A Methodology for the Decomposition of Discrete Event Models for Parallel Simulation

SJE Taylor, SC Winter and DR Wilson

University of Westminster, UK

Abstract

Parallel Simulation has presented the possibility of performing high speed simulation. However, when attempting to make a link between the requirements of Parallel Simulation and Discrete Event Simulation used in commercial areas such as manufacturing a major problem arises. This lies in the decomposition of the simulation into a series of concurrently executing objects.

Using the activity cycle diagram simulation technique as an illustrative example, this paper suggests a solution to this decomposition problem. This is discussed within the context of providing a conceptually seamless methodology for translating simulation models into a form which can exploit the benefits of parallel computing.

1: Introduction

The goal of executing a simulation on a parallel computer is to decrease the time taken for results to be obtained from the simulation. The motivation for this is the acquisition of data from which meaningful information can be generated for use in the decision making process, such as scheduling decisions in a factory. In cases where the model is large and complex, parallel processing techniques may be the only means by which this information can be generated faster than real time.

The field of research dedicated to the development of simulations which efficiently exploit the processing power of parallel computers is called *parallel simulation* [1]. However, this research does not address the correct translation of the model to be simulated into a form which can be executed using parallel simulation techniques. This is seen as a critical omission from parallel simulation methodology as there is no way of validating the relationship between model and implementation. To facilitate this validation process, the paper describes an intermediary stage which separates the implementation of the model from optimisation issues. This is the focus of work performed by other researchers at the Centre for Parallel Computing [2].

2: Parallel simulation

A discrete event model consists of a state, which changes over time, and events, which describe how this state can change. In a simulation program, the model state become a data structure, and events become procedures called *event routines*. Alternatively, events can be combined to become activities or processes depending on the conceptual framework selected [3]. Combining this with an event list, to schedule future state changes, and a simulation clock allows an algorithm called the simulation executive to simulate the model's progress through time [4].

To translate this seemingly monolithic structure into a form which can potentially exploit the multiple processors of a parallel computer, requires that the above be decomposed into a series of computational objects which communicate via message passing [1].

Take the example of a queuing network. Such a network contains nodes, consisting of a queue and a

1066-6192/92 $3.00 © 1992 IEEE

server, which are capable of sending jobs to, and receiving messages from, other nodes in the network. The generation of the objects required for parallel simulation is simple; each node of the model becomes an object. When a node receives a job, event routines model the service of the job and its subsequent arrival at another node by calculating the length of time this takes and scheduling events on the event list.

In this object based scheme, however, global structures such as the event list are not permitted. To permit each object to simulate the node that it represents, each object must contain its own clock, event list and simulation executive. Each object also contains its own state and the event routines required to simulate the object's state changes. To schedule the occurrence of an event at another object, an object sends a timestamped event message to the affected object. The arrival of a job at a node is therefore simulated in this parallel scheme by the corresponding objects sending and receiving a timestamped event message.

The timestamp of an event message allows a receiving object to process the event in the correct order. In a parallel computer it is entirely possible that these messages may arrive out of sequence. In the development of a protocol to correctly implement a discrete event model, the *local causality constraint* (LCC) allows the assumption to be made that all timestamped messages sent to an object arrive in the correct order. When it has been shown that the objects of the parallel simulation conspire to produce the same results as that of the sequential simulation, a *causality maintenance protocol* (CMP) is added to fulfil the role of the LCC. In parallel simulation there are two classes of CMP: *conservative* [5] and *optimistic* [6].

This separation of coding facilitates the development and validation of parallel simulations by focusing on the actual simulation performed by each object rather than a mechanism to order messages arriving at an object.

3: Model development

In discrete event simulation the queuing network model is one of the most simple modelling techniques. To demonstrate the need for a methodology within the context of an existing modelling technique, *Activity Cycle Diagrams* (ACD) [7,8], used extensively in the simulation of manufacturing systems, was selected. Briefly, the physical entities of a system, its jobs, machines, etc., are perceived to pass through an alternating series of active states and idle states. Idle states are represented as queues, active states as activities. Entities participate in activities and wait in queues. For example, a machine and job participate in an activity called *process*, but wait for the activity to begin in queues JQ1 and OQ respectively. An example of an ACD model is shown in figure 1.

In the model there are three entity classes *job*, *operator* and *machine*. As illustrated in figure 1, a job arrives, waits for an operator and machine to become available to process it and then goes on for inspection. At inspection there is a chance that the processed job has some defect that requires that the job must be reprocessed. If this is the case, the job is passed back for processing, otherwise the job leaves the shop.

Each time a machine is used, there is a chance that the machine might breakdown. If this occurs the machine undergoes repair and, once the task of repairing is finished, is returned for use. Machines are checked whenever one is required for processing.

To generate a simulation program from such a model, the implementor is guided by a set of rules defined by a *conceptual framework* [3]. In so called traditional simulation, there exists three frameworks; event oriented, activity oriented and process oriented. The choice of implementation framework is very much dependent on the modelling technique used and the model itself. ACD simulations can be generated according to any of the three. For purposes of this paper, the event orientation will be used.

The behaviourial logic of the model in the event orientation is decomposed into a collection of *event classes*. These form procedure blocks called *event*

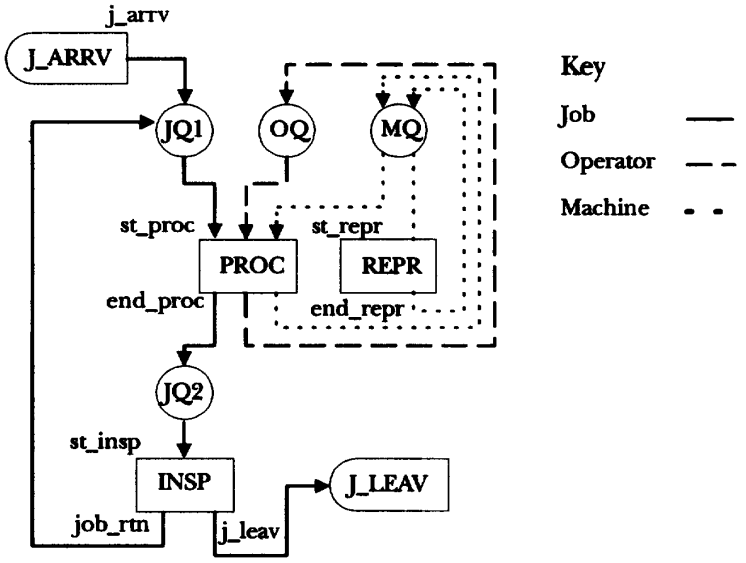

Figure 1 Example ACD Model

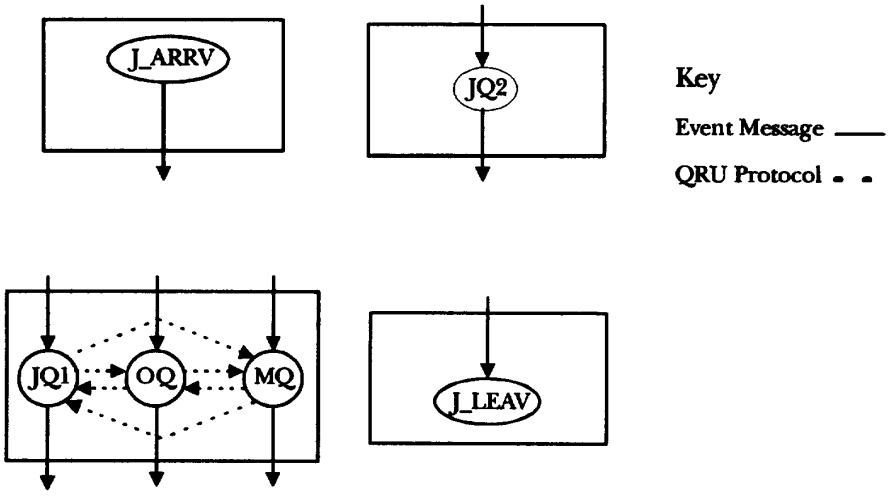

Figure 2 Event Oriented Parallel Simulation Structure of ACD Model

routines and gives this orientation *locality of time* [9]. The simulation executive used to advance the model through time is called the *event scheduling executive* and is shown in table 1. An example of an event routine generated from the job shop model is shown in table 2. Note that the ordering of conditional events is due to prioritising of activities specified in the model (ie. REPR, INSP then PROC).

```
WHILE NOT terminated DO
        FIND next event and advance simulation time
        to next event time
        EXECUTE next event
ENDWHILE
```

Table 1: Event scheduling executive

```
end_proc:    ADD job TO JQ2;
             ADD operator to OQ;
             ADD machine to MQ;
             st_repr;
             st_insp;
             st_psg.
```

Table 2: Example event routine (*end_proc*)

Given the bounds of the event orientation, the aim of this methodology is therefore to generate the objects of a parallel simulation of the model which is consistent with the event orientation.

4: Model decomposition

The event orientation demands the translation of the ACD model in terms of bound events, events which are time dependent, and conditional events, events which are dependent on a set of state conditions. Analysis of the ACD model defines their corresponding event routines.

Observing the LCC allows the CMP aspects of the simulation to be ignored, thus making it possible to focus on model decomposition issues. A suitable basis for the objects used by parallel simulation is now required.

To remain consistent with the strong queuing structure of the ACD model and to avoid additional overhead, the queues of the ACD model form the objects of the event oriented parallel simulation. This is also important in maintaining a strong relationship between model and implementation. Note that in an *activity* oriented simulation, the activities also form objects as well as queues due to the different requirements of that conceptual framework.

For each object, assuming that incoming messages are placed on an event list, the algorithm of table 1 provides the basis for correct event execution. Translating event interaction into message passing framework is more complex due to the existance of events which affect several objects. The effect of this distributed affect is now discussed for each type of event.

4.1: Bound events

A bound event, any event with a time dependent element, in a parallel simulation is scheduled by the transmission of a timestamped event message from one object to another. In this structure a bound event represents the movement of entities from one queue to another. Bound events are timestamped messages sent between objects and contain a reference to the event that they represent and any entities that are to be transferred. Each object will contain the corresponding event routine so that when an event message arrives, the event can be correctly simulated.

Consider the bound event *end_proc* which affects the queues JQ2, OQ and MQ. When a member of this event class is executed, the entities of the three classes are added to queues JQ2, OQ and MQ respectively. The arrival of a job in JQ2 requires that the inspection activity begins. Similarly, the availability of the operator and machine entities allows the possibility of another job to be processed. In the parallel simulation, the

effects of the single *end_proc* event are felt over *several* objects. This gives rise to *end_proc* being known as a *distributed bound event* (DBE).

A DBE is implemented by decomposing it into events local to an object, several *localised bound events* (LBE), one for each entity class affected by the DBE. A DBE therefore becomes several LBEs which are sent to each object that represents the next queue in the corresponding activity cycle. In its conversion into LBEs, *n* LBEs will be therefore generated.

Consider the derivation of the distributed event routine. The original event routine for *end_proc* resulted in the conditional events *st_repr*, *st_insp* and *st_proc* being attempted in an order dictated by the ACD model. The order in which *st_repr* and *st_insp* are attempted will not effect the validity of the simulation as long at the duration of the activity INSP is greater than zero.

In the parallel simulation the ordering of the conditional event execution also has to be enforced. Based on analysis of the scope of the conditional event (ie. the queue states required for the event to be tested) these events can be distributed across the localised elements of the DBE. These elements are shown in table 3.

end_proc at JQ1:	ADD job TO JQ2; st_insp.
end_proc at OQ:	ADD operator TO OQ; st_proc.
end_proc at MQ:	ADD machine TO MQ; st_repr; st_proc.

Table 3: Localised elements of distributed bound event (*end_proc*)

4.2: Conditional events

Conditional events in the parallel simulation generate the event messages which are sent and received by each object. Distribution also presents a problem to these events. Consider *st_proc*. This event occurs whenever a state change in the simulation makes it possible for the activity PROC to begin and is dependent on the states of the queues JQ1, OQ and MQ. It is clear from the ACD model that this event will occur as a consequence of the bound events *j_arrv*, *end_repr* or *end_proc*. If the execution of this event is successful, then the event *end_proc* will be scheduled.

As it is possible for *st_proc* to be instigated by the arrival of a job, operator or machine in the queues JQ1, OQ and MQ respectively, a parallel implementation must enable the conditional event to be instigated from any of these queues. This gives rise to such an event being termed a *distributed conditional event (DCE)*. As with bound events, a conditional event concerning only a single queue, is termed a *localised conditional event (LCE)*. A DCE is identified by a test involving several entity classes while a LCE is identified a test on a single entity class.

The implementation of a DCE is more complicated than a DBE as there is no supporting communication mechanism. Consider the execution of the DBE *end_proc* at JQ1. This causes job to be added to the state JQ1 and the event *st_proc* to be executed. The conditional test of *st_proc* requires the combined states of JQ1, OQ and MQ. This is further complicated when the ACD model is referenced; activity priority dictates that REPR must be attempted before PROC.

To make the conditional test, a mechanism must be provided to allow an object instigating the

DCE to determine whether or not the event can occur. To do this a protocol must be set up between the objects taking part in the test. This is a three stage message exchange and is referred to as the *query-reply-update (QRU) protocol.*

In the QRU protocol, an object instigating a DCE sends timestamped *query* messages to other objects referenced in the conditional test of the DCE. The timestamp of the query message allows these objects to synchronise to the correct point in time at which the DCE can be executed correctly.

On synchronisation, the receiving object can perform one of two actions. It can either return details of its current state or, if such exist, execute other conditional events *and then* return details of its current state. This allows the correct ordering of events.

State information is returned in the form of a *reply* message. The instigating object will then be able to carry out the conditional test of the DCE. It is assumed that the instigating object will have information regarding the queue disciplines of participating queues. This information can form either part of the reply message or be a permanent part of the instigating object.

Once the DCE has been evaluated, the instigating object will have zero or more sets of entities have begun an activity and a set of times at which they are due to end the activity; the instigating object can now schedule the corresponding bound events. It follows that the bound event marking the end of this cooperation will be a *DBE*.

To remain homogenous, the instigating object only schedules the DBE component for its own entity class. The other participating objects are responsible for scheduling their own DBE component. An *update* message is therefore returned to the cooperating objects so that they may update their local state (remove entities now engaged in the activity) and send their own DBE components.

These messages are in the form

query(Event, Source_queue, Destination_queue, Time)

reply(Source_queue, Destination_queue, Current_state)

update(Source_queue, Destination_queue, Update_set)

Where *Current_state* is the current state of a queue at a given timestamp *T* and is in the same form as *Entity_set*,

Update_set is a set of information from which a queue's state can be updated and event messages sent.

Note that regardless of where DBE messages originate, a message of this kind is required so that participating objects can update their local states. The homogeneity of this approach retains the clarity that is consistent with the modelling approach described thus far.

Table 4 presents the derivation of the DCE *st_proc* and its corresponding event routines. As can be seen the table, the translation of a DCE into a object form generates a lot more executable code than its sequential counterpart. This becomes substantially greater when all the objects are considered. Table 4 also lists the mechanisms used by the QRU protocol for JQ1. For example, when the update message update(JQ1,MQ,U_{MQ}) is received by MQ from JQ1, a mechanism is required to remove machine entities from the state of MQ and then to fabricate event messages to be subsequently passed on. This is summarised as

SEND event(Event,Source,Destination,X,T)

where X and T are the representative entity and time components of each member of the *Update_set*.

Conditional events in the parallel simulation represent synchronisation points which cannot be avoided by using CMPs due to the requirement that the conditional tests are made with the object states at the same point in time. This prevents the asynchronous processing of

objects involved in the same DCE and limits parallelism to the simultaneous execution of objects successfully completing
a conditional event. This is quite a serious limitation due to the fine grain nature of most objects encountered in a discrete event simulation. Using this basis it is possible to form *partitions*, groups of tightly coupled objects. When these are implemented in an efficient form, each partition will form one processing element. The four partitions resulting from the model are shown in figure 2.

```
st_proc:  SEND query(st_proc,JQ1,OQ);
          SEND query(st_proc,JQ1,MQ);
          WAIT reply(OQ,JQ1,S_OQ);
          WAIT reply(MQ,JQ1,S_MQ);
          WHILE test condition (st_proc) TRUE DO
            EXECUTE st_proc
            /*
            calculates the duration of the activity
            PROC and any affect that this has on the
            participant entities job, operator and
            machine
            */
          ENDWHILE
          SEND update(JQ1,OQ,U_OQ);
          SEND update(JQ1,MQ,U_MQ);
          WHILE entities left in U_JQ1 DO
            SEND event(JQ1,JQ2,job,T_event)
          ENDWHILE
```

Table 4: Localised element of distributed conditional event for JQ1 (*st_proc*)

5: Multiple time distributions

As illustrated in the discussion of conditional events, one or more event messages are sent if a conditional event is successful. The timestamp increment of these messages is dependent on the duration of the activity being modelled.

Referring to the ACD, consider queue MQ. A machine entity resident in this queue can engage in either the activity REPR or the activity PROC. Depending on which activity the machine begins, the entity will arrive at the preceding queue (in this case MQ) at a time dependent on either of the time distributions of the activities REPR or PROC.

The implication of this to the parallel simulation is that the object representing the queue MQ will be capable of sending timestamped messages based on one of two time distributions. This means that event messages arriving at the proceeding object can *appear* to arrive in the wrong order.

In the model decomposition phase of the methodology this observation is irrelevant as the LCC is in effect. However, when the LCC is removed and the CMP added, this effect can have dire consequences. For example, in conservative protocols the ordering constraint on a link is violated.

6: Multiple event instigation

Multiple event instigation (MEI) is a direct consequence of DBEs and DCEs. Consider the execution of the DCE *st_proc* as instigated by a job arriving at JQ1. After the execution of this event the DBE *end_proc* will be scheduled; components of this event will arrive at JQ2, OQ and MQ. ie.

event(end_proc,JQ1,JQ2,job,T_{end_proc})
event(end_proc,JQ1,OQ,operator,T_{end_proc})
event(end_proc,JQ1,MQ,machine,T_{end_proc})

At JQ2 the event message will be processed at T and the state of the queue will be subsequently inspected. At MQ, the message will add machine to the state of MQ at T and instigate the LCE *st_repr*. This in turn will in turn instigate the DCE *st_proc*. Note that the time at which *st_proc* will occur is *T*. At OQ, on the processing of the message, operator will be added to the state of OQ and *st_proc* executed *also at T*. This leads to *st_proc* being instigated at T *twice*.

In the sequential model, the execution of *end_proc* will cause the activity PROC to begin for all sets of job, operator and machine in JQ1, OQ and MQ. Obviously, for the parallel simulation of the same model to be correct, the execution of the DBE *end_proc* must have the same effect. Clearly, this is

not the case; *both OQ and MQ will instigate the DCE st_proc simultaneously causing MEI to occur.*

The reason for MEI taking place is this. A DBE occurs as a result of multiple classes taking part in the same activity. The entities which have taken part in this activity will therefore arrive at their respective proceeding queues simultaneously. When an entity arrives in a queue a test is triggered to determine if any of the activities proceeding the queue can begin; the conditional events marking the start of these activities are tested. If any of these activities involve multiple classes, then the corresponding conditional event will be a DCE. Clearly, in the parallel simulation model developed so far, *if a DBE is scheduled at two or more queues which then take part in the same DCE, then MEI will occur.* This is also possible if activities exist with *zero durations*.

MEI can be identified on an ACD by consideration of post-activity activity cycles. If two or more entities participating in an activity can subsequently arrive at a subsequent activity *at the same time* then a MEI situation will arise. This can be identified automatically.

7: Conclusions

This paper has suggested how a discrete event model based on an existing modelling technique can be translated into a form which can exploit the potential benefits of parallel simulation. These observations form the basis of a wider methodology to the composition of valid simulations executing on parallel computers.

The use of techniques consistent with the conceptual frameworks used in the modelling technique is very important. This is because this form of simulation is aimed at the engineer, not the computer scientist. There already exist barriers to the use of simulation (perceived cost, investment of skills and resources for the future). It is hoped that the guidelines presented here to the use of parallelism within existing simulation techniques will not complement these problems, but add the benefits of speed up to a potentially beneficial technique.

It was identified that within this domain, partitions will result due to the synchronisation consequences of distributed conditional events. This will have an unavoidable effect on parallelism. This indicates that if the physical system being simulated requires a great deal of global state testing, then the parallelism in the corresponding parallel simulation will be limited.

The two problems inherent in the translation of this modelling technique were shown. One can be addressed at this stage, the other must be addressed during the addition of the CMP.

Both conservative and optimistic CMPs have been implemented successfully [10].

References

1. Fujimoto, R.M. 1990 "Parallel Discrete Event Simulation", *Communications of the ACM*, 33(10), 30-67.

2. Kalantery, N.; S C Winter, A P Redfern; and D R Wilson, 1992. "Performance Visualisation of Conservative and Time Warp Based Parallel Simulation" In *Proceedings of the 1992 European Simulation Multiconference*, York, UK. SCS.

3. Derrick, E.J.; B. Balci; and R.E. Nance. 1989, "A Comparison of Selected Conceptual Frameworks for Simulation Modelling, The Implementation of Four Conceptual Frameworks for Simulation Modelling in High Level Languages." In *Proceedings of the 1989 Winter Simulation Conference*, IEEE, 711-717.

4. Law, A.M. and Kelton, W.M., 1991, Simulation Modelling and Analysis (2nd ed), McGraw Hill, New York.

5. Chandy, K.M. and J. Misra. 1979, "Distributed Simulation of Networks", *Computer Networks*, 3, 105-113.

6. Jefferson, D.R. 1985. "Virtual Time" *ACM Trans. Prog. Lang. and Syst.*, 7(3), 404-425.

7. Carrie, A. 1986, *Simulation of Manufacturing Systems*, John Wiley & Sons., Chichester, UK.

8. Pidd, M. 1992, *Computer Simulation in Management Science*, John Wiley & Sons., Chichester, UK.

9. Overstreet, C.M. 1987, Using Graphs to translate between world views, Proc 1987 WSC, Thesen A., Grant H. and Kelton W.D. (eds), 582-589.

10. Taylor, S.J.E. 1992, *The Development of Parallel Processing In Manufacturing Systems Simulation*, PhD Thesis, Leeds Polytechnic (in preparation).