellation for Time Warp[Gaf88]. The latter prevents unnecessary cancellation of messages that are regenerated after rollback. In contrast, detection of artificial rollbacks directly reduces the recomputation distance for the object that receives a *warp* message. As the recomputation after a rollback also incurs state saving overheads, reductions in the rollback distance also help to reduce the overall state saving overheads. In the remainder of this section, we describe how a variety of artificial rollbacks may be detected transparently by the runtime system.

5.1 Transparent Optimizations

Assume that entity LP_a executes a wait statement at simulation time t_i. The following two variables are defined for every entity:

- $mset_a(t_i)$: set of enabling messages for LP_a at time t_i.
- $tres_a(t_i)$: timestamp(s) on the enabling message(s) accepted by the entity when it resumes execution after executing wait statement at t_i.

Variables *mset* and *tres* are automatically maintained for every entity. Henceforth, we will drop the subscript on *mset*, when the corresponding entity is uniquely indicated by the context.

Assume that a *warp* message (m_w,t_w) is received by an entity when its simulation time is t_n. Let t_l be the latest time preceding t_w at which the entity's state was saved. On receipt of m_w, traditional optimistic simulators immediately initiate a rollback to t_l. However, in the optimized implementation, the entity needs to be rolled back only to the earliest t_r, $t_l \leq t_r \leq t_n$, such that m_w belongs to $mset(t_r)$ and t_w *is less than* $tres(t_r)$. In many cases, t_r may be greater than t_l, and in some cases t_r may be equal to t_n, indicating that the rollback is unnecessary. We present a few examples.

The first example uses the most common form of the wait statement where the resume conditions do not include a guard. Consider the preemptible priority server of section 3.3 that receives messages of type *high* or *low* to represent requests of different priority, where arrival of a *high* message may preempt service of a *low* message. Consider the effect of delivering the message sequence (5,*high*), (9,*low*), (7,*high*), (18,*low*), (14,*high*) to the server. Assume that message (5,*high*) is accepted by the server at time 5. Then, *mset*(5) for the server includes only **timeout** messages; other messages including (9,*low*) and (7,*high*) that are received by the server, will be stored in its message buffer until it receives a **timeout** message. The delivery order for these two messages is immaterial to the correctness of the simulation, as long as message (7,*high*) arrives at the server before simulation time 15 (note that *mset*(15) includes messages of type *high*). Furthermore, if message (14,*high*) is delivered to the entity after simulation time 15, even though the message belongs to *mset*(15), rollback may be unnecessary as *tres*(15)=7, due to the server initiating the service of message (7,*high*).

Maintaining an entity's *mset* when its wait statements refer only to message types has relatively low overheads. As most entities contain a small number of message types (almost never exceeding 30), the *mset* can be typically saved in a single word by using a unique mask for each message type defined by the entity. An additional word is required to store *tres* for every recorded state. The processing overhead is also small: for each recorded *mset* one logical *and* operation and a comparison is required to determine if a *warp* message (m_w,t_w) belongs to the corresponding *mset* and one integer comparison is required to determine if t_w is greater than the recorded *tres*.

We next consider the case where a resume condition also includes a guard. In general, an entity may use any of its state variables and message parameters to add additional discriminatory power to the resume condition. We consider two examples: a bounded buffer and a priority server.

Consider the wait statement executed by a bounded buffer to ensure that it accepts a request for data from a consumer (modeled by a *more* message) only if it is not empty (*nin*>*nout*):

wait until
{ **mtype**(*more*) **st** (*nin*>*nout*)
send next data-item to the consumer;
⋮

The preceding resume condition includes entity variables but no message parameters; hence its *mset* is completely determined when the wait statement is executed and this case is similar to the preceding example. Consider the priority server where the resume condition includes message parameters. Assume that the *request* message includes a parameter, called *prio* which refers to the priority of the request message. Let *cur_prio* be a state variable of *server*, which indicates the priority of the current message being serviced. The following wait statement may be used by an entity that is serving a request to ensure that it is interrupted only by *request* messages that have a priority higher than *cur_prio*:

wait t_l **until**
{ **mtype**(*request*) **st** (**msg**.*request.prio*>*cur_prio*)
preempt and serve higher priority request;
or mtype(timeout)
simulate service completion;}

If a resume condition includes message parameters, computing the *mset* of the entity and determining if a *warp* message belongs to a recorded *mset* are both more expensive than in the previous case. The guard is used to create a parameterized function, where the parameters correspond to the entity variables and message parameters referenced in the guard. Assume t_l, t_n, t_w and t_r as defined previously. In order to determine if a *warp* message belongs to the entity's *mset*, the function must be executed for each recorded *mset* in the interval $[t_l,t_r]$. Also, if the resume condition includes function calls, implementation of this optimization becomes more expensive. Experimental studies are needed to evaluate the cost of this optimization to determine its overall impact on the completion time of a simulation.

In a similar manner, a ranker may be used in a resume condition to specify the order in which messages of a given type are to be serviced. This would permit the runtime system to initiate a rollback only if the rank of the *warp* message is higher (or lower) than that of the messages processed earlier by the entity. The overhead, in terms of recording and scanning recorded *mset*s is of similar magnitude to the previous case, as the ranking parameter and rank of the enabling message can be recorded as a boolean expression. Additional overhead is incurred in maintaining the message buffer as an ordered queue. (Note that in the same entity, it is possible to have another wait statement whose resume condition requires the messages to be ordered by their timestamps or even by a different parameter.) To minimize unnecessary overheads, syntactic tags are used to ensure that this condition is known at compile time. This allows the system to maintain an ordered queue *only when the queue would otherwise need to be maintained by the programmer.* Thus the queue maintenance does not really contribute to additional overhead.

5.2 User-specified Optimizations

This section deals with detecting artificial rollbacks in situations where a message is *processed* rather than simply *delivered* in an incorrect order. In general, the information needed to detect artificial rollbacks when a message has been processed in an incorrect order can rarely be extracted transparently from the simulation and must be specified explicitly by the programmer.

Probe Messages We define a *probe* message to mean a message whose processing does not alter the state of the recipient LP. The primary purpose of such messages is to obtain state information about the destination LP, as for example, whether the LP is currently *active* or *idle*. Processing a *probe* message in an incorrect order would typically result in situations where $F_1=F_2$ but $s_1 \neq s_2$. Although a message may sometimes be detected transparently to be a *probe* message, the overhead may be reduced considerably by using syntactic tags. A message type is declared as a probe by preceding its declaration with the keyword **probe**. The following statement illustrates the use of a probe message called *status*:

probe message *status*{**ename** *jobid*;};

⋮

wait until mtype(*status*)
  invoke msg.*status*.*jobid* **with** *reply*{*idle*};

where *idle* denotes the current status of the LP. If a *warp* message (m_w,t_w) is determined to be a probe, the message is processed in the state that is saved at or immediately prior to t_w. The subsequent events that have already been processed by the entity do not need to be canceled. Once again, if the state of the entity is saved after every event, implementing this optimization adds negligible overhead but may significantly reduce recomputation and consequently, state saving overheads.

Associative Messages The concept of *probe* messages can be extended to the notion of associative messages: sequences r_1 and r_2 defined in the previous section are said to be *associative* if messages in either sequence may be processed without affecting correctness of the simulation. As an example of an associative sequence, consider the following two sequences that are input to a FIFO server: r_1=(5,10,LP_1),(18,7,LP_2),(30,8,LP_1) and r_2=(5,10,LP_1),(30,8,LP_2),(18,7,LP_1) where the message parameters respectively represent the message timestamp, desired service duration and the requesting LP. The two sequences are associative, as the final state of the server and the output message sequences to each customer are the same, regardless of which sequence of input messages is actually processed by the server. Detection of associative sequences is important because it allows messages to be processed out of order, thus reducing rollback and state saving overheads.

As another example of an associative message sequence, consider a bounded buffer that receives data from a producer process via *put* messages and requests for the data from a consumer process via *get* messages. Let (p1,p2,c1,p3,c2) be the 'correct' message sequence received by the buffer, where p1..p3 represent *put* messages and c1..c2 *get* messages. Sequences (p1,c1,p2,p3,c2) and (p1,p2,p3,c1,c2) are both associative with respect to the correct sequence.

A warp message m_w is associative, if the subsequence of messages including m_w that is delivered to the entity is associative. Given a warp message, in general, it is hard to automatically decide if it is associative. Our aim is to suggest language primitives that allow a programmer to specify sufficient conditions to isolate warp messages of a given type. For this purpose, we define a separate, optional section of an entity called the *warp* section. This section consists of a set of warp statements, each of which is syntactically similar to a resume statement. Each warp statement defines a warp condition and warp actions, where the former is a temporal predicate and the latter is a C or Maisie statement. A warp statement has the following form:

```
#define R msg.request
entity server {}
{
  int nojobs=0;
  int idle=true;
  message request {int stim; ename jobid; } ;

  for(;;) {
    idle=true;
    wait until mtype(request) {
        idle=false;
        hold(msg.request.stim);
        invoke msg.request.jobid with done;
        nojobs=nojobs+1;}
    }

    warp{
      mtype(request) st (idle) in (R.tstamp, R.tstamp+R.stim)
        nojobs=nojobs+1;
    }
}
```

Figure 9: A FIFO Server with *Warp* Section

mtype(m_t) **st** b_i [**in** (t_i,t_j)]
statement ;

A warp condition includes a message type, a guard and an optional temporal component that defines a time interval. If omitted, the interval is assumed to be the single time instant corresponding to the timestamp of the warp message. Note that b_i, t_i or t_j may include message parameters. A message of type m_t is associative, if the guard in its warp condition is continuously true at every instant in the corresponding time interval. Assuming that t_n, t_l, t_r and t_w are defined as in the previous section, an associative *warp* message is processed in the state of the recipient entity saved at time t_r. In addition, to ensure that the effect of the warp message is included in the final state of the entity, the specified warp actions must be executed in the state of the entity at t_n. If an entity includes a *warp* section, the runtime system is required to save the state of the entity after every event so that the warp condition may be evaluated over the specified interval.

We illustrate these ideas in the context of a FIFO server. Figure 9 presents the entity definition for a FIFO server. On receiving a *request* message, the entity simulates its service by executing an appropriate hold statement and sends a *done* message to the requesting process. The warp section includes a warp condition for message type *request* which indicates that the entity may process an out of order *request* message, if it was idle during the time the message would be serviced. The warp actions ensure that the count of serviced messages is updated correctly.

To summarize, the rollback optimization algorithm may be informally stated as follows: assume a *warp* message m_i is received by LP_a, where the timestamps t_l, t_n, t_w and t_r are as defined in the previous section. For a given m_i, if $t_r \geq t_n$, rollback is unnecessary and m_i is simply inserted into LP_a's message buffer. If $t_r < t_n$ and m_i is a probe message, m_i is processed by the entity in its state defined at t_r, and recomputation of LP_a is unnecessary. If $t_r < t_n$ and m_i is an associative message, rollback is again unnecessary; m_i is processed by the entity in its state defined at t_r, and the corresponding warp actions are executed in the current state of the entity at t_n. The rollback and recomputation process is initiated by the system, only if none of the preceding optimizations are applicable.

Dead States The state of an entity is typically saved after each event to minimize rollback distance[Fuj89]. However, some states in an entity may be *dead* states. A dead state is a state that is never used to initiate a recomputation. Consider a timeout message that is scheduled as a definite event. From the definition of a definite event in section 3.3, it follows that if the sequence of messages received by the entity preceding some timeout message is correct, the timeout message must also be correct; the timeout message can never be the first incorrect message. In other words, the state immediately preceding the receipt of the timeout message is a dead state that will never be used to initiate a recomputation, and hence need not be saved. For entities with large states, this may be a significant improvement. For a specific application, it may be possible for an analyst to identify other states as dead states. The programmer may explicitly flag some resume statement r_i, to indicate that if the entity resumes its execution by executing r_i, the preceding state need not be saved. Such a resume statement is indicated simply by replacing keyword **mtype** in the resume condition by keyword **ctype**. Note that, in the worst case, incorrectly labeling a state as a dead state may degrade the completion time by increasing the rollback distance, but will not affect its correctness. Of course, if the entity also includes a warp section, the dead states must nevertheless be saved to allow the warp condition to be tested exhaustively.

6 Optimizations for Conservative Algorithms

The performance of conservative algorithms can be improved by reducing synchronization overheads. Synchronization overheads, in turn, can be reduced if each process has good *lookahead*[Fuj88]. A lookahead process is defined to be a process whose behavior can be predicted for some future time interval. A process is said to have lookahead ϵ, if for any t^3, the state of the process can be predicted in the interval $[t, t+\epsilon)$. In order to have good lookahead, it is important that a process have information about the state of each of its predecessor process. For instance, consider a fifo server that has only one predecessor process. Such a server has excellent lookahead: whenever it receives a job it can immediately predict the time at which it will depart. However, if the server has two predecessors, say P and Q, the server can predict the departure time of an arriving job only after it has received a message from both P and Q. If the predecessors feed the server at different rates, the server must explicitly synchronize with its predecessors to determine the departure time of an incoming jobs. This section describes optimizations which allow the lookahead for some type of objects to be extracted transparently by the runtime system. In addition specific primitives are provided to allow programmers to explicitly encode lookahead in an entity.

Assume that the *source-set* and *dest-set* data structures are maintained for each process as described in section 4.1. A basic conservative algorithm may be implemented transparently as follows: whenever an entity sends a (non-null) message, say (m_i, t_i) it also sends a *null* message

[3] Assume that t is quantified over the simulation interval

timestamped t_i to every other entity in its *dest-set.* A message say (m_i,t_i) is delivered to an entity only if the entity has received some message timestamped t_i or greater from every entity in its *source-set.* As long as every cycle of entities in the model has at least one lookahead process, progress is guaranteed[Mis86]. The basic scheme outlined above may perform poorly for many applications. However, as discussed in [Fuj88], the performance can be improved if lookahead for the various entities is exploited aggressively. In a Maisie program, lookahead for an entity may often be extracted transparently: if the *mset* of a suspended entity only contains timeout messages, the wait-time specified in the most recent wait statement represents its lookahead and may be used to advance its simulation clock even in the absence of a message from all members of its *source-set.*

For some entities, it may be possible to extract the lookahead only using application-specific information. For instance, as described in [Nic88] and [LL90], presampling of random numbers may be used to generate lookahead for a fifo server as also for a priority server. Every Maisie entity includes a compiler-defined local variable called **lookahead**. When an entity schedules a definite future event, the runtime system automatically updates this variable to reflect the lookahead time for the entity. In addition, an entity may explicitly compute its lookahead and store it in this variable before executing a wait statement. The control graph model described in [CS89c] uses a similar feature to permit automatic extraction of lookahead. Figure 10 illustrates lookahead computation for a priority server. When the server is idle, its lookahead is the presampled service time for the next request (represented by variable *ntime*). When serving a *low* message, its lookahead is the minimum of the remaining service time for the request (*rtime*) and the presampled service time (*ntime*) for any *high* message that may interrupt it. By setting *rtime=MAXINT* when the server is idle, its lookahead in the preceding two cases is simply the minimum of rtime and ntime as shown in the figure (line 14). The server uses a hold statement to service a *high* message, where its lookahead can be computed automatically by the runtime system (line 19).

7 Implementation Issues

Maisie has been implemented on both sequential and parallel architectures. The current implementation does not support rankers or compound resume statements. Implementation of these constructs is in progress. The remainder of this section discusses the implementation of wait statements whose efficiency has a significant impact on the execution efficiency of Maisie programs.

The Maisie wait statement allows an entity to accept a message from its buffer, only when it satisfies the guard in some resume condition. This implies that a sequential Maisie model is executed using an *interrogative* simulation algorithm where a message is delivered to an entity only when it is ready to accept it. In contrast, an *imperative* algorithm will deliver a message at the simulation time specified by the message timestamp; if the entity is not ready to process the message, it must be buffered internally. As discussed in the previous sections, the Maisie wait statements allow an analyst to express the enabling condition for an event directly, which facilitates the design of concise programs and can significantly reduce model development time. Also, the wait statement is useful in reducing simulation overheads in parallel execution of Maisie models: as discussed previously, it may be used to separate some definite events from conditional events and may also be used to identify a variety of artificial rollbacks. However, in general, event management with an interrogative algorithm may not be as efficient as with an imperative algorithm. The primary source of inefficiency in the former is the cost of selecting the next message for delivery to the entity (which is not necessarily the message with the earliest timestamp). In this section, we examine the factors that contribute to this cost and discuss techniques to reduce their effect on the overall

```
#define MIN(a,b)   ((a < b)?a : b)
entity server { mean }
   int mean;
{
   message high { ename hisid; } ;
   message low { ename hisid; } ;
   ename jobid;
   int rtime, ntime, busy, otime;
   rtime=MAXINT;
   busy=0;
   ntime=expon(mean);
   for(;;)
   {
      otime=rtime + sclock();
      wait rtime until
      {  mtype(high)
         { rtime=otime - sclock();
           hold(ntime);
           ntime=expon(mean);
           invoke msg.high.hisid with done; }
         or mtype(low) st(!busy)
         { busy=1; jobid=msg.low.hisid;
           rtime=ntime;
           ntime=expon(mean);
           lookahead=MIN(ntime,rtime) + sclock(); }
         or mtype(timeout)
         { busy=0; rtime=MAXINT;
           invoke jobid with done; }
      }
   }
}
```

Figure 10: A Priority Server with Lookahead

efficiency of the implementation.

For every wait statement executed in an entity, we define a metric called *lbuffer*, where *lbuffer* is the number of messages that must be inspected from its message buffer before some enabling message is selected for delivery to the entity. For an imperative algorithm, the *lbuffer* is at most 1, because the selected message is simply the message at the head of its buffer. For the interrogative algorithm, in general, the upper bound on *lbuffer* cannot be defined as tightly. However, if every resume condition in a wait statement references a single message type and has a local guard, the interrogative algorithm can be implemented almost as efficiently as the imperative algorithm. In this case, the *mset* of a suspended entity is completely specified by a few message types: a message of type m_t belongs to the *mset* only if the most recent wait statement executed by the entity included a message type m_t with a *true* guard. As most entities define a small number of message types (almost never exceeding 30), the *mset* may be stored as a single word bit mask: each bit represents a unique message type and is set to 1 if and only if the corresponding type belongs to the current *mset*. The *mset* is stored outside the data-space of the entity. Furthermore, the message buffer of an entity is implemented as a number of separate lists, where each list contains messages of a unique type that are ordered by their timestamps (or alternately by the ranker). Messages in different lists are also linked in the order of their timestamps. This implies that the time to identify the earliest enabling message has a tight upper bound given by the number of message types defined for the entity. In many cases, the *lbuffer* may be exactly 1; in particular, if the wait statement executed by an entity does not restrict the messages that may be received by it, which corresponds approximately to using the imperative algorithm, the *lbuffer* for the entity would be at most 1. With a naive implementation, this time may instead be proportional to the number of messages in the entity's buffer, which varies dramatically.

If a resume condition references message parameters, the *mset* of an entity can no longer be specified by message type alone. If the guard for message type m_t references a message parameter, the guard must be evaluated for successive m_t messages from the buffer until some enabling message (not necessarily of type m_t) is identified or it is determined that no resume condition is enabled. This implies that, in general, the time to identify an enabling message is now bound by the number of messages of type m_t in the buffer. A simple monitoring facility is attached to each wait statement to track its *lbuffer* (average, maximum, median etc.). If the *lbuffer* for such a wait statement is found to be large, it may be more efficient to refine the corresponding resume condition such that the messages are buffered internally and can be searched efficiently by the programmer. The built-in monitoring facility can be used to determine if the elaboration is likely to yield any benefit. As the code for internal buffering and its efficient searching can be reasonably complex, it is desirable to postpone this refinement until appropriate information about its possible impact is available. (Note that if messages in the internal buffer are ordered on the same basis as in the external buffer and searched sequentially, the efficiency gains will be minimal.)

If an entity includes a wait statement with a compound resume condition, it is transformed to an entity with simple resume conditions such that every incoming message is buffered internally in the entity. Compound resume conditions are effectively translated into equivalent if statements with compound boolean expressions that check if the set of messages needed to enable a resume condition have been received. This implementation has the benefit that buffer management routines that were used for simple resume conditions can continue to be used. Its primary drawback is that because a message is now accepted by the entity before it is actually processed, opportunities to optimize artificial rollbacks may be missed. Note that the *lbuffer* of the entity is still computed on the basis of the number of messages that are inspected before some resume condition is found to be enabled.

8 Conclusion

Simulations are typically large and complex programs. This paper described a language called Maisie that encourages iterative design of simulations using step-wise refinement. The initial Maisie program is a concise prototype that specifies events and their enabling conditions at a high level of abstraction. The central construct of Maisie is the wait statement which allows an entity to delay accepting messages that would otherwise have to be buffered internally. Appropriate use of the wait statement leads to succinct programs and reduces program development time. Monitoring facilities are transparently attached to an entity to track the cost of determining when an event is enabled. This metric can be used to refine certain parts of the model to improve its efficiency. The initial program is executed using a sequential simulation algorithm and may be tested on a workstation or PC. If the completion time of the sequential simulation is not acceptable, it may be refined for parallel execution.

The initial transformation of a Maisie model to a parallel implementation simply allocates Maisie processes among available processors. At this stage, the simulationist 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 both conservative and optimistic algorithms. After identifying the most suitable simulation algorithm, the final refinements to the model are dictated by the specific nature of the simulation algorithm. These refinements use application and algorithm specific information to reduce the completion time for the simulation program. An optimistic implementation attempts to reduce rollbacks by using state correction techniques or identifying probe messages and reduces state saving overheads by identifying dead states. A conservative implementation reduces synchronization overheads by distinguishing between definite and conditional events and by aggressively exploiting the lookahead in an application.

References

[Abr88] Marc Abrams. The object library for parallel simulation (OLPS). In *Proceedings of the 1988 Winter Simulation Conference*, pages 210–219, San Diego, California, December 1988.

[And81] G.R. Andrews. Synchronizing resources. *ACM TOPLAS*, 3(4):405–430, October 1981.

[BCL91] R.L. Bagrodia, K.M. Chandy, and W. Liao. A unifying framework for distributed simulation. Technical report no. UCLA-CSD-910071, Computer Science Department, UCLA, Los Angeles, CA 90024, October 1991.

[BCL92] R.L. Bagrodia, K.M. Chandy, and W. Liao. An experimental study on the performance of the Space-Time simulation algorithm. In *the Proceedings of 6th Workshop on Parallel and Distributed Simulation*, January 1992. To appear.

[BCM87] R. Bagrodia, K.M. Chandy, and J. Misra. A message-based approach to discrete-event simulation. *IEEE Transactions on Software Engineering*, SE-13(6):654–665, June 1987.

[BL90a] R.L. Bagrodia and Wen-toh Liao. Maisie: A language and optimizing environment for distributed simulation. In *Proceedings of 1990 SCS Multiconference on Distributed Simulation*, pages 205–210, San Diego, California, January 1990.

[BL90b] R.L. Bagrodia and Wen-Toh. Liao. Parallel simulation of the sharks world problem. In *Proceedings of the 1990 Winter Simulation Conference*, pages 191–198, New Orleans, Louisiana, December 1990.

[BLU90] Dirk Baezner, Greg Lomow, and Brian W. Unger. Sim++: The transition to distributed simulation. In *Proceedings of 1990 SCS Multiconference on Distributed Simulation*, pages 211–218, San Diego, California, January 1990.

[CS89a] K.M. Chandy and R. Sherman. The conditional event approach to distributed simulation. In *Proceedings of 1989 SCS Multiconference on Distributed Simulation*, pages 93–99, Tampa, Florida, March 1989.

[CS89b] K.M. Chandy and R. Sherman. Space-Time and Simulation. In *Distributed Simulation Conference*, pages 53–57, Miami, March 1989.

[CS89c] B.A. Cota and R.G. Sargent. Automatic lookahead computation for conservative distributed simulation. Technical Report CASE Center No. 8916, Simulation Research Group and CASE Center, Syracuse University, New York, December 1989.

[Fuj88] R. Fujimoto. Lookahead in parallel discrete event simulation. In *International Conference on Parallel Processing*, August 1988.

[Fuj89] Richard M. Fujimoto. Time Warp on a shared memory multiprocessor. *ACM Transactions of Society for Computer Simulation*, 6(3):211–239, July 1989.

[Gaf88] Anat Gafni. Rollback mechanisms for optimistic distributed simulation systems. In *Proceedings of 1988 SCS Multiconference on Distributed Simulation*, pages 61–67, San Diego, California, February 1988.

[GL90] I. Greenberg, A.G. Mitrani and B. Lubachevsky. Unbounded parallel simulations via recurrence relations. In *1990 ACM SIGMETRICS Conference on Measurement and Modeling of Computer Systems*, pages 1–12, 1990.

[GMRR89] D.H. Gill, F.X. Maginnis, S.R. Rainier, and T.P. Reagan. An interface for programming parallel simulations. In *Proceedings of 1989 SCS Multiconference on Distributed Simulation*, pages 151–154, Tampa, Florida, March 1989.

[JBWea87] D. Jefferson, B. Beckman, and F. Wieland et al. Distributed simulation and the time warp operating system. In *Symposium on Operating Systems Principles*, Austin, Texas, October 1987.

[Jef85] D. Jefferson. Virtual Time. *ACM TOPLAS*, 7(3):404–425, July 1985.

[Jha91] Vikas Jha. Performance evaluation and optimization of the conditional event simulation algorithm. Technical report, Computer Science Department, UCLA, Los Angeles, CA 90024, 1991. In preparation.

[KR88] B.W. Kernighan and D.M. Ritchie. *The C Programming Language, 2nd Edition.* Prentice-Hall, 1988.

[LL90] Yi-Bing Lin and Edward D. Lazowska. Exploiting lookahead in parallel simulation. *IEEE Transactions on Parallel and Distributed Systems*, 1(4):457–469, October 1990.

[Mis86] J. Misra. Distributed discrete-event simulation. *ACM Computing Surveys*, 18(1):39–65, March 1986.

[Nan81] Richard E. Nance. The time and state relationships in simulation modeling. *CACM*, 24(4):173–179, April 1981.

[Nic88] D.M. Nicol. Parallel discrete event simulation of FCFS stochastic queueing networks. In *Parallel Programming: Experience with Applications, Languages and Systems*, pages 124–137. ACM SIGPLAN, July 1988.

[Pre89] Bruno R. Preiss. The Yaddes distributed discrete event simulation specification language and execution environments. In *Proceedings of 1989 SCS Multiconference on Distributed Simulation*, pages 139–144, Tampa, Florida, March 1989.

[Rey82] P. Reynolds. A shared resource algorithm for distributed systems. In *9th International Symposium on Computer Architecture*, pages 259–266, Austin, Texas, 1982.

[Rey88] Paul Reynolds. A spectrum of options for parallel simulation. In *Proceedings of the 1988 Winter Simulation Conference*, pages 325–332, San Diego, California, December 1988.

[Sch86] H. Schwetman. CSIM: A C-based, process-oriented simulation language. In *Proceedings of the 1986 Winter Simulation Conference*, pages 387–396, Washington, DC, December 1986.

[Sei85] C.L. Seitz. The cosmic cube. *CACM*, 28(1):22–33, January 1985.

[SS89] Wen-king Su and C.L. Seitz. Variants of the Chandy-Misra-Bryant distributed simulation algorithm. In *1989 Simulation Multiconference: Distributed Simulation*, pages 38–43, Miami, Florida, March 1989.

[Ste91] Jeff Steinman. SPEEDES: Synchronous parallel environment for emulation and discrete event simulation. In *the Proceedings of 1991 SCS Multiconference on Advances in Parallel and Distributed Simulation*, pages 95–103, Anaheim, California, January 1991.

[WM88] Joel West and Alasdar Mullarney. ModSim: A language for distributed simulation. In *Proceedings of 1988 SCS Multiconference on Distributed Simulation*, pages 155–159, San Diego, California, February 1988.

A Maisie Syntax

The Maisie syntax is an extension of the C programming language grammar described in [KR88]. In the following description, undefined symbols have the same specification as in [KR88] and optional symbols are subscripted with $_{opt}$ or enclosed in $[\ ]_{opt}$. For brevity, the description does not include the syntax for the optimization constructs discussed in section 5.

maisie-program:
    translation-unit
translation-unit:
    external-declaration
    translation-unit external-declaration
external-declaration:
    declaration
    function-definition
    entity-definition
declaration:
    declartion-specifiers init-declarator-list$_{opt}$;
declaration-specifiers:
    type-specifier declaration-specifiers$_{opt}$
    storage-class-specifier declaration-specifiers$_{opt}$
    type-qualifer declaration-specifiers$_{opt}$
type-specifier:
    Maisie-type-specifier
    C-type-specifier
maisie-type-specifier:
    ename
    clocktype
    message-specifier
message-specifier:
    probe$_{opt}$ **message** *ident* [{ *struct-declaration-list* }]$_{opt}$
entity-definition:
    entity *entity-declarator compound-statement*
    extern **entity** *ident*
entity-declarator:
    ident { *ident-list*$_{opt}$ } *declaration-list*$_{opt}$
new-statement:
    [*unary-expr* =]$_{opt}$ **new** *ident* { *argument-expr-list*$_{opt}$ } [**at** *expr*]$_{opt}$
invoke-statement:
    invoke *expr* **with** *ident* = *expr*
    invoke *expr* **with** *ident* { *argument-expr-list*$_{opt}$ }
hold-statement:
    hold (*expr*)
wait-statement:
    wait *expr*$_{opt}$ [**until** *resume-compound-statement*]$_{opt}$
resume-compound-statement:
    resume-statement
    { *declaration-list*$_{opt}$ *resume-statement-list* }
resume-statement-list:
    [*resume-statement-list* **or**]$_{opt}$ *resume-statement*
resume-statement:
    resume-condition statement

resume-condition:

[*resume-condition* **and**]$_{opt}$ [*ident* =]$_{opt}$ `mtype` (*ident*) [`st` (*expr*)]$_{opt}$ *ranker*$_{opt}$

ranker:

`max` *ident*

`min` *ident*

trace-statement:

`trace` *expr* [`when` *expr*]$_{opt}$