Proceedings of the 1998 Winter Simulation Conference
D.J. Medeiros, E.F. Watson, J.S. Carson and M.S. Manivannan, eds.

ALTERNATIVE IMPLEMENTATIONS OF MULTITRAJECTORY SIMULATION

John B. Gilmer Jr.
Frederick J. Sullivan

Wilkes University
P.O. Box 111
Wilkes-Barre, PA 18766, USA

ABSTRACT

Multitrajectory Simulation allows random events in a simulation to generate multiple trajectories and explicitly manage the set of trajectories. The original prototype combat simulation used to demonstrate and test the concept used code embedded in the functional modules, e.g. those that implement "move", "shoot", etc. A much improved method provided a class library that hid the messy details. When a random choice is made, a random choice method appears to return twice (or more) for a given call, once in the context of the original state, and once each for the new trajectories. These techniques both have significant shortcomings. For example, the second (more preferable) technique really needs to be able to overwrite the C++ "this" variable, an option unavailable on C++ compilers. Since these issues surfaced, three additional implementation techniques have been developed. These include periodic copying of states to provide reference copies and duplication of trajectories up to the branch point, reformulation of the simulation into a discrete event style, and reformulation into a tail recursive style. We review these techniques. Each has advantages and disadvantages. Multitrajectory simulation is not dependent on the particular limitations of any one method.

1 BACKGROUND

The goal of multitrajectory simulation is to explore the outcome space of a simulation, that is, the set of all possible outcomes, more systematically and less expensively (for a given quality of understanding) than can be achieved with conventional stochastic simulation. In some senses this could be considered a variance reduction technique, but the analysis goals may be formulated not only in terms of better estimates of statistical properties of the outcome set, e.g. a mean and variance for a Measure of Effectiveness(MOE), but also representative instances of extreme behavior or other "interesting" cases.

The heart of the proposed method is to explicitly track each possible trajectory, as illustrated in Figure 1. When an event that would normally be stochastic occurs, instead of one outcome, multiple outcomes are generated, each constituting a trajectory having its own state. In concept, such a multiple trajectory simulation is integrated with its support system in such a way that its use provides outcomes with probabilities associated with each, an accounting for the key events or circumstances leading to the differences, and some measure of confidence in these results.

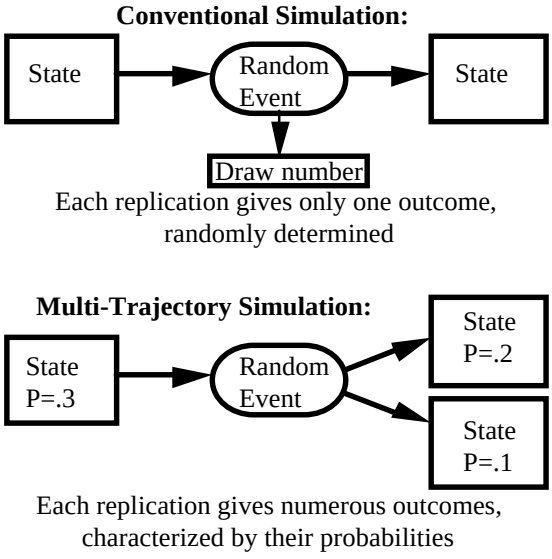


Figure 1: Concept for Explicitly Tracking Trajectories

This conceptual ideal must be compromised in various ways if the approach is to be practical: Discrete distributions must be used; in practice, choices in the military simulation domain of interest have usually been binary, for example whether a target is acquired or not. Only some types of events that might be treated with the multitrajectory technique actually are resolved that way in a given run, to limit the focus of investigation and effort. Fnite resources and the potential exponential explosion in the number of trajectories makes necessary some form of

management in which not all trajectories are followed. This is no worse than limiting the number of replications for stochastic simulation. Indeed, by having a systematic way of managing the treatment of uncertainty in the simulation support software, the analysis can be better and more efficiently tailored, without changing the model software proper (Gilmer and Sullivan, 1996).

Two techniques have been used which trade off coverage for the sake of keeping resources bounded. One is the "truncation" management technique that explicitly decides, for each event in some trajectory, whether to resolve the event in multitrajectory fashion (resulting in the creation of a new trajectory) or to instead resolve it deterministically or stochastically. This is illustrated in Figure 2.

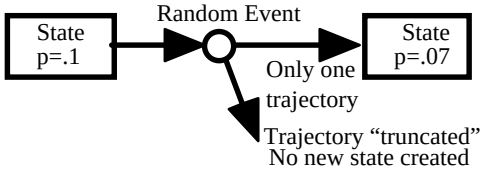


Figure 2: Trajectory Truncation

The second approach is to look for and consolidate states that are "similar", where similarity is judged by a metric that estimates the sum of differences between two given states, as shown in Figure 3. During an analysis "run", simulation trajectory management sacrifices some trajectories, perhaps those of lowest probability or of least interest with respect to some MOE, for the sake of keeping the number of trajectories within bounds.

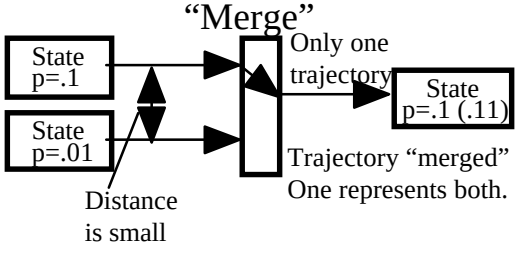


Figure 3: Trajectory Merge, or Consolidation

2 SAMPLE MULTITRAJECTORY SIMULATION

Much of this research has been conducted using a simplified, unclassified surrogate for the military simulations of interest. The simulation "eaglet" was designed to resemble the Corps level simulation "Eagle" in important respects, but to be of manageable simplicity. It includes Lanchester square law combat, movement by nominally battalion sized units along routes with multiple paths, decisionmaking, and artillery support. Figure 4 illustrates the smallest scenario we have used with "eaglet". Two Blue units attack a Red unit. A second Red unit counterattacks from the flank.

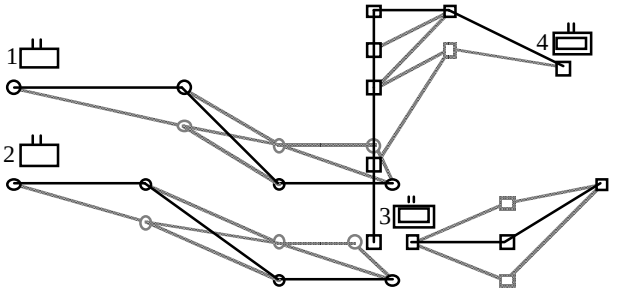


Figure 4: "eaglet" Scenario with Multitrajectory Routes

Note that the route objects show multiple paths; when there are multiple links from a given node, a multitrajectory event occurs for the choice of which path to follow. Figure 5 illustrates the process of creating a new trajectory when this happens.

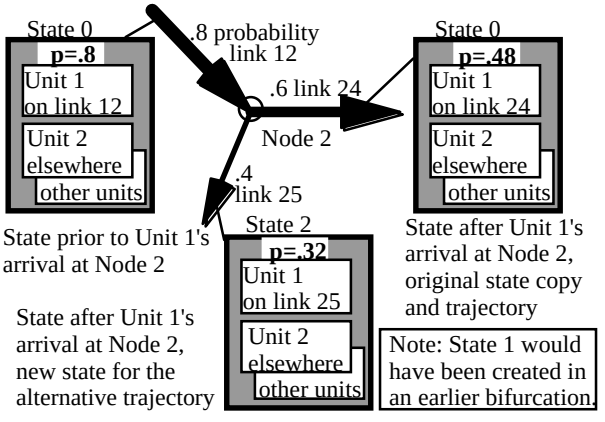


Figure 5: New Trajectory Creation for a Move Event

Multitrajectory attrition (variations in combat losses), decisionmaking (whether a decision rule fires), acquisition (whether a unit sees another) and acquisition loss have also been implemented. Figure 6 illustrates some of the results from previous research showing how the multitrajectory approach compared to stochastic runs, for the case of movement events (only) being resolved in multitrajectory fashion. The shading of squares is darker with increasing sum of the probabilities of states having Measures of Effectiveness values within that square. In this limited scenario, the multitrajectory approach can be exhaustive. As the number of stochastic runs increases, the multitrajectory distribution is approached.

With larger scenarios and a greater variety of multitrajectory events, it is not possible to exhaustively develop the outcome set. Multitrajectory management

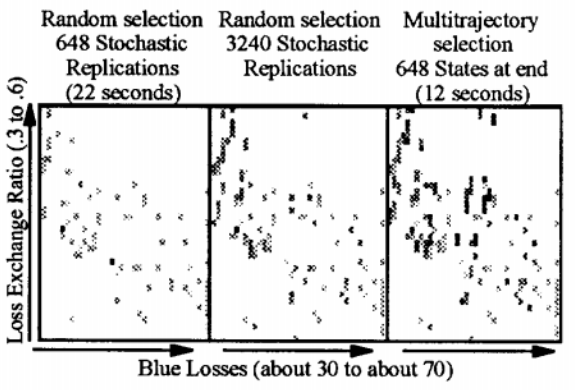


Figure 6: Small Scenario Outcomes for "eaglet"

becomes more important. Figure 7 illustrates this for a larger scenario. In this case, the simplest management approach is used: all events are multitrajectory until a fixed limit is reached, then events are resolved deterministically. Event outcomes of lower probability that give interesting results come too late to be captured, while stochastic resolution at least gives a sampling of these outcomes. The multitrajectory outcome set has a total probability higher than that of the stochastic set, but is not representative.

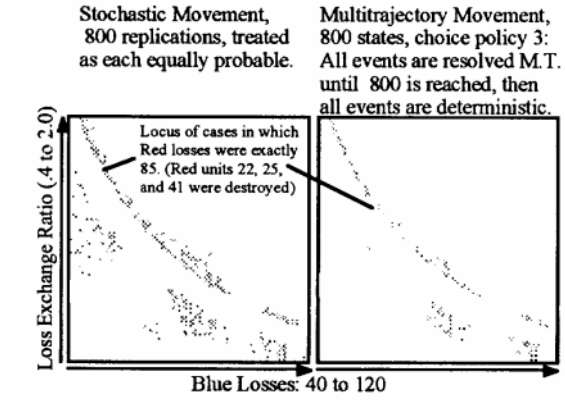


Figure 7: Larger Scenario Outcomes with an Inadequate Trajectory Management Technique

Figure 8 shows outcomes for this same scenario with a better management technique, and for several types of event. Improved management techniques that are sensitive to model Measures of Effectiveness have been developed (Al-Hassan, Gilmer, and Sullivan, 1997). This allows the decision on whether to generate or truncate a new trajectory to be based on how "interesting" it is, in the sense of whether it has an extreme value of some MOE. Research into scaling issues and improvement of management techniques is underway, but is outside the scope of this paper.

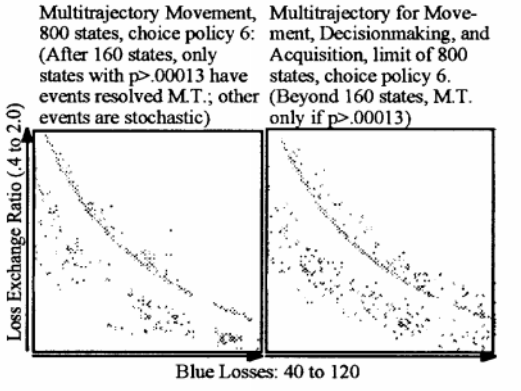


Figure 8: Larger Scenario Outcomes with Improved Trajectory Management

3 THE APPROACHES

This section gives an overview of the various multitrajectory techniques. In choosing among these techniques, the issues are efficiency, transparency, and feasibility. Efficiency concerns the resources (time and memory) consumed when the method is operating. Transparency concerns the degree to which the programmer (of the simulation functional modules) must be conscious of the multitrajectory approach and take it into account when coding. For all methods, it is necessary to explicitly recognize stochastic events as decisions rather than merely random number draws. Beyond that, the degree to which the code may have to be convolved to fit the requirements of the method differ. Feasibility concerns whether the technique can be implemented for a given machine and language.

3.1 "Gilmer's method"

This was the original implementation of multitrajectory simulation. It is included for the sake of completeness.

When an event takes place, it is likely to be deeply embedded in nested functions. The new state is created as an image of the old one at the event, deep in the calling hierarchy. But the new state will have been only partially processed for the time step (or event) that was being processed. The time step must be finished. This re-entry is explicitly programmed. So, at the beginning of each routine, an if-test is needed to determine if this is an original entry or a re-entry for a partially processed state. If the latter local variables such as counters must be initialized to values appropriate to the state of the routine when the event took place. Figure 9 illustrates.

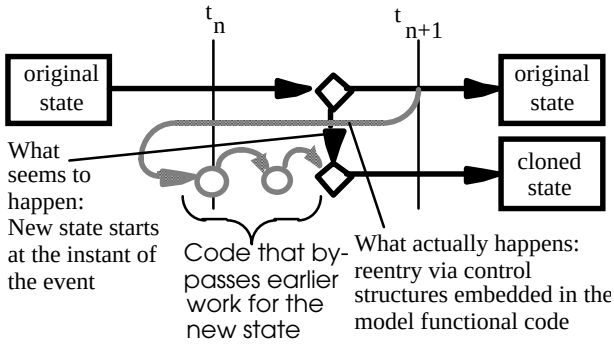


Figure 9: Processing Events Using "Gilmer's Method"

The structure of the code is essentially unchanged from what might be written by a programmer unaware of multitrajectory techniques, but must have the control structures to choose the method for event resolution, and to manage the reentry. Figure 10 illustrates.

The actual code is much, much messier. This example does not show more complicated resolution methods such as those that check the state limit, MOE's, or run in an "event following" mode (which follows the event record trace of an earlier run). This method can be improved by using choice policy routines to determine resolution method for a given event. There appears to be no feasibility problem. This method is less preferable than those which hide messy choice and reentry details.

3.2 "Sullivan's Method":

This technique was developed by Sullivan as a part of the original project to investigate the multitrajectory technique (Sullivan and Gilmer, 1996).

```
Unit::lose_aquisition(State *ps){
  (various declarations, etc.)
  //Handle the reentry case: The state has reentry info.
  if(ps->status_unit()==Id&&
     ps->status_event()==LOSE_ACQ)
         i=ps->status_itteration();
  else i=0;
  int n=N_acquired;
  for(;i<n;i++){
    if(lose_acq_evt==DETERMINISTIC){
      Acquisition_list[i]=0;
      N_acquired--;}
    else if(lose_acq_evt==STOCHASTIC){
      rand_num=rand()/32768.0;
      if(rand_num<pct_lose){
        Acquisition_list[i]=0;
        N_acquired--;}}
    else if(lose_acq_evt>=MULTITRAJECTORY){
      if(ps->status_event()==LOSE_ACQ&&
         ps->status_unit()==Id&&
         p_s->status_itteration()==i){
        Acquisition_list[i]=0;
        N_acquired--;
        ps->create_status(0,Id,0);}//reset status
      else{
        p_new_state=new State(ps,pct_lose);}}}
```

Figure 10: Example of Code for "Gilmer's Method"

When a multitrajectory event takes place, a copy of the stack is made. When the new state is finally ready to be executed, the stack is reinitialized to duplicate the condition it was in when the state was created and the context saved with a call to setjmp. A call to longjmp then puts execution back into the simulation functional code at that point. This amounts to "throwing" a continuation. From the functional model programmer's perspective, it looks like the choice returns twice (or more) for events that are resolved with the multitrajectory technique. The messy details are hidden in base classes upon which the simulation is built. Figure 11 illustrates.

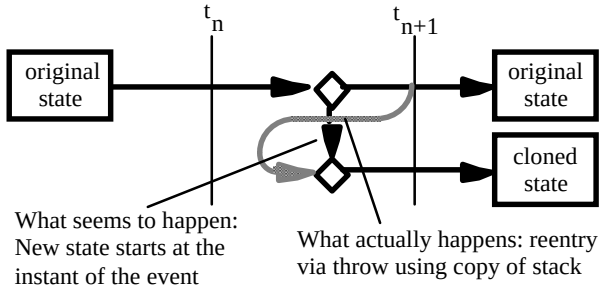


Figure 11: Origination of New Trajectories Using "Sullivan's Method"

The simulation functional code structure is very similar to that of a non-multitrajectory simulation, except for the use of choosers rather than random number generator calls. Figure 12 shows how the code looks for the same acquisition loss event as illustrated earlier.

Note the use of the variable "self" as a substitute for "this" in the references to Unit object members "Acquisition_list" and "N_Acquired" that follow the chooser call. Any pointers in the stack upon restoration are pointed to the wrong state, and need to be fixed. (The initial reference to "N_acquired" is not a problem, since no multitrajectory event can have occurred yet.) Thus, some discipline is necessary to ensure that pointers are fixed after a potentially multitrajectory event. (This we refer to as the "this problem", since the "this" variable is the best example of a pointer that no longer points to the correct place. Unfortunately, the C++ compilers that we are using (primarily g++) do not allow "this" to be modified.) The chooser and other messy details are hidden in base classes upon which the simulation object is built. Figure 13 illustrates this, which also applies to most implementation methods.

```
Unit::lose_aquisition(State *ps){
  (various other declarations, etc.)
  n=N_acquired;
  for(i=0;i<n;i++)
    DiscreteChoice ch[2]={{pct_lose,0},{1.-pct_lose,1}};
    Chooser C_Ch->Initialize (ch,2,LOSE_ACQ);
    Unit *self=this;
    choice = Sim.RandomChoice(C_Ch);
    self=ps->get_new_this(Id);
    if(choice ==0){
      self->Acquisition_list[i]=0;
      self->N_acquired--;}}}
```

Figure 12: Illustration of "Sullivan's Method"

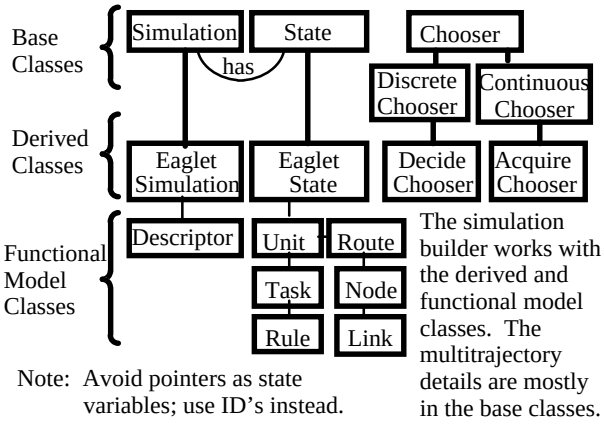


Figure 13: Classes for Multitrajectory Simulation

This technique is efficient, but the code to copy the stack and execute second return is typically machine platform dependent. Only a very small amount of machine dependent code is needed. (We use setjmp and longjmp, but must work around the limitation that longjmp is supposed to be used to jump out of, rather than into, nested function calls. This gets messy, but the mess is hidden in the base classes.) Furthermore, we have found no way to implement this technique in Java without changes to the virtual machine. The practicality of making such changes has not yet been investigated. Otherwise, this method would seem preferable to the others. This method is more fully described in an earlier paper (Sullivan and Gilmer, 1996).

3.3 "Koch's Method"

This method was suggested by John Koch of Wilkes University as a simpler alternative to Sullivan's method.

At some point prior to when multitrajectory events may occur, a copy is made of each state. This reference copy remains unchanged as the trajectory executes. The executing trajectory keeps a record, at each event, of the choices made. (This is generally done anyway to allow the option to re-play a particular trajectory.) When a multitrajectory event occurs, the reference state rather than the current state is copied, together with a record of events up until that point, and the choice to be made for the current event in the new trajectory. At a convenient time when the new trajectory begins to execute, it re-executes all events up until the creation event starting at the time the reference state was copied. Upon arrival at the creation event, it begins to follow a divergent path from the original state. It may, in turn, generate yet more trajectories in like manner. Figure 14 illustrates.

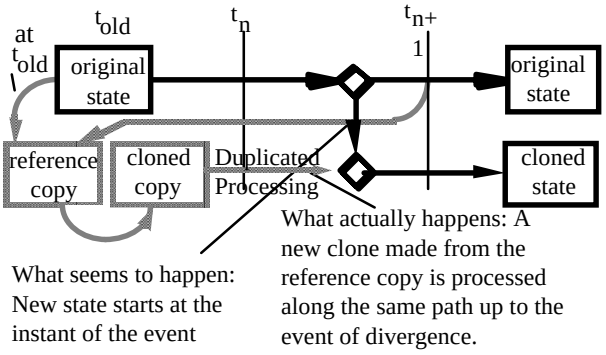


Figure 14: Illustration of Koch's Method

The frequency with which reference copies of states are made is a tradeoff. In the worst case, all go back to the original state. This approach is no worse than classical stochastic simulation, in which multiple runs start from a same initial state. (Indeed, it is in practice better since the cloning of states will typically be cheaper than creating an initial state from text or other input data.) With Koch's method no extra burdens are put on the functional code programmer, other than the use of "choosers" rather than random numbers and thresholds within the model code. There is no reentry or "this" problem. Thus, the code would look much like that of Figure 12, except that the use of "self" is not necessary.

However, this method does require the use of more memory. Indeed, one must periodically double the number of states by making reference copies. So this method can take up to twice as much memory for a given number of trajectories. It also requires expenditure of CPU time to make the copies, one of which, the reference copy, is wasted for each time step for each state. (One cannot ever used the reference copy as a clone, since there is always the possibility of another multitrajectory event that will need to refer to the reference copy.)

This method has not yet been prototyped, but is considered relatively low risk. The most important issue is how to record and later follow the event record, which is not an entirely trivial issue.

We have already been using event records as a way to generate "leftist" tree mode multitrajectory runs. In a "leftist mode" run, the event record from a trajectory generated by an earlier run is used to resolve events for the reference state (State 0). At each event, a new trajectory is

generated for the other possibilities. These new trajectories do not bifurcate further. They attempt to continue resolving events consistent with the outcomes in the event record; if synchronization is lost they become deterministic. Figure 15 illustrates. This mode allows, for a given trajectory, a comparison between the reference trajectory and alternatives that are similar except for the outcome of a given event. The technique would be similar in an implementation of "Koch's Method".

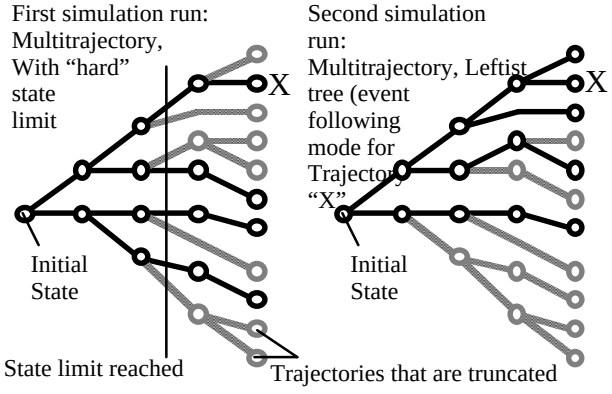
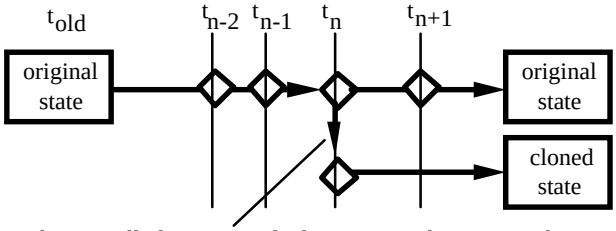


Figure 15: Event Following in "Leftist Tree" Mode

4. "Burlington Method"

This technique was prototyped for the "ACP" (Advanced Conceptual Prototype) simulation, developed by Ben Wise of SAIC, to prototype innovations in Command Control representation. Conversion of this simulation to add multitrajectory capability was performed by SAIC Burlington personnel as part of the "Course of Action Analysis" (COAA) project.

The concept, shown in Figure 16, is to design the simulation code so that potentially multitrajectory events directly correspond to events in the sense of that term in "Discrete Event Simulation" (DES). Events are dispatched only in the simulation executive



What actually happens and what seems to happen are the same, since the events of the simulation at the dispatch (DES) level are the multitrajectory events (and any that may be multitrajectory). Events embedded deeply in model code are prohibited.

Figure 16: Illustration of the "Burlington Method"

When a multitrajectory event occurs, additional states are created and the functional code is called for each outcome. The executive would include a dispatcher that would function, in effect, as shown in Figure 17. For simplicity, we show a version that only handles Boolean events. Note that for at least some kinds of events, the probability may have to be determined at the time of event execution, rather than when the event is scheduled.

```
State::Dispatch(Event *pe){
  State *ps;
  float r;
  int method=choice_method(pe);
  float probability=pe->eval_p(this,method);
  if(probability==0.)
    pe->Do_event(0,this,1.0);
  if(probability==1.)
    pe->Do_event(1,this,1.0);
  else if(method==STOCHASTIC){
    r=rand()/32768.0;
    if(r<probability)
      pe->Do_event(0,this,probability);
    else
      pe->Do_event(1,this,1.-probability);
  else if(method==MULTITRAJECTORY){
    ps = new State(this,1.-probability)
    modify_probability(probability)
    pe->Do_event(1,this,probability)
    pe->Do_event(0,ps,1.-probability)}}
```

Figure 17: Multitrajectory Discrete Event Dispatch

The limitation of this method is that the simulation must be entirely written in a discrete event style, with events scheduled for lightweight tasks such as perception trials, adding overhead in processing and memory usage to handle the events. Any events that may be multi-trajectory for some analysis must be treated this way.

Given that this method has been implemented for a different simulation, it has not been possible to make analytic comparisons to assess the cost of the additional overhead. There are no "this" problems or other known implementation barriers, other than the need to have a DES structure. One reason for implementing a time stepped "eaglet" was that this was regarded as the more difficult, and more general, problem.

3.5 "Sullivan's Prime (2nd) Method"

Sullivan developed this technique after failing to find an acceptable technique for implementing his original method in Java. It is similar to the original Sullivan's method in the use of a chooser object, but also has similarities to the Burlington method in that it forces a particular style for the simulation code. The central idea is that an event is controlled by a multichoice object which embodies all the choices which can be made. When an event occurs, a multichoice object is obtained, and each of its choices is then processed. Code which does the processing for each choice must come after the choice is obtained, and therefore the choice cannot be embedded in a loop. In general, therefore, loops must be implemented as

recursions, so that the continuation of the loop processing appears explicitly at the end.

An example is used to illustrate this. If we could write loops, the code would look as shown in Figure 18.

```
State::timeStep() {
  for (i = 0; i < UNIT_COUNT; i++) {
    for (j = 0; j < UNIT_COUNT; j++) {
      Multichoice acqMC = acquireChooser.getChoices();
      choice = acqMC.nextChoice();
      while (choice != null) {
        if (choice) {
          units[i].acquire(units[j]);}
        choice = acqMC.nextChoice();}}
    units[i].shoot();
    units[i].decide();
    if (units[i].atNode()) {
      Multichoice mvMC = moveChooser.getChoices();
      choice = mvMC.nextChoice();
      while (choice != null) {
        units[i].changeLink(choice);
        choice = mvMC.nextChoice();}}
    else {
      units[i].followCurrentLink();}}}
```

Figure 18: Conceptual model, Sullivan's Prime Method

In this example, we loop over the units to do target acquisition, combat, and decision making. We then loop over the units again to do movement. The chooser method getChoices returns successive choices each time it is called, and null when the choices are exhausted.

This doesn't work, because we don't finish the time steps for newly created states, since the continuation of the time step goes back to the top of the loop. Instead, we turn the loops into recursive calls and put the continuation inside the recursive call. See Figure 19. This example is then further transformed into the recursive style as shown in Figure 20.

```
State::timeStep() {
  i = 0;
  while (i < UNIT_COUNT; i++) {
    j = 0;
    while (j < UNIT_COUNT; j++) {
      Multichoice acqMC = acquireChooser.getChoices();
      ch = acqMC.acquireChooser.getChoices();
      while (ch != null) {
        if (ch.getValue()) {
          units[i].acquire(units[j]);}
        ch = acqMC.nextChoice();}}
    units[i].shoot();
    units[i].decide();
    if (units[i].atNode()) {
      Multichoice mvMC = moveChooser.getChoices();
      ch = mvMC.nextChoice();
      while (ch != null) {
        units[i].changeLink(ch.getValue());
        ch = mvMC.nextChoice();}}
    else {
      units[i].followCurrentLink();}}}
```

Figure 19: Sullivan's Prime Method, Example with Transformation to While Loops

```
State::timeStep() {
  processUnits(0);}

State::doUnits(int i) {
  if (i < UNIT_COUNT) {
    doOtherUnits(i, 0);}}

State::doOtherUnits(int i, int j) {
  if (j < UNIT_COUNT) {
    MultiChoice acqMC = acquireChooser.getChoices();
    doAcquireChoices(i, j, acqMC);}}

State::doAcquireChoices(int i, int j, Multichoice
acqMC){
  Choice ch = acqMC.nextChoice();
  if (ch != null) {
    State self = currentState();
    if (ch.getValue()) {
      self.units[i].acquire(self.units[j]);}
    if (j < UNIT_COUNT) {
      doOtherUnits(i, j + 1);}
    else {
      self.units[i].shoot();
      self.units[i].decide();
      if (self.units[i].atNode()) {
        Multichoice mvMC =
                  moveChooser.getChoices();
        doMoveChoices(i, mvMc);}
      else {
        self.units[i].followCurrentLink();
        doUnits(i + 1);}}
    doAcquireChoices(i, j, acqMC);}}

State::doMoveChoices(i, MultiChoice mvMC) {
  Choice ch = mvMC.nextChoice();
  if (ch != null) {
    State self = currentState();
    self.changeLink(ch.getValue());
    doUnits(i + 1);
    doMoveChoices(i, mvMC);}}
```

Figure 20: Sullivan's Prime Method with Recursion

The essential part of the transformation is that the continuation of an event (any code which is executed following the event) has to explicitly follow the event, rather than implicitly as in a loop. The advantage of the original Sullivan's Method is that continuations are handled by low-level programming magic, whereas in the Sullivan's Prime method, they must be handled by the programmer. Note that in Sullivan's (original) Method new states are reentered following completion of all existing states (although this doesn't have to be true), while in this method new states are reentered immediately following completion of the state that generated them, and thus resembles the Burlington Method as seen in Figure 16.

This recursive style would have to be pervasive throughout the simulation. Programmers used to an iterative style, or from outside the computer science discipline where recursion is highly prized, may have difficulty writing code in this style. Sullivan believes, but has not yet demonstrated with a working prototype, that it is possible to build a translator that will convert more usual style code into the required form for this method. Such a translator has been beyond the scope of research performed to date.

This approach to multitrajectory simulation will require a larger stack than would normally be the case.

(The fact that g++ implements tail recursion as a loop helps.) The "this" problem of Sullivan's Method can be present here as well, but can be dealt with by the same disciplines. This is a minor consideration compared to the programming style issues. No language dependen-cies would prevent a Java implementation. Efficiency should be similar to that of the other faster techniques.

4 CONCLUSION

All of the methods have in common the need to write the model functional code in a manner that makes clear that a choice is being made when a random event occurs. Even without multitrajectory techniques, the analyst gains explicit control over treatment of events, without having to directly embed control features in the model functional code. Choice policies can be standardized, or customized to apply different criteria for different kinds of units, circumstances, or resource usage. There is an explicit mechanism for calculating end state probability (given the scope of variability selected by the analyst).

Multitrajectory capability can be implemented with a variety of techniques, each having advantages and disadvantages. At this time Koch's method appears to be the most straightforward, but is wasteful of resources. Sullivan's method would be preferred except for the "this" problem complications and the lack of a Java implementation approach. If rewriting a simulation into a style that may differ from a conventional procedural approach can be tolerated, either the Burlington method or Sullivan's Prime method should be considered.

ACKNOWLEDGMENTS

This project continues research that was funded by the US Army Research Office under Grants DAAH04-95-1-0350 and DAAG55-97-1-0360, with the sponsorship of the US Army Concepts Analysis Agency. Mr. Gerry Cooper and Col.. Andrew Loerch have been very helpful in advice and assistance. We also thank Dr. Robert Alexander of SAIC, whose interest and support have been essential to the development of this research, and whose COAA team developed the "Burlington Method".

REFERENCES

Gilmer, John B. Jr., and Frederick J. Sullivan, 1996. Combat Simulation Trajectory Management. In *Proceedings of the 1996 Military, Government, and Aerospace Simulation Conference*, ed. Michael J. Chinni, 236-241. Society for Computer Simulation, San Diego, California.

Sullivan, Frederick J., and John B. Gilmer, Jr., 1996. Managing Multiple Trajectory Simulation. In *Proceedings of the High Performance Computing Conference*, ed. Adrian Tentner, 320-323. Society for Computer Simulation, San Diego, California.

Al-Hassan, Sadeq, John B. Gilmer Jr., and Frederick J. Sullivan, 1997. A Simulation State Management Technique Sensitive to Measures of Effectiveness. In *Proceedings of the 1997 Military, Government, and Aerospace Simulation Conference*, ed. Michael J. Chinni, 95-100. Society for Computer Simulation, San Diego, California.

Working papers and other documents, and simulation / analysis screen shots, can be found at http://calvin.mathcs.wilkes.edu/mts.

BIOGRAPHIES

JOHN B. GILMER, JR worked in the development of combat simulations, with a focus on C2 representation and parallelism, at BDM, Inc. He was the chief designer of the Corban combat simulation. He has a Ph.D. in EE from VPI and currently teaches Electrical and Computer Engineering at Wilkes University.

FREDERICK J. SULLIVAN teaches Computer Science at Wilkes University, and earlier did so at Rose-Hulman and SUNY Binghamton. His expertise is in operating systems and object oriented software. His Ph.D. is in Mathematics, from LSU.