A RAND NOTE

PSE: An Object-Oriented Simulation Environment Supporting Persistence

Stephanie J. Cammarata, Christopher Burdorf

RAND

The research described in this report was sponsored by the Defense Advanced Research Projects Agency under RAND's National Defense Research Institute, a federally funded research and development center supported by the Office of the Secretary of Defense and the Joint Staff, Contract No. MDA903-90-C-0004.

Published 1991 by RAND
1700 Main Street, P.O. Box 2138, Santa Monica, CA 90407-2138

A RAND NOTE

N-3385-DARPA

PSE: An Object-Oriented Simulation Environment Supporting Persistence

Stephanie J. Cammarata, Christopher Burdorf

Prepared for the
Defense Advanced Research Projects Agency

RAND

PSE: an object-oriented simulation environment supporting persistence[1]

by Stephanie J. Cammarata & Christopher Burdorf

The RAND Corporation, 1700 Main St., Santa Monica, CA 90407-2138 and School of Mathematical Sciences, Univerity of Bath, Bath, Avon BA2 7AY, England

This paper describes the Persistent Simulation Environment (PSE), which combines object-oriented simulation with a persistent object repository and domain-dependent object prefetching facilities. The goals of PSE are threefold: (1) to augment a contemporary object-oriented programming language with discrete event and process-based simulation facilities equaling those found in simulation languages such as Simscript and Simula; (2) to tightly couple an object-oriented simulation language with a secondary storage facility to achieve the persistence of simulation objects; and (3) to improve the swapping of persistent simulation objects between main memory and secondary storage through the use of object prefetching. The PSE prototype we developed is implemented in the Common Lisp Object System (CLOS) and runs in Allegro Common Lisp on Sun/3 and Sun/4 workstations. This environment is a complete, yet flexible, set of CLOS class definitions and methods fulfilling these objectives.

The results of this research will contribute to the Productivity Improvements in Simulation Modeling (PRISM) project supported by the Air Force Human Resources Laboratory. The goal of the PRISM project is to improve productivity and responsiveness of organizations within the Air Force that provide mission capability assessments through discrete event simulation models. The simulation facilities of PSE were modeled predominantly after those available in Simscript and Simula [Russe79, Dahl67]. In addition, we incorporated many traditional simulation features that were not supported in the Lisp-based Ross object-oriented simulation language such as probability distributions and process-based simulation [McArt84].

[1] This project was sponsored by the Air Force Human Resources Laboratory through the Defense Advanced Research Projects Agency under the auspices of RAND's National Defense Research Institute, a federally funded research and development center sponsored by the Office of the Secretary of Defense and Joint Chiefs of Staff. Views and conclusions contained in this document are those of the authors and should not be interpreted as representing the official opinion of DARPA, AFHRL, the U.S. Government, or any person or agency connected with them.

Persistent object systems (POSs), such as PSE, have the advantage that objects are no longer tightly coupled to the simulation system, i.e., objects reside in their own repository and can be independently perused before, during, or after a simulation session. Therefore, input and output simulation data may be maintained permanently in the persistent store of PSE. Moreover, persistent object systems enable object-oriented simulations to be scaled up to efficiently support and maintain many more objects than memory-based object-oriented languages. For example, large-scale simulations, such as those done at RAND, may contain thousands of objects. Our laboratory has generated 80,000+ map objects for terrain-based modeling. We find that up to 20,000 of these objects can be loaded into the CLOS environment on a workstation with 16mb of main memory before the virtual memory system will need to perform excessive paging to manage the size of the virtual image. Such excessive paging can greatly degrade the performance of the simulation. Our initial results indicate a fourfold speedup when reading 20,000+ previously formatted objects stored in our PSE object management system, compared to reading and formatting the same objects each time for non-persistent CLOS.

Many POS projects are concerned with seamless integration of simulation language features and traditional data management capabilities such as transaction management and multiuser access [Atkin87, Ford88, Khosh89]. Although these issues are critical to the success of persistent object systems, much of our efforts were focused on a different problematic aspect of POS: efficient access of persistent simulation entities from secondary storage. In a POS, persistent objects entail disk accesses when the simulation requires objects not resident in the simulation's virtual image. PSE incorporates techniques for reducing the number of "object faults" through object usage prediction and prefetching.

In the next section, we present the simulation facilities supported by PSE including examples that demonstrate the use of events, processes, and resources. The following sections address the persistent object system within PSE and describe the methodol-

ogy we developed for object prefetching. We discuss two PSE applications and identify limitations and future work in the final two sections.

SIMULATION CAPABILITIES IN PSE

PSE supports both event-based and process-based discrete simulation. Events are actions that occur instantaneously; processes are actions that have a time duration and that may or may not consume resources. Events are scheduled programmatically (or by the user) to occur at the current simulation time or at some time in the future. Processes are also scheduled to begin at a certain time; however, depending on the availability of necessary resources and the priorities of competing processes, their activation cannot always be predicted. Instead, the PSE scheduler controls the activation, interruption, reactivation, and termination of processes.

The event scheduling methodology and simulation primitives are based on those found in the Ross object-oriented simulation language. A global clock object maintains the scheduling and processing of events. However, because PSE is CLOS-based, PSE takes advantage of CLOS generic functions described in more detail in the following section. In contrast to message-passing languages like Ross, which discriminate methods on only a single argument, generic functions allow methods to discriminate on multiple arguments. In addition, we have incorporated into PSE routines for sampling from normal, Poisson, and exponential probability distributions to facilitate nondeterministic stochastic processing not available in Ross.

PSE's process facilities are modeled after those found in Simscript and Simula. Once a process is scheduled, control is turned over to PSE for activating the process. In many cases, processes utilize resources; and, if a required resource is not available, indefinite delays can occur. When the resource is relinquished by another process, it is then assigned to the scheduled process and activation begins. Below, we discuss the simulation capabilities of PSE in more detail and present some explanatory examples.

EVENT-BASED SIMULATION

PSE's event-based simulation facilities include a global clock and built-in functions for scheduling and processing events. The clock object maintains information about events that are scheduled for the future such as the objects referenced by the event and the time at which the event is to occur. The clock advances to the time at which the next scheduled event is to occur. The scheduler then executes the event. The simulation continues executing until all scheduled events are processed. An example of an event defined as a PSE method is the following:

```
;;;
;;; add an auto to a carwash's input queue.
;;;
(defmethod add-to-queue ((carwash resource) (object auto))
    .
    .
  < other functions associated with adding an auto to a carwash queue>
    .
    . )
```

The method add-to-queue can be scheduled as an event by the PSE do-at function:

```
;;;
;;; schedules the method "add-to-queue" to occur every 10 time
;;; units
(defun run-carwash (carwash list-of-vehicles)
    .
    .
  (setq wash-time (current-time))
  (dolist (object list-of-vehicles)
      (do-at carwash wash-time '(add-to-queue ,carwash ,object))
      (setq wash-time (+ wash-time 10)))
    .
    . )
```

The function do-at will add the method add-to-queue to the list of scheduled events. Because the first add-to-queue event is scheduled for the current time, the scheduler will process the event before the clock advances. Another similar PSE function for scheduling events is do-after. The function do-after has the same format as do-at; however, the time parameter indicates a time in the future relative to the current time.

CLOS generic functions give additional modeling power to PSE's simulation facilities not found in message-based simulation languages like Ross. For example, in Ross there can be only one method for add-to-queue defined on a resource object. In the example below, we show how PSE (and CLOS) supports additional methods for add-to-queue that discriminate on the second parameter object. This version of the method is invoked for add-to-queue events where the second argument, object, is an instance of type truck.

```
;;;
;;; add a truck to a carwash's input queue.
;;;
(defmethod add-to-queue ((carwash resource) (object truck))
    .
    .
  < other functions associated with adding a truck to a carwash queue>
    .
    . )
```

PROCESS-BASED SIMULATION

The operational differences between processes and events stem from the definition of a process as an activity that occurs over a duration of time rather than an event that is instantaneous. Processes, like events, are defined as methods and activated as function calls. However, most processes include a *resource* argument. Resources are declared as a subclass of the built-in class resource and therefore inherit methods defining their behavior within process calls. When a PSE process is activated, the system determines if a required resource is free. If an instance of the necessary resource is available, it is automatically assigned to the active process. Control of the resource belongs to the process until it terminates. Scheduling of processes and allocation and deallocation of resources is controlled exclusively by PSE and is transparent to the

user and programmer. In Simscript and Simula, resources must be requested and relinquished by the programmer within the process definition code.

Another feature of PSE processes is the assignment and management of process priorities. Priorities are useful when modeling a scenario with processes of differing precedence. For instance, in a job shop simulation critical time-dependent tasks should be serviced immediately when they are scheduled. However, lower priority "busy work" tasks can be performed at any time or interrupted if higher priority tasks are waiting. Suppose an active process is utilizing a resource, and, subsequently, a higher priority process, requesting the same resource, is scheduled. PSE will suspend the lower priority process, execute the higher priority process, and then resume the suspended process. All process suspension and resumption is managed internally by the PSE system. A user need only specify priorities as an optional argument when defining processes. Simscript and ModSim also support process priorities but require that the simulation application code compare priorities of processes and explicitly suspend processes when necessary [Herri90]. Simula has no built-in capabilities for prioritizing processes.

SINGLE RESOURCE QUEUE VS. MULTIPLE RESOURCE QUEUES

Two variations of process-based simulation are available in PSE: single queue and multiple queue. Single queue processes utilize a single queue for each class of resource that has been declared. Invoking a process that requires a resource instance results in scheduling the resource request on a queue associated with the class of the resource. When a resource instance of the class becomes available, the system will activate the scheduled process. When the resource is relinquished, PSE will select the queued process with the highest priority to execute next.

For resource classes with multiple queues, a request by a process is queued directly on an instance of the resource class. The system determines which resource instance on which to queue the process request by first looking for a free resource and, if none exist, scheduling the process for the resource instance with the shortest queue. The differences between the implementation code and simulation results for single queue and multiple queue simulations are illustrated below in a simple bank teller simulation.

```
;;;
;;; Code segments for teller simulation comparing single and multiple teller
;;; queues
;;;
;;;
;;; Choose one of the following two resource declarations:
;;;
(defresource teller single () ())
;;;(defresource teller multiple () ())

;;;
;;; Define a customer class
;;;
(defclass customer ()
  ((name :accessor name :initform (gensym))
   (service-time :accessor service-time)))

;;;
;;; Define a "service" process whereby a customer is serviced by a teller
;;;
(defprocess service 1 :resource (tel teller) ((cu customer))
  (work tel 'service (service-time cu)))

;;;
;;; The top level function which creates tellers and customers, schedules
;;; service processes, and executes the teller simulation
;;;
(defun run-teller ()
  (setq *clock* (make-clock))
  (let ((customers nil))
       (setf (get 'teller 'resources) nil)
       (make-resource 'teller)
       (make-resource 'teller)
       (setq customers (cons (make-instance 'customer :service-time 100)
                             customers))
       (setq customers (cons (make-instance 'customer :service-time 30)
                             customers))
       (setq customers (cons (make-instance 'customer :service-time 30)
                             customers))
       (setq customers (cons (make-instance 'customer :service-time 30)
                             customers))
       (dolist (c customers)
               (process-at 'teller (current-time) '(service ,c)))
  (run *clock*))
```

In the above code, defresource defines a teller resource class. The first argument of defresource declares the resource class name; the second argument indicates whether the resource is a single or multiple queue resource. The remaining arguments for defresource are identical to those for the CLOS defclass function. The function defprocess defines a simulation process. The first argument passed to defprocess is the process name, the second argument of the process definition provides the process priority, and the list following the :resource keyword indicates the required resource. The other parameters of defprocess are the same as the parameters of the CLOS defmethod statement. A call to the function work within the process definition is used for advancing time during a process. In the function run-teller, the code first creates two tellers and four customers with service times of 100, 30, 30, and 30 units respectively. The call to process-at for each customer queues four service processes. In addition to process-at, which schedules processes at an absolute time, the analogous function process-after schedules processes at a time in the future relative to the current time. Finally, run puts the clock into motion.

Figure 1 shows the results of two versions of the teller simu-

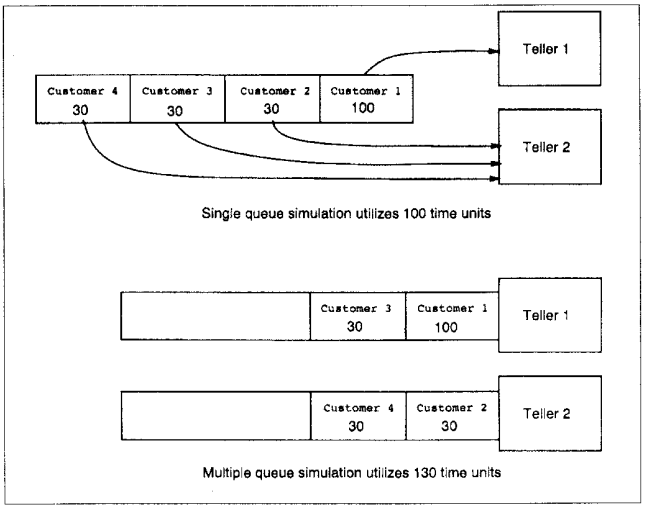

Figure 1. Results of two versions of teller simulation.

lation: one using a single teller queue and the other with multiple teller queues, one per teller. In the single queue version, the customers are placed on a single queue based on their order of arrival. Customers are removed from the queue and assigned to the first available teller. With multiple queues, customers are assigned to the shortest individual teller queue upon arrival. For the given service times, the single queue version will terminate in 100 time units; the multiple queue version requires 130 units to process all customers.

In all our examples so far, processes have required a single instance of a resource class; however, processes can also be defined without the need for resources using the following functions:

```
(process-without-resources-at <time> '(<process-name>
                                       <process-parameters>))
(process-without-resources-after <time> '(<process-name>
                                          <process-parameters>))
```

In such a case, the scheduler will execute the process at the scheduled time. No waiting is necessary because no resources need to be assigned to the process.

MULTIPLE RESOURCE INSTANCES PER PROCESS

Another unique feature of PSE, not available in Simscript or Simula, is the ability to schedule processes requiring multiple instances of a single resource class. For example, in a job shop simulation, a work process may require more than one instance of an identical machine tool or other resource. This feature can be utilized only for single queue resource classes and only for processes without a priority parameter. Each process waiting on a resource queue advances through the queue in the same sequence as it was scheduled. A queued process waits until the required number of resource instances is available before it begins processing. When the resources are free, they are assigned to the waiting process and cannot be used or interrupted by other processes. When the process terminates, all resource instances are relinquished and available for use by other processes. The following PSE functions for dispatching a process with multiple resources correspond to process-at and process-after:

```
(process-mres-at <resource-class> <time>
                                  <number-of-resource-instances>
        '(<process-name> <process-parameters>))
(process-mres-after <resource-class> <time>
                                  <number-of-resource-instances>
        '(<process-name> <process-parameters>))
```

MIXED PROCESSES AND EVENTS

Similar to most other simulation languages, PSE supports the combination of processes and events in a single simulation. An example of mixing processes and events is illustrated in the following code, which is part of a carwash simulation. We have presented a segment of the code representing the beginning of the simulation when the driver of the automobile pays the attendant for the carwash before the car is queued for washing. The activity of paying the attendant could be modeled by a process that represents the exchange of money, transfer of receipt, etc.; however, since none of these individual activities are critical to the simulation, we choose to model carwash payment by use of a single event. As the code describes, the driver first pays the attendant and subsequently a carwash process is scheduled. This example also demonstrates stochastic processing by the use of a normal probability distribution for sequencing autos and for the duration of the carwash process.

```
;;;
;;; Before an auto can get washed, the driver must pay the attendant. This
;;; is the method for the event "pay-attendant".
;;;
(defmethod pay-attendant ((dr driver) (au auto))
  (setf (attendant-paid au) (current-time))
  ;;; After attendant is paid, the car is scheduled for washing
  (process-after 'vacuumer
          (normal *attendant-delay-mean* *attendant-delay-sd*)
          '(vacuum ,au)))


;;;
;;; The top level function which initiates the carwash simulation. The
;;; parameter autoinstances is a list of autos to be dispatched for washing.
;;;
(defun run-carwash (autoinstances)
  (let ((start 0))
    (dolist (auto autoinstances)
        ;;; schedules the "pay-attendant" event
        (do-at (driver auto) start '(pay-attendant ,(driver auto) ,auto))
            ;;; payment of attendant for each auto is time sequenced
            (setq start (+ start (normal *start-mean* *start-sd*))))
    (run *clock*)))
```

RECORDING SIMULATION EVENTS AND PROCESSES IN PSE

Collecting and analyzing the results of simulation trials is a critical component of a simulation lifecycle. Most simulation languages have statistics-gathering routines that can be included in the simulation application code during implementation. PSE has adopted a different approach by transparently maintaining a database of simulation activities. Every simulation activity, including event dispatching, process activation, process suspension, and resource utilization, is recorded in PSE's activity database. With such a complete audit trail of the simulation's activity, a postsimulation trace can be produced in many different formats. Below we illustrate two different formats that can be modified by users to accommodate their own analysis requirements. The first trace is a time-based account of the single queue teller simulation presented in the section on single vs. multiple resource queues. Note, however, that this trace is not generated during simulation processing; rather, the required data is recorded during the simulation and the trace is recreated by retrieving data from PSE's activity database.

```
Time: 0.0
  process service g392 is scheduled with args (#<customer 42346236>)
  process service g392 is started on #<teller 42325446> with args
(#<customer 42346236>)
  process service g393 is scheduled with args (#<customer 42345606>)
  process service g393 is started on #<teller 42322436> with args
(#<customer 42345606>)
  process service g394 is scheduled with args (#<customer 42345156>)
  process service g395 is scheduled with args (#<customer 42347291>)

Time: 30.0
  process service g393 is terminated on #<teller 42322436> with args
(#<customer 42345606>)
  process service g394 is started on #<teller 42322436> with args
(#<customer 42345156>)

Time: 60.0
  process service g394 is terminated on #<teller 42322436> with args
(#<customer 42345156>)
  process service g395 is started on #<teller 42322436> with args
(#<customer 42347291>)

Time: 90.0
  process service g395 is terminated on #<teller 42322436> with args
(#<customer 42347291>)

Time: 100.0
  process service g392 is terminated on #<teller 42325446> with args
(#<customer 42346236>)
```

An alternate trace format, presented below, is organized by process identifier and process status. For each process that is generated, a set of associated data is recorded. This format provides a different organization of the same data presented above:

```
pid = g392
pname = service
scheduled-time = 0.0
start-time = 0.0
resources = #<teller 42325446>
end-time = 100.0
suspended = nil
work-time = (100)
arguments = (#<customer 42346236>)

pid = g393
pname = service
scheduled-time = 0.0
start-time = 0.0
resources = #<teller 42322436>
end-time = 30.0
suspended = nil
work-time = (30)
arguments = (#<customer 42345606>)

pid = g394
pname = service
scheduled-time = 0.0
start-time = 30.0
resources = #<teller 42322436>
end-time = 60.0
suspended = nil
work-time = (30)
arguments = (#<customer 42345156>)

pid = g395
pname = service
scheduled-time = 0.0
start-time = 60.0
resources = #<teller 42322436>
end-time = 90.0
suspended = nil
work-time = (30)
arguments = (#<customer 42347291>)
```

PERSISTENCE IN PSE

Persistent object systems support four major functions: sharing, maintaining, inspecting, and reusing objects. *Sharing* allows the concurrent use of persistent objects by more than one application program similar to a database management system that supports access by multiple programs. Object *maintenance* (insertion, deletion, and updating of simulation objects) can be performed during simulation processing or through maintenance routines applied directly to objects in the persistent object repository external to any simulation program. Objects modified during simulation processing will be transparently updated in the persistent repository so that consistency is maintained between virtual objects in the simulation and secondary storage persistent objects. Likewise, objects can be retrieved and *inspected* during simulation processing and at any time before or after the simulation. Finally, with a persistent object repository simulation ob-

jects can be *reused* without recreating and initializing objects for each simulation trial. For simulations with thousands of objects, reusability contributes significantly to performance improvement. PSE supports three of the four functions described above; sharing of persistent objects has not been addressed because it involves issues of transaction management and is not one of our primary goals. Nevertheless, other persistent object languages are pursuing this topic and their results will contribute to the success of persistent object systems.

PSE ARCHITECTURE

An object that is declared to be a persistent object is retained in secondary storage after program execution terminates. In PSE, once a class has been declared to be persistent those persistent objects are referenced identically to nonpersistent simulation objects. Furthermore, fetching and instantiating a persistent object from secondary storage is performed transparently by the underlying PSE kernel. We based the kernel implementation of PSE on Rowe's shared object hierarchy (SOH) methodology [Rowe86, Rowe88].

PSE is composed of the following components pictured in Figure 2: *persistent object files*, *object space*, and an *object directory*. The object files store an ASCII representation of the objects in secondary storage. Object space denotes the area in main memory where the virtual memory object structures reside and the object directory contains one handle per object, which maps an object identifier into the object handle. The object handle contains metainformation about the object and always remains in main memory. A handle includes information such as a pointer to the object's memory location (which is "nil" if the object is not in the object space), the object's location in the object file, whether or not the object has been modified, and the object's update mode. The update mode indicates how the object will be modified on disk. If the mode is "direct-update" the object will be updated immediately upon modification. If it is "deferred-update," the persistent object will be updated when the number of objects in the object space reaches capacity thereby triggering garbage collection of the object directory and updating of necessary objects. "Local-copy" objects only exist in main memory and therefore are not updated on disk.

During program execution, object handles are used as parameters to represent simulation objects. When a slot in an object is referenced, one of two actions is taken. If it is determined that the object is not in main memory, then it is fetched and instantiated before the slot value is returned. Alternatively, if the object is already in main memory the value of the slot is simply returned. As discussed earlier, the determination of the object's location, fetching, and instantiation are handled by the persistent object system and are transparent to the programmer. For more detailed discussion concerning the architecture of PSE, see [Burdo90].

PSE SYSTEM PARAMETERS

PSE's persistent object system includes a set of parameters that can be modified by the user to tune performance and to measure the system's behavior. The parameter *memory-full* is a global variable that indicates the size of the object space, i.e., the maximum number of objects that the system will allow in memory (or object space) before garbage collecting the object directory. Another useful parameter, *instance-count*, indicates how many persistent objects are currently in the system. The variable *object-faults* records the number of times any object was requested by an application but was not in primary memory and, therefore, needed to be read and instantiated by the system. Finally, *directory-size* is the size of the object directory. If a larger directory structure is needed due to the creation of persistent objects, the system will dynamically allocate more space for the object directory. The combination of these system parameters, with the three choices of update modes, provides users with facilities for comparing performance under different PSE system constraints.

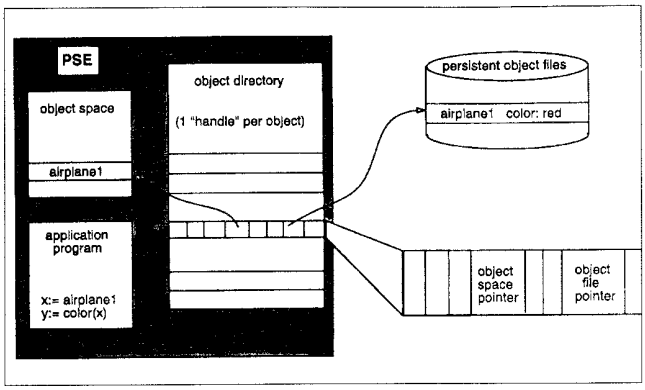

Figure 2. Components of PSE.

PREFETCHING IN PSE

One goal of PSE is to streamline the access of secondary storage objects by "object prefetching." During the execution of a typical POS, objects are retrieved from secondary storage when required by the application program. Object replacement algorithms similar to those used for virtual memory, such as "least recently used," are generally employed for swapping objects in and out of memory. Our methodology promotes a "supply-driven" model of object swapping rather than traditional "demand-driven" algorithms. A supply-driven methodology predicts in advance which objects the simulation will need and loads them into primary memory before the simulation requests them. However, to make predictions about the simulation's future data requirements, knowledge of the application and simulation scenario is needed. Therefore, we categorize our work as "semantic-based object prefetching." Our techniques are based on the identification of a "working set" of objects for any active object being processed by the simulation. The working set consists of objects that have the potential to be subsequently requested. A working set can be defined by geographic locale, temporal locale, or semantic similarity with respect to the active object. One of the two testbed applications, described in the following section, utilizes a working set based on geographic locality; the other application incorporates temporal locale.

The rules for semantic-based prefetching differ depending on

the application. Therefore, PSE does not incorporate any specific prefetching algorithms but instead provides entry points and a set of prefetching methods that application developers can use to enable prefetching. A programmer needs to identify an "active" object and have associated rules for identifying the working set corresponding to a particular object. Many persistent object systems support a concept called "clustering" that tries to attain results similar to those obtainable with prefetching. Clustering, performed by the programmer, is a process by which objects that are frequently referenced together are stored on the same disk page. When one object on the page is retrieved during object fetching, the entire page is loaded into main memory. Clustering is a predominantly static-based organization of objects for improving object fetching. Prefetching in PSE, on the other hand, takes a more active approach to supplying application programs with the objects they may need in the future.

APPLICATIONS

We developed two applications for testing PSE that cover a range of application characteristics. A *route planning* application required a large number of objects and the processing time was consumed predominantly with object maintenance tasks, like instantiating objects from secondary storage and storing objects back into the object files. A second application, *activity networks,* required more compute-bound processing and fewer resources for object maintenance. Below, we describe the applications in more detail and contrast the differences in prefetching performance between the two simulations. The results of our simulation experiments and detailed analysis of performance data for both applications will be presented in a forthcoming paper.

ROUTE PLANNING

A common operation in terrain-based simulations is the generation of shortest path routes. For this reason, we chose the computation of Dijkstra's shortest path algorithm as one testbed application for PSE. The goal of this simulation was to determine the shortest path through a map network of roads where intersections correspond to graph vertices and roads are represented as edges. Dijkstra's algorithm, executing in a traditional persistent object system, results in excessive disk-to-memory thrashing when applied to a large road network (e.g., 10,000 or more objects) because the number of referenced objects quickly exceeds the maximum number allowed in main memory.

In this application, PSE prefetching predicts the future use of objects based on the geographic locale of objects. Geographic locale relates objects that geographically reside "near" each other. Our underlying premise is that as a vehicle traverses the terrain it is more likely to interact with objects in nearby geographic locations. The algorithm based on this premise prefetches any object (edge or vertex) that is directly connected (in the graph) to the object currently being processed by PSE. Our initial results indicate that object maintenance in the PSE implementation of Dijkstra's algorithm accounts for 97% of the total execution time. By using geographic-based prefetching, PSE, on average, can predict the need for 20% to 25% of the objects which previously resulted in object faults. Although this percentage is relatively low, two additional factors must be considered. First, for an application that is so heavily "object-bound" predicting even 20% of the object faults can result in a significant improvement. Finally, in subsequent analysis we have recognized that "smart" prefetching requires more than simply accessing an object before it is needed; the prefetching algorithm should be synchronized so that (1) the object is fetched before it is accessed and (2) the object is not swapped out of memory between the time when it has been fetched and the time when it will be referenced. In future versions, we will be refining our heuristics to incorporate these factors.

ACTIVITY NETWORKS

Activity networks serve as an abstract model of the operation of a logistics maintenance task. Simulating the traversal of tokens in an activity network, therefore, corresponds to throughput in a logistics task. By developing activity networks as an abstract model, the simulation user can parameterize an activity network corresponding to a particular logistics task or set of tasks. Simulation proceeds by the nondeterministic traversal of a given activity network by "tokens." As a token traverses an activity network, it decides along the way (1) what activity it should pursue, (2) what and how many resources to consume, (3) how much time to utilize within a given activity, and (4) what subsequent transition to select (i.e., what activity to traverse next).

Although we are using a network-based simulation model, the edges in activity networks represent temporal sequencing and synchronization of processes rather than spatial distances. Therefore, in contrast to the geographically-based network prefetching, this application requires temporal-based prefetching rules. The rules we have included in our activity network model are based on (1) the resources that are utilized by a process node and (2) the probability of transition between nodes. PSE prefetches all resources associated with an activity node currently being processed. In addition, the process that has the highest probability of subsequently being traversed to is also prefetched. Although activity networks are unlimited in their size, those that we have experimented with contain fewer objects than the map networks used by the shortest path traversal; nevertheless, more computation occurs at each node. The results show a much lower percentage of object maintenance time (20%) compared to map traversal (97%). However, we found that prefetching performance was substantially higher for activity networks. PSE predicted approximately 60% of object faults. Although object prediction is better, prefetching only improves the performance of object maintenance time, which in this application is a smaller percentage of total execution time. By contrasting these two applications, we have determined the wide range of factors that contribute to the overall effectiveness of prefetching.

LIMITATIONS AND FUTURE WORK

PSE is a proof-of-concept prototype. During its design, we focused

on our original goals and, therefore, sidestepped some of the critical issues facing persistent object systems. Future work toward improving the robustness, flexibility, and generality of PSE will address the limitations described in this section.

PSE does not incorporate or interface with a true object management system or object-oriented database management system. It currently interfaces with a system of flat files shown in Figure 2 as "persistent object files." Thus, it is difficult to examine objects in the "database" and PSE has no facilities for modifying the file-based objects. All object editing must be performed through PSE application programs. Furthermore, PSE does not support the modification of class objects once they are declared persistent. Routines for propagating the modifications to all subclasses and instances is necessary to support class modification. Finally, because PSE has no facility for insuring the integrity of competing transactions PSE objects cannot be shared between simulation programs simultaneously. Consistency maintenance of this type, across applications, may also be provided by future object management systems.

The second major shortcoming that affects potential performance improvements afforded by PSE's object prefetching is its uniprocessor architecture. When executing PSE on a single processor, prefetching has no positive effect on performance. However, since the costs of accessing and instantiating an object from secondary storage are high and can have a major impact on performance it would be advantageous to interface object prefetching with a parallel or multiprocessor system. A multiprocessor PSE architecture with prefetching will provide a separate processor to handle the input and instantiation of objects before they are requested by the simulation.

The PSE extensions and refinements discussed above suggest additional directions and goals toward providing simulation developers with even more power and flexibility.

REFERENCES

[**Atkin87**] Atkinson, M.P. and O.P. Buneman. Types and persistence in database programming languages, *ACM Computing Surveys,* **19**(2), 105-190, 1987.

[**Burdo90**] Burdorf, C. and S. Cammarata. Prefetching simulation objects in a persistent simulation environment, *Proceedings of the Society of Computer Simulation Multiconference on Object-Oriented Systems,* San Diego, 1990, pp. 68–74.

[**Dahl67**] Dahl, E. and K. Nygaard. *Simula: A Language for Programming and Description of Discrete Event Systems,* Norwegian Computing Center, Oslo, Norway, 1967.

[**Ford88**] Ford, S. J. J., S.E. Langworthy, D.F. Lively, G. Pathak, E. R. Perez, R. W. Peterson, D.M. Sparacin, S.M. Thatte, D.L. Wells, and S. Agarwala. Zeitgeist: database support for object-oriented programming, *Proceedings o the 2nd International Workshop on Object-Oriented Database Systems,* Bad Munster am Stein-Edernburg, FRG, September 1988, pp. 22-42.

[**Herri90**] Herring, C. ModSim: a new object-oriented simulation language, *Proceedings of the Society of Computer Simulation Multiconference on Object-Oriented Systems,* San Diego, 1990.

[**Khosh88**] Khoshafian, S. A persistent complex object database language, *Data Knowledge Engineering,* 3, 225-243, 1989.

[**McArt84**] McArthur, D., P. Klahr, and S. Narain. *Ross: An Object-Oriented Language for Constructing Simulations,* R-3160-AF, The RAND Corporation, Santa Monica, CA, 1984.

[**Rowe86**] Rowe, L.A. A shared object hierarchy, *Proceedings of the IEEE International Workshop on Object-Oriented Database Systems,* Pacific Grove, CA, September 1986.

[**Rowe88**] Rowe, L.A. Picasso shared object hierarchy, *Proceedings of the First CLOS Users and Implementors Workshop,* Palo Alto, CA, 1988.

[**Russe79**] Russel, E. *Simulating with Processes and Resources in Simscript II.5,* CACI, San Diego, CA, 1979.