New Advances in Asynchronous Agent-based Scheduling

Jack Harris[1] **and Matthias Scheutz**[2]
[1]School of Informatics and Computing, Indiana University, Bloomington, IN 47404, USA
[2]Department of Computer Science, Tufts University, Medford, MA 02155, USA

Abstract— *Traditional discrete event simulations enumerate the event-space in a sequential manner to guarantee the consistency of the simulation. Rrecent asynchronous agent-based scheduling work has demonstrated that it is also possible to achieve consistent simulations under certain constraints even when all agents are at different time steps. This paper extends asynchronous event-based scheduling for agent-based simulation by introducing a scheduling policy based on the notion of* Dynamic Goal-based Agent Prioritization *(D-GAP) which provides the opportunity for evaluating a simulation signifcantly faster and with fewer computational steps.*

Keywords: parallel scheduling, discrete event simulation, asynchronous scheduling policies, scheduler efficiency

1. Introduction

Scheduling in traditional discrete event simulation enumerates the event-space sequentially by updating all entities at time t before evaluating any entity at the next time step t+1. This serial temporal updating limits the utility of large-scale computational resources by requiring parallel processes to synchronize at every time step in order to maintain a consistent state configuration. Scheutz and Schemerhorn defined a methodology to identify entity sets that are "update-independent" and could temporally advance in the simulation without requiring synchronization updates, thereby maximizing parallel processing for discrete event simulations [1]. Scheutz and Harris, utilizing the notion of asynchronous discrete event scheduling, published a set of policies explicitly aimed at minimizing the runtime of distributed agent-based simulations [2]. This paper builds upon that prior work. First, we introduce an important performance optimization to the previously published general asynchronous scheduling algorithm. This improvement identifies the minimum set of agent state updates required by simulations to asynchronously advance a given agent to the next time step. Secondly, we introduce a novel scheduling policy, *Dynamic Goal-based Agent Prioritization* (D-GAP). This method utilizes a directed search technique to explore a simulation's discrete event-space targeting the events that will lead to simulation completion as defined by the modeler. Together, these two advances further optimize discrete event simulations for both serial and parallel execution environments by evaluating a simulation faster and with fewer computational steps.

2. Background and Related Work

The study of discrete event simulation has been driven by the need to efficiently model complex system interactions. To facilitate the development and execution of these models, formalisms such as the Discrete Event System Specification (DEVS) have been created [3]. Parallel and distributed resources are commonly leveraged to increase the performance of model execution runtime environments for DEVS and other discrete event simulators. (See [4] for an example of an agent-based simulation modeled using DEVS methodology and executed in parallel using the High Level Architecture (HLA) distributed simulation protocol.)

Though these discrete event systems utilize parallel resources, their efficiency is reduced by the need to continuously synchronize the distributed resources and update each agent in the simulation at each time step. It is sometimes interesting (or required) to evaluate the state of every element in a simulation at simulation termination, but it is often common for a modeler to only be interested in a small set of critical simulation elements (e.g., whether one agent reaches a particular location in the environment)..

In 2006, Scheutz and Schermerhorn introduced the notion of "update-independence" for segregating groups of agents across distributed resource pools and for determining how long they could operate in isolation without sharing agent state information. Formally, an agent A_1 is update-independent from A_2 if A_1 will advance from configuration state C at time t to C' at t+1 regardless of the existence of A_2 in the simulation. Fortunately, it is possible to identify some forms of update-independency without actually advancing a simulation from t to t+1 provided that there is some a priori constraints on agent interactions (e.g., for spatial agent-based simulation: maximum velocities, sensory ranges and effector ranges of agents). For example, if an agent A_1 is outside of agent A_2's maximum interaction range and A_2 is outside of A_1's maximum sensor range then A_1 is update-independent of A_2. However, the reciprocal relation must also exist for A_2 to be independent of A_1. Both A_1 and A_2 must be mutually update-independent for either to update without causing a misconfiguration in the simulation. Clearly, it is possible that A_1 would not be affected by A_2 even if it was not outside of these ranges, but this cannot be known for certain without executing the simulation. A conservative estimation of dependence in a simulation will always ensure consistent configurations are produced.

The notion of an agent's "event-horizon" was introduced to facilitate update-independence determination. An "event-horizon" defines the spatial region containing all possible positions for that agent in a future time step. This region is determined by assuming an agent traveled at its maximum velocity from current simulation time to the projected future time in all directions simultaneously. This assumption allows "event-horizons" to be calculated using only simple geometry. Using "event-horizons", it is possible to identify agents that must be update-independent from other agents even when they exist at different simulation times.

When determining update-independence and projecting "event-horizons" of an agent, a subset of interdependent agents are identified for a given time step. This subset of agents defines a local world that could function independently from the rest of the agents in the simulation. This local world is essentially a "transitive closure" of dependency, where dependency is recursively identified for a given agent and time. Scheutz and Schemerhorn exploited this characteristic of local world independence and used it to aid the parallelization of agent-based simulations. By distributing these sets of agents across different computational resources, performance gains were achieved. These parallel units could update asynchronously until the "event-horizons" of agents inside and outside the closure intersect. When such an intersection occurs the independence of the sets is removed and the agents at the future time step become blocked and unable to continue to advance asynchronously without additional information. Therefore, to maintain a consistent simulation, the asynchronous scheduler must never run an agent asynchronously beyond the point of being blocked [1].

The formalism and correctness proofs of Scheutz and Schermerhorn provided the foundation for future agent-based asynchronous discrete event scheduler research. Subsequent research by Scheutz and Harris explored new update scheduling algorithms to optimize the discrete-event scheduler and minimize the runtime of simulations [2]. From that work novel asynchronous agent-based scheduling policies were presented; each optimizing different heuristics. For example, the $Remote - Event - First$ policy attempted to minimize simulation runtimes by minimizing the occurrences of blocked agents in a distributed context. This was accomplished by preemptively advancing agents whose projected "event-horizons" would first interact with agents on other parallel systems.

3. Improving Asynchronous Scheduling

This section outlines efficiency improvements made to the general asynchronous scheduling algorithm originally proposed by Scheutz and Harris [2]. These advancements focus on minimizing the number of dependent agent updates required to advance a given agent asynchronously in the simulation. Before the new algorithm is proposed, the original methodology is discussed as a counterpoint. Following the description of the new algorithm, a proof for correctness is presented.

Algorithm 3.1: ORIGINAL METHOD(W)

procedure ISCOMPLETE(W)
return (has terminating criterion been met)

procedure PICK(W)
comment: select and return an agent to update

procedure GETTRANSITIVECLOSUREFOR(a, t, W, S)
comment: gets a's recursively dependent set at time t
for each $d \in senseOrAffectAt(a, t, W)$ **and** $d \notin S$
do $S \leftarrow S \cup \{d\} \cup$ GETTRANSITIVECLOSUREFOR(d, t, W, S)
return (S)

procedure ISOLATEDEPENDENTSUBSET(a, W)
$Ta \leftarrow$ GETTRANSITIVECLOSUREFOR($a, time(a), W, \emptyset$)
return (Ta)

procedure UPDATESET(S)
comment: advance agents youngest to oldest
$St \leftarrow sortByLocalTime$(S)
for $i \leftarrow 0$ **to** $size(St) - 1$
do if $i < size(St) - 1$

then { **if** $time(St[i]) <= time(St[i+1]$
  then { UPDATEAGENT($St[i], S$); $i \leftarrow 0$
  else UPDATEAGENT($St[i], S$)

procedure UPDATEAGENT($a, transitiveClosureForA$)
comment: transitions a's state from time(a) to time(a)+1

main
repeat
$a \leftarrow Pick(W)$
$S \leftarrow IsolateDependentSubset(a, W)$
UPDATESET(S)
until ISCOMPLETE(W)

3.1 Full Transitive Closure Update

The original algorithm, *Full Transitive Closure Update*, ensured consistency in simulations by fully synchronizing a set of agents when a dependency relationship was identified. For example, suppose agent A_1 is at time t and agent A_2 is at time t_0 | if agent A_1's sensory range intersects with the projected "event-horizon" of agent A_2 then agent A_2 would have to advance to time t before agent A_1 could update. Furthermore, this update requirement for agent A_2 is recursively required for any agents that agent A_2 could potentially have interacted with when projected to time t. Essentially, the sum of all A_1's recursive dependencies when projected to time t would have to be updated to time t before A_1 could advance. (See Algorithm 3.1 above for the abstracted pseudocode description.)

To maintain a consistent simulation configuration, the original method updates the entire transitive closure to the same time step; however, this level of synchronization is not necessary. When utilizing a *Full Transitive*

Closure Update, notice that an agent returned in the $IsolateDependentSubset$ calculation will be updated to t+1 even if the agent's actions removes the possibility of any future interactions with other agents after a single time step. For example, if the agent travels in the opposite direction its projected "event-horizon" would look very different after just a single update.

Algorithm 3.2: NEW METHOD(W)

procedure ISCOMPLETE(W)
return (has terminating criterion been met)

procedure PICK(W)
comment: select and return an agent to update

procedure INCREMENTALDEPENDENTSET(a, t, W, S)
comment: gets a's incrementally dependent set
for each $d \in senseOrAffectAt(a, t, W)$ **and** $d \notin S$
do $\begin{cases} i \leftarrow time(d) \\ S \leftarrow S \cup d \\ S \leftarrow S \cup \text{INCREMENTALDEPENDENTSET}(d, i, W, S) \end{cases}$
return (S)

procedure ISOLATEDEPENDENTSUBSET(a, W)
comment: isolate youngest dependent subset
$D \leftarrow \text{INCREMENTALDEPENDENTSET}(a, time(a), W, \emptyset)$
if for all $b : b \in D$ **and** $time(a) = time(b)$
then return (D)
$Y \leftarrow getYoungestElementsInSet(D)$
return (Y)

procedure UPDATESET(S)
comment: advance agents in the temporally synchronized set
for $i \leftarrow 0$ **to** $size(S) - 1$
do UPDATEAGENT($St[i], S$)

procedure UPDATEAGENT($a, transitiveClosureForA$)
comment: transitions a's state from time(a) to time(a)+1

main
repeat
$a \leftarrow Pick(W)$
$S \leftarrow IsolateDependentSubset(a, W)$
UPDATESET(S)
until ISCOMPLETE(W)

3.2 Incremental Dependency Removal

The new algorithm, *Incremental Dependency Removal*, 'loosens' the constraints on an agent's update-independence by updating only a subset of the agents identified by the original *Full Transitive Closure Update* method a single time step on each scheduler iteration. This allows for the possibility that dependencies between agents will be removed with the additional state information provided by the incremental update. Like the previous algorithm, an agent is identified to update asynchronously at the beginning of each scheduling loop; however, unlike the previous algorithm, that agent will not update unless every agent in its transitive closure is at the same time step as that agent; otherwise, only a subset of that transitive closure will run (see Algorithm 3.2).

This means that after picking an agent to advance (A), we first identify the youngest agents within agent A's $IncrementalDependentSet$. The $IncrementalDependentSet$ is generated by recursively identifying partial agent dependencies starting with agent A. This is different from at transitive closure calculation only in that the transitive closure calculates all dependencies projected to agent A's time, while $IncrementalDependentSet$ calculates the recursive dependency relative to the local time of each agent. From this $IncrementalDependentSet$, we then obtain the set of youngest dependent agents (Y). Note that Y is update-independent, since all agents in the transitive closure for any agent in Y is also contained in Y. (A proof sketch is provided below.) The agents in Y are then advanced one time step in the simulation. For agent A to advance, it would have to be selected by the scheduler's $Pick$ method until it no longer has dependencies remaining in the past and, therefore, would advance as a member of set Y.

To prove correctness of *Incremental Dependency Removal* we must show that the agents returned from $IsolateDependentSubset$ are update-independent of all other agents in the simulation. Note that $IsolateDependentSubset$ selects the youngest members (Y) returned from $IncrementalDependentSet$; therefore these members must all be at the same time step $t = time(youngest)$. Since dependency calculations were made for each of these agents in Y (because $IncrementalDependentSet$ calls $senseOrAffectAt$ for each of them and their dependents), it follows that these dependents must all be at the same time step. Now observe that $IncrementalDependentSet$ is the same algorithm as $GetTransitiveClosureFor$ when the time of each agent is the same, since each recursive call will pass in the same value (t) for the time parameter. Therefore, $IncrementalDependentSet$ contains a transitive closure for each of the youngest agents in that set and no member of these transitive closures exist at a time step other than t. Furthermore, selecting all the youngest agents at time t results in a set with no external dependencies (i.e., an update-independent set).

3.3 New Dynamic Goal-based Agent Prioritization (D-GAP) Policy

Past asynchronous scheduling policies were designed to optimize system resources and minimize simulation runtimes in parallel environments. However, often times the schedulers would pick suboptimal agents to advance asynchronously through the simulation due to the fact that they had no knowledge of the simulation goals or an agent's likelihood of achieving those goals. This issue motivated the idea of biasing the scheduler's agent selection process by

dynamically prioritizing agents based on their probability of achieving a desired goal for the simulation. While still maintaining all the constraints required for ensuring consistent simulation configurations, the *Dynamic Goal-based Agent Prioritization Policy* (D-GAP) allows agent priorities to be adjusted during a simulation execution thereby influencing which agent will advance sooner.

The D-GAP policy, like any other asynchronous scheduling policy, will not speed up every simulation. For example, if a modeler wants to know the cause of death for all agents in a simulation, the D-GAP policy would have to make the same number of updates as a traditional discrete event scheduler. However, in a distributed context it may still be beneficial to run different asynchronous policies for maximizing parallel resources and ultimately reaching simulation completion faster. However, there are many scenarios when simulations are carried out with a more refined termination condition or goal for a simulation. For example, suppose a modeler designed a simulation experiment with the intent to understand what the cause of death will be for a particular agent. The modeller would set the terminating condition for the simulation to be when that partial agent is no longer alive. For this event to occur it is quite likely that not all agents need to be updated to the same simulation cycle, but instead only the agents required for the particular event to occur. The ideal scheduling policy for this example would be to utilize the D-GAP policy.

In situations involving a subset of agents from the simulation, where a particular event or phenomenon is being studied the D-GAP policy can greatly improve a scheduler's efficiency and lead to much faster runtimes. Setting the agent of interest to a higher priority will let the scheduler identify the agents to update that will push the agent of interest through the simulation event-space fastest. This will ultimately lead to any phenomena relating to those higher priority agents to also take place sooner since less superfluous updating of other agents would occur. The agents of interest do not exist separate of their environment and other agents; therefore, other agents with lower priority would still need to be updated. These lower priority agents would update when the agents of interest progress far enough into the simulation timeline that they required additional updated state information from agents at lower priority levels.

4. Evaluation Method

The performances of the D-GAP policy and the new asynchronous scheduler optimizations were measured using a new standard metric for evaluating asynchronous agent-based schedulers. Scheduling efficiency is determined by comparing the number of agent updates conducted in the new methods with what would have been required using a traditional sequential discrete event scheduler. The number of agent updates can be calculated by incrementing a counter each time an agent's update function is called or by summing the local simulation times of each agent in the simulation. This method of comparing the entity's *update count* is preferable to simply comparing runtimes because the former efficiency metric is more general in that it is not dependent on confounding factors such as simulation-unique agent computational costs or communication delays of a particular distributed context. Therefore the evaluation of the D-GAP policy and new asynchronous scheduler updates focused on comparing the three different configurations running a simulation using: (1) a traditional sequential discrete event scheduler, (2) the D-GAP policy with the original asynchronous scheduler, and (3) the D-GAP policy with the improved asynchronous scheduler.

To evaluate the scheduling policies, an a-life simulation was constructed within the SimWorld agent-based simulation framework [12]. The SimWorld system provides an extensible framework for authoring agent-based simulations complete with an integrated reusable asynchronous scheduling system. This scheduler was modified to include the new agent-based prioritization policy as well as the ability to use the incremental dependency removal method. The Alife simulation consisted of a modified version of Scheutz, Harris and Boyd's computational agent-based model of biological model organism *Hyla versicolor* ("gray treefrog") that was originally used to identify the dominant mating strategy of these animals [13].

In the model, each male calls at a given rate and females select the closest mate that exceeds some threshold of call quality. The distribution of the agents can be observed in Figure 1. In this modified configuration the simulations were run until a particular agent of interest (i.e., male2) mated. After each scheduler iteration, the SimWorld scheduler checks if the simulation should terminate (i.e., if male2 has mated). Also at this time the agent update priorities were dynamically modified to bias the scheduler towards updating agents that would bring about the terminating condition.

5. Case Study: Results

This section contains the results of the three different run conditions discussed in the previous section. Results consist of an image for the final state of each simulation configuration as well as an efficiency metric, *update count*, presented for each simulation. Finally, an efficiency comparison is made between the three conditions.

a) Traditional Sequential Discrete Event Simulation.: The base case for this comparison was to execute the simulation using a traditional sequential discrete event system. This provides a total picture of all agent updates in the simulation, including those of interest and those that are irrelevant to the phenomenon of study. After running the initial base case of the simulation (see top image in Figure 1) we see that the simulation terminates after 49 time steps with female9 mating with the agent of interest, male2. To reach

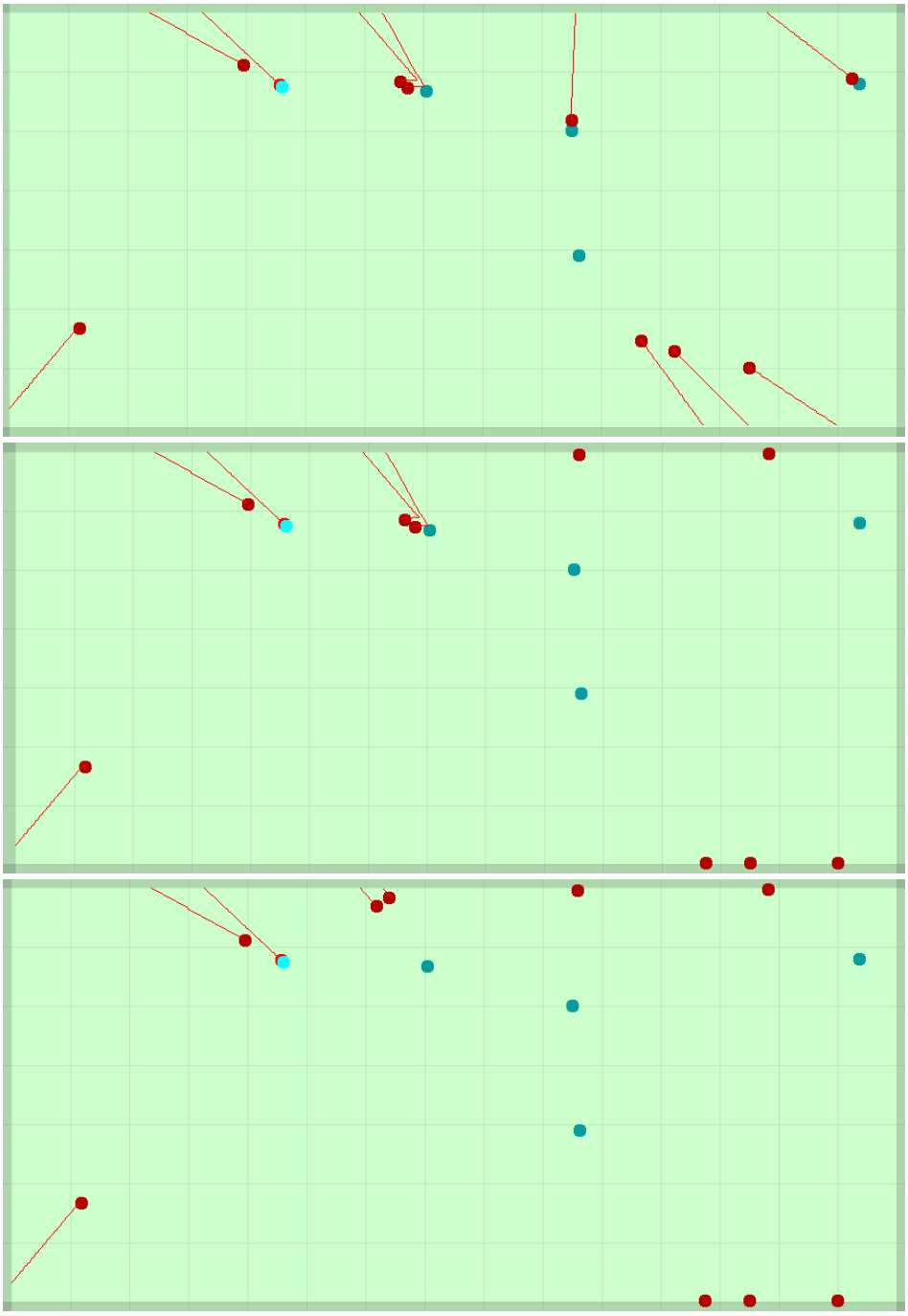

Fig. 1: *Top:* Traditional Sequential Discrete Event Simulation; the final configuration of the Alife tree-frog mating simulation. Female tree frogs (red) approach male tree frogs (blue) based on proximity and male call quality. Ultimately male2 mates with female9 (light blue) at time step 49 after executing **735** agent updates. *Middle:* D-GAP policy + Original *Full Transitive Closure Update* Asynchronous Scheduler. *Bottom:* D-GAP policy + New *Incremental Dependency Removal* Asynchronous Scheduler

the simulation's terminating condition (i.e. male2 finding a mate), **735** *agent updates* had to be calculated. From this image we see that a total of 3 females set out to attempt to mate with male2 at the beginning. By the time that the mating occured at time step 49, we can see that 2 other females had also chosen and began to pursue male2 for a

mate, however we also can notice that most of the other agents in the simulation have no bearing on male2.

b) D-GAP policy + Original 'Full Transitive Closure Update' Asynchronous Scheduler.: The simulation was then reinitialized using the exact same conditions and terminating criteria using the D-GAP policy with the originally published asynchronous scheduling algorithm. Since the simulation only updated agents in a way to ensure no inconsistent simulation states could emerge, the predictions of the simulation were identical to that of the base case; that is, male2 mated at cycle 49 with female9. However, in this case it only took **347** *agent updates* to generate this mating prediction. This algorithm, therefore, produced a scheduling efficiency gain of almost 53% for this scenario $[(735-347)/735*100]$. This a massive speed up given that the same number of processors were used in both cases and the predictive models of the simulation were not changed in anyway. When studying the middle image of Figure 1, we notice that most of the agents in the simulation did not advance far into the simulation's timeline. We will now walk through the logic by which some agents were updated and others were not. Upon simulation initialization all agents had the same priority level (unset) and all agents therefore were allowed to advance synchronously one time step. Following this initial step the *isComplete* check ran, returned false, and set the priority of the 3 leftmost females to have a higher priority since they had chosen to pursue the agent of interest Since females in this simulation do not interact with each other, these agents were *update-independent* until they could interact with a male. Therefore, the following simulation iterations allowed these three females to update asynchronously until female9 entered the interaction range of male2. Since male2 was in the past, he would need to be updated before female9 could advance and mating could take place. However, male2 could not advance independently to the female9's local time step since the "event-horizons" of two other females when extrapolated to female9's time could have potentially caused an earlier interaction with male2. Furthermore, these additional females could also have interacted with an another male when projected to female9's local time step, and, therefore, this male would also have to be included into the group that defined female9's transitive closure which would need to be updated before female9 could advance. From the simulation graphic we see that these additional female agents chose to travel in the opposite direction from male2 and essentially removed themselves from blocking female9's *update-independence*. Unfortunately the original general asynchronous scheduling algorithm still updated these agents to the female9's time. Following these updates female9 was free to proceed to mate with male2 at time step 49.

c) D-GAP policy + New Incremental Dependency Removal Asynchronous Scheduler.: Like the previous condition, the final condition utilized the D-GAP policy for selecting agents to run. However, in this case the new *Incremental Dependency Removal* optimization to the general asynchronous scheduling algorithm was also implemented resulting in the calculation of even fewer *agent updates*. Notice from the bottom image of Figure 1 that the two middle females only progressed a few time steps into the simulation's timeline as compared with the previous simulation (middle image of Figure 1). In this condition only **221** *agent updates* had to be calculated to produce the same predictions as the previous 2 cases. This results in a scheduling efficiency gain of almost 70% $[(735-221)/735*100]$.

The logical event trace for this simulation starts out similarly to the previous case. All agents update one cycle since no agent priority is initially set. Following that, the 3 females that chose to pursue the agent of interest (male2) receive higher priority in the system. The 3 female agents can all update independently until female9 becomes dependent on male2's state prior to attempting to mate. Male2 was then required to update to the same time step as female9. However, unlike the previous case, male2 was allowed to update by only one time step per scheduler cycle. At each scheduler cycle the high priority agents including female9 were selected. This again resulted in a male2 update and this continued until male2 was no longer update-independent with respect to the top middle females. The additional females were incrementally run for one step. The process continued with female9 requiring male2 to update, but in some cases male2 was able to update independently through the timeline since the top middle females actually move away from the agent of interest. The result of this incremental update strategy was that the two top middle females did not have to progress through the simulation timeline nearly as far and did not require the additional male to update at all (see bottom image of Figure 1).

5.1 Efficiency Comparison

Ultimately this case study illustrates a large performance gain from using the D-GAP algorithm and addition efficiency gain as a result of the optimization added to the general asynchronous scheduling algorithm (see Figure 2).

The D-GAP policy when coupled with the original asynchronous algorithm drastically outperforms the traditional synchronous discrete event simulator for this scenario. This improvement comes from the ability of the policy to prune unnecessary computations from the timeline based on prioritizing agents that are most likely to accomplish the research goals of the simulation. Essentially the timeline is expanded in a greedy manner such that the important events occur sooner and with less updating of agent configurations.

The additional efficiency gain achieved from using the *Incremental Dependency Removal* stems from the way this algorithm handles updating dependent agents. Instead of blindly updating agents to the point of projected possible

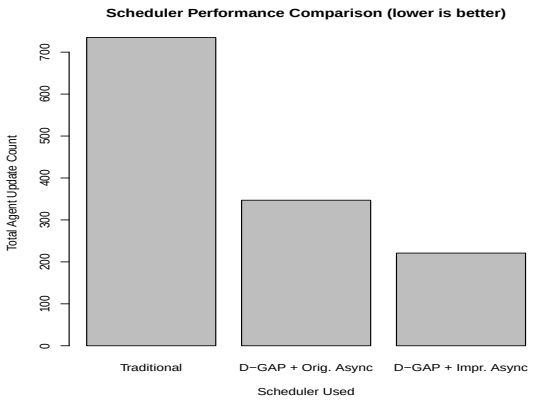

Fig. 2: Comparison of the efficiency of the 3 scheduling methods in terms of the number of *agent updates* that had to be accomplished before reaching simulation completion. The D-GAP policy with the improved asynchronous scheduling algorithm produced the most efficient runs.

interaction, dependent agents are updated only as far as necessary. Instead of updating all the agents in female9's *original* transitive closure, only the most dependent agents had to be updated incrementally. This allowed them the possibility of removing themselves from future dependencies and the need to update as much.

6. Discussion and Conclusion

The power of the D-GAP policy, like any greedy heuristic search, is only as strong as the heuristic used. In the case study, the priority assignment was clearly connected to the agent that could bring about the terminating condition. However, if priority was assigned to agents in an unrelated way to the goals of the simulation or even in an opposite way, the simulation could potentially take longer than traditional sequential scheduling. In the extreme case, it is possible for agent prioritization to be assigned such that the simulation never terminates. For example, if agent was given the highest priority in the simulation and that agent moved at maximum velocity away from the rest of the agents, it is possible that it would not interact with any other agent's projected "event-horizons" and maintain update independence indefinitely. This would lead to an agent being updated each simulation cycle that would never bring about simulation termination.

Therefore, the assignment of priority to agents must be done in an admissible way that guarantees simulation termination. There are many ways that this can be implemented. For example, adding stochasticity to the priority assignment will guarantee simulation termination even in the worst-case scenario when agent prioritization is completely assigned backwards. Another method would be to progressively penalize agents too far in the future so that there is a maximum time gap between the oldest and youngest agents in the simulation. The latter provides a bounded level of asynchrony and greedy search while maintaining the assurance of simulation termination.

Asynchronous discrete event scheduling is an extremely new concept. There have only been a handful of algorithms and policies developed to exploit the advantages of asynchronous scheduling. Policies developed so far have either optimized the characteristics of the distributed environment in which they were implemented (e.g. $Remote - Event - First$ or $Youngest - Unblocked - First$ [2]) or have attempted to optimize the scheduling using some heuristic to bias local scheduling of agents (e.g., the D-GAP policy described in this paper). Interesting future work would include policies that consider both the goals of the simulation and the distributed environment in which they are executed when making decisions on which agent to asynchronously guide through the simulation space first.

References

[1] M. Scheutz and P. Schermerhorn, "Adaptive algorithms for the dynamic distribution and parallel execution of agent-based models," *Journal of Parallel and Distributed Computing*, vol. 66, no. 8, pp. 1037–1051, 2006.

[2] M. Scheutz and J. Harris, "Adaptive scheduling algorithms for the dynamic distribution and parallel execution of spatial agent-based models," in *Parallel and Distributed Computational Intelligence*, ser. Studies in Computational Intelligence, F. Fernández de Vega and E. Cantú-Paz, Eds. Springer, 2010, vol. 269, pp. 207–233.

[3] B. P. Zeigler, H. Praehofer, and T. G. Kim, *Theory of Modeling and Simulation, Second Edition*, 2nd ed. Academic Press, Jan. 2000.

[4] M. Lees, B. Logan, and G. Theodoropoulos, "Distributed simulation of agent-based systems with HLA," *ACM Transactions on Modeling and Computer Simulation*, vol. 17, no. 3, p. 11, 2007.

[5] M. B. Ptacek, "Calling sites used by male gray treefrogs, Hyla versicolor and Hyla chrysoscelis, in sympatry and allopatry in Missouri," *Herpetologica*, vol. 48, no. 4, pp. 373–382, 1992.

[6] G. M. Fellers, "Aggression, territoriality, and mating-behavior in North-American treefrogs," *Animal Behaviour*, vol. 27, no. FEB, pp. 107–119, 1979.

[7] ——, "Mate selection in the gray treefrog, Hyla-versicolor," *Copeia*, no. 2, pp. 286–290, 1979.

[8] M. E. Ritke and R. D. Semlitsch, "Mating-behavior and determinants of male mating success in the gray treefrog, Hyla-chrysoscelis," *Canadian Journal of Zoology-Revue Canadienne De Zoologie*, vol. 69, no. 1, pp. 246–250, 1991.

[9] O. M. Beckers and J. Schul, "Phonotaxis in Hyla versicolor (Anura, Hylidae): the effect of absolute call amplitude," *Journal of Comparative Physiology a – Neuroethology Sensory Neural and Behavioral Physiology*, vol. 190, no. 11, pp. 869–876, 2004.

[10] S. L. Bush, H. C. Gerhardt, and J. Schul, "Pattern recognition and call preferences in treefrogs (Anura : Hylidae): a quantitative analysis using a no-choice paradigm," *Animal Behaviour*, vol. 63, pp. 7–14, 2002.

[11] H. C. Gerhardt, S. D. Tanner, C. M. Corrigan, and H. C. Walton, "Female preference functions based on call duration in the gray tree frog (Hyla versicolor)," *Behavioral Ecology*, vol. 11, no. 6, pp. 663–669, 2000.

[12] M. Scheutz, P. Schermerhorn, R. Connaughton, and A. Dingler, "Swages–an extendable parallel grid experimentation system for large-scale agent-based alife simulations," in *Proceedings of Artificial Life X*, June 2006, pp. 412–418.

[13] M. Scheutz, J. Harris, and S. Boyd, "How to pick the right one: Investigating tradeoffs among female mate choice strategies in treefrogs," in *Proceedings of the Simulation of Adaptive Behavior 2010*, 2010, pp. 618–627.