Proceedings of the 2003 Winter Simulation Conference
S. Chick, P. J. Sánchez, D. Ferrin, and D. J. Morrice, eds.

SPADES — A DISTRIBUTED AGENT SIMULATION ENVIRONMENT WITH SOFTWARE-IN-THE-LOOP EXECUTION

Patrick F. Riley

Computer Science Dept.
Carnegie Mellon University
Pittsburgh, PA 15213-3891, U.S.A.

George F. Riley

College of Engineering
School of Electrical and Computer Engineering
Georgia Institute of Technology
Atlanta, GA 30332-0250, U.S.A.

ABSTRACT

Simulations are used extensively for studying artificial intelligence. However, the simulation technology in use by and designed for the artificial intelligence community often fails to take advantage of much of the work by the larger simulation community to produce distributed, repeatable, and efficient simulations. We present the new system known as *System for Parallel Agent Discrete Event Simulator*, (*SPADES*), which is a simulation environment for the artificial intelligence community. *SPADES* focuses on the *agent* as a fundamental simulation component. The *thinking time* of an agent is tracked and reflected in the results of the agents' actions by using a *Software–in–the–Loop* mechanism. *SPADES* supports distributed execution of the agents across multiple systems, while at the same time producing repeatable results regardless of network or system load. We discuss the design of *SPADES* in detail and give experimental results. *SPADES* is flexible enough for a variety of application domains in the artificial intelligence research community.

1 INTRODUCTION

Simulations are an accepted and widely used method for studying artificial intelligence techniques for multi–agent interaction. By simulating the environment and agent actions, a researcher can systematically tune the parameters of the environment and execute the large number of trials often required for machine learning. However, commonly used simulation techniques often do not address the special concerns of the artificial intelligence community. In particular, previously used simulation environments do not track and model the computation time of an agent in response to sensed environmental events. Existing simulation methods used in artificial intelligence research are often non–repeatable, being sensitive to network and system loads at the time of the simulation execution. Finally, many simulators created in the artificial intelligence community fail to take advantage of existing work in the parallel and distributed simulation community for designing distributed, efficient, and repeatable simulations.

This paper demonstrates the application of well–known parallel and distributed simulation methods for time management to agent–based distributed simulation for artificial intelligence research. In addition, we introduce the concept of *Software–in–the–Loop* simulation, which we have found to be particularly useful for multi–agent artificial intelligence research. Our *Software–in–the–Loop* technique provides for the tracking of the computation time used by those agents, and including that so–called *think time* in the simulation. By taking advantage of prior work in parallel discrete event simulation, the *SPADES* system eases the design of a simulation by hiding many of the system details required to handle distributed simulation in an efficient and reproducible way.

2 RELATED WORK

The problem of creating efficient simulations has attracted substantial attention for decades from a wide range of sources, including the AI community, scientific computing, computer networking, industry, and government. While the notion of software *agents* has been known for some time, the agent-based or agent-oriented simulation methods are relatively new in the simulation community. Much of the groundwork for agent–based simulations is by Uhrmacher (Uhrmacher 1996, Uhrmacher 1997, Uhrmacher and Schattenberg 1998, Uhrmacher and Gugler 2000, Uhrmacher, Tyschler, and Tyschler 1997, Uhrmacher and Krahmer 2001). For example, the *James* system (Uhrmacher and Gugler 2000) is a Java–based simulation environment for agent modeling, similar in concept to our *SPADES* simulator.

Agent–based simulation methods have existed for much longer in the artificial intelligence community. Many AI simulation environments are quite specific to the domain for which they were created. The *GENSIM* system (Anderson and Evans 1995) is one exception. It attempts to provide

support for general agent based simulation, including a vision like model of sensation and computation time tracking of the agents. The agents are given sensations at fixed intervals and have a fixed amount of computation to respond to each sensation. The simulation is written in LISP and requires all agents to be also. A distributed version called *DGENSIM* (Anderson 2000) was created which has an architecture much like *SPADES*. However, *DGENSIM* has no methods to handle network and machine delays and requires all agents advance in time synchronously.

The MESS system by Anderson (1995, 1997) is similar is spirit to *GENSIM*. It also requires all agents to be written in LISP, but provides much more flexible tracking of agent computation.

Some work in the AI community has been done on distributing agent based simulation but typically leaves many of the issues in distribution management to the world designer. For example, Lees, Logan, and Theodoropoulos (2002) provide an HLA based distributed simulation, but fail to provide the simulation creator with a world view that is unaffected by how objects are distributed or any support for handling synchronous, conflicting actions of the agents.

On the larger scale of agent simulation, the *MACE3J* system (Gasser and Kakugawa 2002) is a highly flexible, java-based agent simulation system. Scaling up is a main design criteria of the system; it has been run with up to 50 processors and 5000 agents. It relies on a Shared System Image system to provide distributed machines with a consistent image of the model.

Much of the work in creating efficient distributed simulations deals with how to break down a simulation into components such that the communication requirements between the components is low. For example, in agent based simulation, Logan and Theodoropolous (2001) discuss how "spheres of influence" can be used to adaptively and flexible organize simulation objects and agents for efficient distributed simulation. *SPADES* takes a different approach. The breakdown of components is fixed (agents and a world model). What *SPADES* reasons about is how to allow as many agents as possible to compute without violating causality. Notice that *SPADES* does allow executions out of time order as long as they do not violate causality.

3 THE *SPADES* SIMULATION ENVIRONMENT

This section discusses the major features of the *SPADES* simulation system. When discussing these features, we will use the terms *simulation time* or just *time* to refer to the simulation time within the simulation environment. The term *wall clock time* will be used to refer to the real–time as measured by a watch outside the simulated environment. *CPU Time* will refer to the amount of time a process has used in the central processing unit on the computer system performing the simulation.

SPADES supports agent-based *execution*, as opposed to agent-based modeling or implementation (Uhrmacher 1997). In this context, agent-based execution means that the system explicitly models the sensing, thinking, and acting components (and their latencies) which are the core of any agent. Figure 1 represents a typical timeline for executions within a cycle. Time point A represents the point at which a sensation occurs in the environment. Time period AB represents the elapsed time for an agent to identify and classify the event (such as the video frame capture time). Period BC is the CPU time required for the agent to decide what to do in response to the event, and CD is the time it takes before the action begins to have an effect on the world. *SPADES* allows arbitrary latencies for each of the above time periods, and allows overlapped actions as shown. However, two *think* cycles are not allowed to overlap, since a typical deployed agent only has a single CPU to use for the thinking step. We model this *thinking* action by our *Software–in–the–Loop* methodology described next.

A basic premise used by *SPADES* is that the amount of time an agent takes to *think* is non-negligible, and must be included in the simulation model. Further, the thinking time for actions is not constant, varying based on the type of sensed event, current world state, and other variables. Finally, we assume that the actual software used in the deployed agents to think about sensation events is included as part of the *SPADES* simulation. Given these assumptions, we developed our novel *Software–in–the–Loop* methodology which allows accurate modeling of the thinking time. Since the deployed software is included in the simulation, the amount of CPU time used by the simulated thinking process is identical to that used by the deployed agent in the same environment (subject to a linear scale factor to account for differing CPU speeds). We simply measure the CPU time used by the thinking process in the simulation by using the Linux *perfctr* (`<http://sourceforge.net/projects/perfctr/>`) feature. This feature includes a patch for the standard Linux kernel which provides per-process counts of CPU cycles and instructions executed by the process. After measuring the CPU time used by the simulated *think* process and applying a linear scale factor, *SPADES* schedules the *act* event at the appropriate delayed simulation time. We point out that the term *Software–in–the–Loop* has been used previously in the simulation literature (Choi and Kwon 1999), referring to

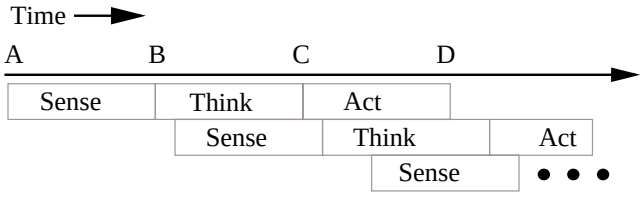

Figure 1: Example Timeline for the Sense-Think-Act Loop of an Agent

a method whereby some hardware portions of *hardware–in–the–loop* simulations are replaced by software–based simulations. While this is similar in spirit to our approach, it is substantially different.

In order to provide maximum inter-operability, *SPADES* makes no requirements on the agent architecture (except that it supports the sense-think-act cycle) or the language in which agents are written (except that they can write to and from Unix pipes). In the same spirit as the SoccerServer (Noda, Matsubara, Hiraki, and Frank 1998), *SPADES* provides an environment where agents built with different architectures or languages can inter-operate and interact in the simulated world.

SPADES is a conservative parallel discrete event simulator as described in Misra (1986). In conservative simulations, events are not processed until it can be guaranteed that casual event ordering will not be violated. In contrast, optimistic simulations (Jefferson 1985) process events without regard to causality, but instead support a rollback mechanism that is invoked in case events are found to have been executed out of order. Debates over the merits of conservative and optimistic simulation are common and several surveys discuss the issues (Ferscha and Tripathi 1996, Fujimoto 1990). Our choice of the conservative methodology was simply a practical choice due to ease of implementation. However, our design does allow some degree of out–of–order event execution, if those events are known to be not causally related.

An effect of the discrete event nature of our distributed simulation environment is that agents' interactions are not necessarily synchronized. Any subset of the agents can have actions take effect at a given time step. This is in contrast to many simulations in the AI community, that require that all agents choose an action simultaneously, with the state of the world model updated once based on all these actions. *SPADES*-based simulations do not require the agents' actions to be synchronized in this manner. In particular, smaller time quanta for simulation of the world model do not increase the simulation's network load. In other words, the affect of agents' actions are realized precisely at the correct time in the simulation, as opposed to the artificially imposed time–step actions of other simulators.

Finally, the *SPADES* system provides reproducible simulation results. Given the same set of initial conditions and the same random seeds, *SPADES* will produce identical results for every simulation execution, as demonstrated in section 4.

3.1 System Architecture

Figure 2 gives an overview of the entire *SPADES* system, along with the components users of the system must supply (shaded in the diagram). The dotted lines denote possible machine boundaries. The simulation engine and the

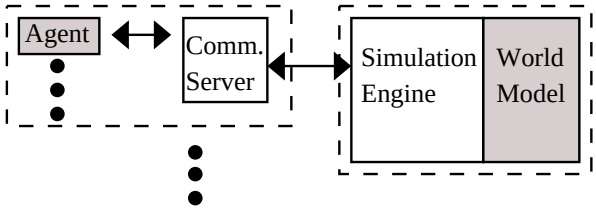

Figure 2: Overview of the Architecture of *SPADES*

communication server are supplied as part of *SPADES*. The world model and the agents are created by a user to simulate a particular environment.

The simulation engine is the heart of the discrete event simulator. All pending events are queued here, and the engine coordinates all network communication. A communication server must be run on each machine on which agents run. The communication server manages all communication with the agents (through a Unix pipe interface) as well as tracking the CPU usage of the agents to calculate the thinking latency. The communication server and simulation engine communicate over a TCP/IP connection.

The world model is created by a user of *SPADES* to create a simulation model of a particular environment. The simulation engine is a library to which the world model must link, so the simulation engine and world model exist in the same process. The world model must provide such functionality as advancing the state of the world to a particular time and realizing an event (changing the state of the world in response to an event occurring). *SPADES* provides a collection of C++ classes from which objects in the world model can inherit in order to interact with the simulation engine.

The agents communicate with the communication server via pipes, so the agents are free to use any programming language and any architecture as long as they can read and write to pipes. From the agent's perspective, the interaction with the simulation is fairly simple:

1. Wait for a sensation to be received
2. Decide on a set of actions and send them to the communication server
3. Send a *done thinking* message to indicate that all actions were sent.

One of the communication server's primary jobs is to track the thinking time of the agent to support the *Software–in–the–Loop* methodology. When sending a sensation to an agent, the communication server begins tracking the CPU time used by the agent. When the *done thinking* message is received, the communication server calculates the total amount of CPU time used to produce these actions. All actions are given the same time stamp of the end of the think phase.

The agents have one special action which *SPADES* understands: a *request time notify*. The agent's only opportunity to act is upon the receipt of a sensation. Therefore if

an agent wants to send an action at a particular time (such as a stop turning command for a robot), it can request a time notify. On the receipt of the time notify, the agent can returns actions as for any other sensation. In order to give maximum flexibility to the agents, *SPADES* does not enforce a minimum time in the future that time notifies can be requested. However, all actions, whether resulting from a regular sensation or a time notify, are still constrained by the action latency.

3.2 Discrete Event Simulator

This section describes the simulation algorithm used by *SPADES*. This algorithm is a modification of a basic discrete event simulator.

In order to insure that all events will be executed in causal order, the simulation environment must determine whether or not it is possible to receive a future event with a timestamp less than the next pending event. This so–called *time–management* function of parallel simulators is well studied, and there are a number of approaches that can be used (Chandy and Misra 1981, Bryant 1977, Mattern 1993, Chandy and Misra 1979, Chandy and Sherman 1989, Lubachevsky 1989, Steinmann 1991, Nicol 1993, Riley, Fujimoto, and Ammar 2000). Much of the complexity of these approaches is due to the fact that typically a distributed simulation will manage private event lists for each process in the distributed environment. In other words, each process manages its own event list, and schedules events to and from this list independently from other processes (within the constraints imposed by the time management algorithms). For ease of implementation, we chose another well–known approach known as a *centralized event list*. In this approach, a single composite event list is managed by a *master* process, which is responsible for scheduling events and managing the event list for all other processors. Any process that needs to schedule a future event must notify the master process (the manager of the central event list) to get the event scheduled. This master process has complete knowledge at all times of pending events, and can independently determine which pending events can be safely processed. A drawback of the central event list approach is that each process must notify the central scheduler that it has finished processing a prior event and is ready to process more events. The design of the agents using the sense–think–act paradigm mitigates this drawback, since all agents produce an action in response to sensed events, which serves as notification to the scheduler that the processing has completed. An obvious major drawback of this approach is efficiency and scalability, since a single process coordinates activities for all agents. This single coordination point could become a bottleneck and slow down the entire simulation. For our purposes, the total number of agents is reasonably small, and we haven't observed significant overhead in the centralized event list management. The performance graphs given later show clearly the overall execution time is dominated by the agents' CPU requirements for processing sensation events.

It is well understood that any conservative parallel discrete event simulator requires a non–zero *lookahead* property in order to achieve good parallel performance (Ferscha and Tripathi 1996). Simply stated, the *lookahead* value is a lower bound on the simulation time difference between the generation of an event on any processor A and the realization of that event on some other processor B. Larger lookahead values are known to give rise to better parallel performance. We now discuss the the lookahead algorithm of *SPADES*. We will first cover a simple version which covers some of the fundamental ideas and then describe the *SPADES* algorithm in full.

An explanation of the events that occur in the normal think-sense-act cycle of the agents must first be given. The nature of this cycle illustrated in Figure 3. First, an event is put into the queue to create a sensation. Typically, the realization of this event reads the state of the world and converts this to some set of information to be sent to the agent. This set is encapsulated in a sense event and put into the event queue. *SPADES* requires that the time between the create sense event and the sense event is at least some minimum sense latency, which is specified by the world model. When the sense event is realized, this set of information will be sent to the agent to begin the thinking process. Notice that the realization of a sense event does not require the reading of any of the current world state since the set of information is fixed at the time of the realization of the create sense event. Upon the receipt of the sensation, the communication server begins timing the agent's computation. When all of the agent's actions have been received by the communication server, the computation time taken by the agent to produce those actions is converted to simulation time. All the actions and the think latency are sent to the simulation engine (shown as "Act Sent" in Figure 3). Upon receipt, the simulation engine adds the action latency (determined by querying the world model) and puts an act event in the pending events queue. Similar to the minimum sense latency, there is a minimum action latency which *SPADES* requires between the sending time of an action and the act event time. The realization of that act event is what actually causes that agent's actions to affect the world. Note that the "Act Sent" time is circled because unlike the others that represent events in the queue, "Act Sent" is just a message from the communication server to the engine and not an event in the event queue.

Note that a single agent can have multiple sense-think-act cycles in progress at once, as illustrated in Figure 1. For example, once an agent has sent its actions (the "Act Sent" point in Figure 3), it can receive its next sensation even though the time which the actions actually affect the world (the "Act Event" point in Figure 3) has not yet occurred.

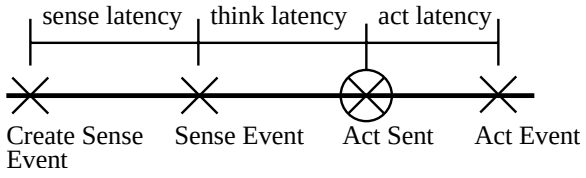

Figure 3: Events in Sense-Think-Act Cycle of an Agent

The only overlap *SPADES* forbids is the overlapping of two think phases.

Note also that all actions have an effect at a discrete time. Therefore there is no explicit support by *SPADES* for supporting the modeling of the interaction of parallel actions. For example, the actions of two simulated robots may be to start driving forward. It is the world model's job to recognize when these actions interact (such as in a collision) and respond appropriately. Similarly, communication among agents is handled as any other action. The world model is responsible for providing whatever restrictions on communication desired.

The sensation and action latencies provide a lookahead value for that agents and allows the agents to think in parallel. When a sense event is realized for agent 1, it cannot cause any event to be enqueued before the current time plus the minimum action latency. Therefore it is safe (at least when only considering agent 1) to realize all events up till that time without violating event ordering.

The quantity we call the "minimum agent time" determines the maximum safe time over all agents. The minimum agent time is the earliest time which an agent can cause an event which affects other agents or the world to be put into the queue. This is similar to the Lower Bound on Timestamp (LBTS) concept used in the simulation literature. The calculation of the minimum agent time is shown in Table 1. The agent status is either "thinking," which means that a sensation has been sent to the agent and a reply has not yet been received, or "waiting," which means that the agent is waiting to hear from the simulation engine. Besides initialization, the agent status will always be thinking or waiting. The current time of an agent is the time of the last communication with the agent (sensation sent or action received). The receipt of a message from a communication server cannot cause the minimum agent time to decrease. However, the realization of an event can cause an increase or a decrease. Therefore, the minimum agent time must

Table 1: Code to Determine the Minimum Time that an Agent Can Affect the Simulation

```
calculateMinAgentTime()
  ∀i ∈ set_of_all_agents
    if (agent_i.status = Waiting) agent_time_i = ∞
    else agent_time_i = agent_i.currenttime
                        + min_action_latency
  return min_i agent_time_i
```

Table 2: Code for Parallel Agent Discrete Event Simulator for Strict Timestamp Order

```
repeat forever
  wait for messages
  next = pending_events.head
  min_agent_time = calculateMinAgentTime()
  while (next.time < min_agent_time)
    advanceWorldTime (next.time)
    pending_events.remove(next)
    realizeEvent (next)
    next = pending_events.head
    min_agent_time = calculateMinAgentTime()
```

be recalculated after each event realization. However, this algorithm could be modified to be incremental so that the entire agent set does not have to be scanned each time.

Based on the calculation of the minimum agent time, we can now describe a simple version of the parallel agent discrete event simulator, which is shown in Table 2. The value `min_agent_time` is used to determine whether any further events can appear before the time of the next event in the queue.

While this algorithm produces correct results (all events are realized in time stamp order) and achieves some parallelism, it does not achieve the maximum amount of possible parallelism. Figure 4 illustrates an example with two agents. When the sense event for agent 1 is realized, the minimum agent time becomes A. This allows the create sense event for agent 2 to be realized and the sense event for agent 2 to be enqueued. However, the sense event for agent 2 will not be realized until the response from agent 1 is received. However, as discussed above, the effect of the realization of a sense event does not depend on the current state of the world. If agent 2 is currently waiting, there is no reason not to realize the sense event and allow both agents to be thinking simultaneously.

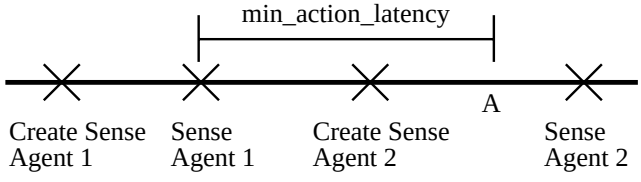

Figure 4: An Example Illustrating Possible Parallelism that the Simple Parallel Agent Algorithm Fails to Exploit

However, this allows the realization of events out of order; agent 1 can send an event which has a time less the time of the sense event for agent 2. Certain kinds of out of order realizations are acceptable (as the example illustrates). In particular, we need to verify that out of order events are not causally related. The key insight is that sensations received by agents are casually independent of sensations

received by other agents. In order to state our correctness guarantees, we will define a new sub-class of events "fixed agent events" which have the following properties:

1. They do not depend on the current state of the world.
2. They affect only a single agent, possibly by sending a message to the agent.
3. Sense events and time notify events are both fixed agent events.
4. Fixed agent events are the only events which can cause the agent to start a thinking cycle, but they do *not* necessarily start a thinking cycle.

The correctness guarantees that *SPADES* provides are:

1. All events which are not fixed agent events are realized in time order.
2. The set of fixed agent events for a particular agent are realized in time order.

In order to achieve this, several new concepts are introduced. The first is the notion of the "minimum sensation time." This is the earliest time that a *new* sensation (i.e. fixed agent event) *other than a time notify* can be generated and enqueued. The current implementation of *SPADES* requires that the world model provide a minimum time between the create sense event and the sense event (see Figure 3), so the minimum sensation time is the current simulation time plus that time.

The time notifies are privileged events. They are handled specially because they affect no agent other than the one requesting the time notification. *SPADES* also allows time notifies to be requested an arbitrarily small time in the future, before even the minimum sensation time. This means that while an agent is thinking, the simulation engine cannot send any more fixed agent events to that agent without possibly causing a violation of correctness condition 2. However, if an agent is waiting (i.e. not thinking), then the first fixed agent event in the pending event queue can be sent as long as its time is before the minimum sensation time.

To insure proper event ordering, one queue of fixed agent events per agent is maintained. All fixed agent events enter this queue before being sent to the agent, and an event is put into the agent's queue only when the event's time is less than the minimum sensation time.

There are several primary functions dealing with the agent queue. First, enqueueAgentEvent puts a fixed agent event into the queue. The doneThinking function is called when an agent finishes its think cycle. Both functions call a third function checkForReadyEvents. Pseudo-code for these functions is shown in Table 3. Note that in checkForReadyEvents, the realization of an event can cause the agent status to change from waiting to thinking.

Using these functions, we describe in Table 4 the main loop that *SPADES* uses. This is a modification of the algorithm given in Table 2. The two key changes are that in the first while loop, fixed agent events are not realized,

Table 3: Code for Maintaining the Per-Agent Fixed Agent Event Queues

```
checkForReadyEvents(a: Agent)
    while (true)
      if (agent_a.status = thinking)
        return
      if (agent_a.pending_agent_events.empty())
        return
      next = agent_a.pending_agent_events.pop()
      realizeEvent(next)

enqueueAgentEvent(e:Event)
    a = e.agent
    agent_a.pending_agent_events.insert(e)
    checkForReadyEvents(a)

doneThinking(a: Agent, t:time)
    agent_a.currenttime = t
    checkForReadyEvents(a)
```

but are put in the agent queue instead. The second loop (the "foreach" loop) scans ahead in the event queue and moves all fixed agent events less that the minimum sensation time into the agent queues. Note that in both cases, moving events to the agent queue can cause the events to be realized (see Table 3).

4 EMPIRICAL VALIDATION

In order to test the efficiency of the simulation and to understand the effects of the various parameters on the performance of the system, we implemented a simple world model and agents and ran a series of experiments. We tracked the wall clock time required to finish a simulation as a measure of the efficiency.

4.1 Sample World and Agents

The simulated world is a two dimensional rectangle where opposite sides are connected (i.e. "wrap-around"). Each agent is a "ball" in this world. Each sensation the agent receives contains the positions of all agents in the simulation, and the only action of each agent is to request a particular velocity vector. The dynamics and movement properties are reasonable if not exactly correct for small omni-directional robots moving on carpet, except that collisions are not modeled. The world model advanced in 1ms increments.

We created two kinds of agents. The "wanderer" moves randomly around the world. The "chaser" chases one of the randomly moving agents by setting its requested velocity directly towards the current observed location of that agent.

Table 4: Code for Efficient Parallel Agent Discrete Event Simulator as Used by *SPADES*

```
repeat forever
  wait for messages
  next = pending_events.head
  min_agent_time = calculateMinAgentTime()
  while (next.time < min_agent_time)
    advanceWorldTime (next.time)
    pending_events.remove(next)
    if (next is a fixed agent event)
      enqueueAgentEvent(next)
    else
      realizeEvent (next)
    next = pending_events.head
    min_agent_time = calculateMinAgentTime()
  min_sense_time = current_time
                   + min_sense_latency
  foreach e (pending_events) /* in time order */
    if (e.time > min_sense_time)
      break
    if (e is a fixed agent event)
      pending_events.remove(e)
      enqueueAgentEvent(e)
```

4.2 Experimental Setup

All experiments were run on the *Ferrari* Linux cluster at Georgia Tech. The cluster consists of sixteen identical Linux boxes, each with 2 Pentium III CPU's running at 850Mhz. The operating system is RedHat Linux version 7.3. Each system has 2GB of main memory, and all systems are connected via a private Gigabit Ethernet network and a Foundry BigIron router.

For these experiments, we varied three parameters of the simulation environment:

- The number of machines, varying from 1 to 13 (hardware problems prevented using all 16 machines).
- The number of agents, varying from 2 to 26.
- Computation requirements of the agents. To simulate agents that do more or less processing, we put in simple delay loops. We used 3 simple conditions of fast, medium, and slow agents. Fast agents simply parse the sensations and compute their new desired velocity with a some simple vector calculations. The medium and slow agents add a simple loop that counts to 500,000 and 5,000,000 respectively. On an 850MHz Pentium III, this translates to approximately 1.0ms and 9.0ms average response time. Only the fast and slow performance graphs are shown, due to space considerations.

Every experimental condition was run five times and the median of those five times is reported. Each simulation was run for 90 seconds of simulation time. In all experiments, the agents received sensations every 95–105 milliseconds (actual value chosen uniformly randomly after each sensation). The sensation latency was chosen uniformly randomly between 30 and 40 milliseconds for each sensation and action.

The processes were distributed to the machines as follows: The same machine always runs the simulation engine and world model process. Then, all machines (including the machine running the simulation engine) run a communication server, with the agents as equally distributed as possible to all communication servers.

4.3 Results

Figure 5 shows speedup compared to running the simulation on a single machine. The performance charts show some interesting of interesting results.

(a) Fast

(b) Slow

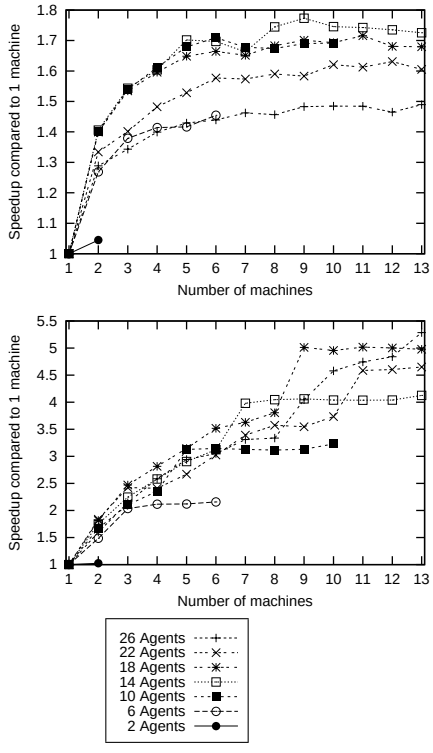

Figure 5: Speedup Results with the Sample World Model and Agents (note that the y-axes have different scales)

Moving from a single processor to two processors never slows down the simulation, and in most cases achieves speedups between 1.3 and 1.75.

As expected, there is significant diminishing return as the number of machines increases, due to the additional inter–processor overhead required as more processors are added to the simulation.

The detrimental affect of the communication overhead is quite pronounced in the *fast* agents case (Figure 5(a)). While we always get non–zero speedup in the 2–processor case (in the range of 1.3 to 1.75), the communication overhead for additional processors begins to dominate the simulation's performance, with little further speedup beyond 5 processors. For larger processing time (Figure 5(b)), the communication overhead becomes less significant, with continued speedup improvement up to 13 processors in some cases. The affect of proper load balancing is obvious. Observe the speedup chart in Figure 5(b). Notice the large jump in speedup in the 14 agents case (marked with the hollow boxes) when the number of processors increases from 6 to 7. With 14 agents distributed on 6 processors, 4 of the processors are assigned 2 agents, and 2 of the processors have 3 agents. Since the overall progression of time in the simulation is bounded by the slowest running processor, the performance is bounded by the processors with 3 agents each. When 7 processors are assigned to this scenario, each processor gets exactly 2 agents and a noticeable speedup jump occurs at this point. The speedup line for the 14 agent case then remains nearly flat up to 13 processors, since one or more processors must have 2 agents up to that point. Similar results can be seen for the 18 agents case. A large speedup jump occurs when increasing the processors from 8 to 9, which is the point where all processors have exactly two agents. Again, the speedup remains reasonably flat beyond 9 processors, for the same reasons.

The affect of proper load balancing is obvious. Observe the speedup chart in Figure 5(b). Notice the large jump in speedup in the 14 agents case (marked with the hollow boxes) when the number of processors increases from 6 to 7. With 14 agents distributed on 6 processors, 4 of the processors are assigned 2 agents, and 2 of the processors have 3 agents. Since the overall progression of time in the simulation is bounded by the slowest running processor, the performance is bounded by the processors with 3 agents each. When 7 processors are assigned to this scenario, each processor gets exactly 2 agents and a noticeable speedup jump occurs at this point. The speedup line for the 14 agent case then remains nearly flat up to 13 processors, since one or more processors must have 2 agents up to that point. Similar results can be seen for the 18 agents case. A large speedup jump occurs when increasing the processors from 8 to 9, which is the point where all processors have exactly two agents. Again, the speedup remains reasonably flat beyond 9 processors, for the same reasons.

4.4 Repeatability

In order to verify the reproducibility of the *SPADES* system, we ran a further set of of experiments. For every combination of 4, 12, and 24 agents running at the fast, medium, and slow speeds (as described above), we repeatedly ran simulations with the same random seeds given to both the world model and the agents. For each combination, we ran trials using from 1 to 8 machines. Two trials were run while no other significant processes were run on the machine, and two were run with no control over extra processes and artificial load added to half of the machines. The artificial load consisted of five processes running in infinite loops.

In all cases, the results of the simulation in terms of the positions, the sensations, and the actions of all the agents are exactly identical. It should be noted that the order of event realization is not identical, as *SPADES* allows certain out of order executions which do not violate causality.

Further note that perfect reproducibility can also be achieved without the *perfctr* based timer. *SPADES* also supports the the recording of thinking times from one run to be replayed during a subsequent run.

ACKNOWLEDGMENTS

This research was sponsored in part by the United States Air Force under Grants Nos. F30602-00-2-0549 and F30602-98-2-0135, by an NSF Fellowship, by NSF Grants Nos. ANI-9977544, ANI-0225417, ECS-0225471, and DARPA Contract number N66002-00-1-8934. The content of this publication does not necessarily reflect the position or the policy of the sponsors and no official endorsement should be inferred.

REFERENCES

Anderson, J. 2000. A generic distributed simulation system for intelligen agent design and evaluation. In *Proc. of the 10th Int. Conference on AI, Simulation, and Planning in High Autonomy Systems*, 36–44.

Anderson, J., and M. Evans. 1995, October. A generic simulation system for intelligent agent designs. *Applied Artificial Intelligence* 9 (5): 527–562.

Anderson, S. D. 1995. A simulation substrate for real-time planning. Technical Report 95-80, University of Massachusetts at Amherst Computer Science Department. (Ph.D. thesis).

Anderson, S. D. 1997. Simulation of multiple time-pressured agents. In *Proc. of the 1997 Winter Simulation Conference*, ed. S. Andradóttir, K. J. Healy, D. H. Withers, and B. L. Nelson, 397–404.

Bryant, R. E. 1977. Simulation of packet communications architecture computer systems. In *MIT-LCS-TR-188*.

Chandy, K., and J. Misra. 1979. Distributed simulation: A case study in design and verification of distributed programs. In *IEEE Transactions on Software Engineering*.

Chandy, K., and J. Misra. 1981. Asynchronous distributed simulation via a sequence of parallel computations. In *Communications of the ACM*, 24.

Chandy, K. M., and R. Sherman. 1989. The conditional event approach to distributed simulation. In *Proc. of the SCS Multiconference on Distributed Simulation*.

Choi, S.-G., and W. H. Kwon. 1999. Real-time distributed software-in-the-loop simulation for distributed control systems. In *Proc. of the 1999 IEEE International Symposium on Computer Aided Control System Design*, 115–119.

Ferscha, A., and S. Tripathi. 1996. Parallel and distributed simulation of discrete event systems. In *Parallel and Distributed Computing Handbook*, ed. A. Y. Zomaya, 1003–1041. McGraw-Hill.

Fujimoto, R. M. 1990. Parallel discrete event simulation. *Communications of the ACM* 33 (10): 30–53.

Gasser, L., and K. Kakugawa. 2002. MACE3J: Fast flexible distributed simulation of large, large-grain multi-agent systems. In *AAMAS-02*, 745–752.

Jefferson, D. R. 1985. Virtual time. *ACM Transactions on Programming Languages and Systems*, 7: 404–425.

Lees, M., B. Logan, and G. Theodoropoulos. 2002. Simulating agent-based systems with HLA: The case of SIM_AGENT. In *Proc. of the 2002 European Simulation Interoperability Workshop (ESIW'02)*, 285–293. Simulation Interoperability Standards Organisation and Society for Computer Simulation.

Logan, B., and G. Theodoropolous. 2001. The distributed simulation of multi-agent systems. *Proc. of the IEEE* 89 (2): 174–185.

Lubachevsky, B. D. 1989. Efficient distributed event-driven simulations of multiple-loop networks. *Communications of the Association for Computing Machinery* 32 (1): 111–123.

Mattern, F. 1993. Efficient algorithms for distributed snapshots and global virtual time approximation. In *Journal of Parallel and Distributed Computing*.

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

Nicol, D. M. 1993. The cost of conservative synchronization in parallel discrete event simulations. *Journal of the Association for Computing Machinery* 40 (2): 304–333.

Noda, I., H. Matsubara, K. Hiraki, and I. Frank. 1998. Soccer server: A tool for research on multiagent systems. *Applied Artificial Intelligence* 12: 233–250.

Riley, G. F., R. M. Fujimoto, and M. H. Ammar. 2000. Network aware time management and event distribution. In *Proc. of the 14th Workshop on Parallel and Distributed Simulation*.

Steinmann, J. 1991. Speedes: Synchronous parallel enviornment for emulation and discrete event simulation. *Advances in Parallel and Distributed Simulation, SCS Simulation Series* 23:95–103.

Uhrmacher, A. M. 1996. Concepts of object and agent–oriented simulation. In *Proc. of the Workshop on Multiagent Systems and Simulation.*

Uhrmacher, A. M. 1997. Concepts of object and agent–oriented simulation. *Tran. of the Society for Computer Simulation* 14 (2): 59–67.

Uhrmacher, A. M., and K. Gugler. 2000. Distributed parallel simulation of multiple deliberative agents. In *Proc. of the 14th Workshop on Parallel and Distributed Simulation*.

Uhrmacher, A. M., and M. Krahmer. 2001. A conservative approach to the distributed simulation of multi–agent systems. In *Proc. of the European Multi–Conference on Simulation*.

Uhrmacher, A. M., and B. Schattenberg. 1998. Agents in discrete event simulation. In *Proc. of the European Simulation Symposium*.

Uhrmacher, A. M., P. Tyschler, and D. Tyschler. 1997. Concepts of object and agent–oriented simulation. *Tran. of the Society for Computer Simulation* 14 (2): 59–67.

AUTHOR BIOGRAPHIES

PATRICK F. RILEY is a Ph.D. student at Carnegie Mellon University. As an artificial intelligence researcher, he focuses on agent-based coaching, where one agent instructs another. His interest in distributed simulation grew out of his long involvement with the RoboCup robot soccer research initiative. He can be reached at <pfr@cs.cmu.edu> and his web page is <http://www.cs.cmu.edu/~pfr>.

GEORGE F. RILEY is an Assistant Professor of Electrical and Computer Engineering at the Georgia Institute of Technology. He received his Ph.D. in computer science from the Georgia Institute of Technology, College of Computing, in August 2001. His research interests are large–scale simulation using distributed simulation methods. He is the developer of *Parallel/Distributed ns2* (*pdns*), and the Georgia Tech Network Simulator (*GTNetS*). He can be reached via email at <riley@ece.gatech.edu>.