SWAGES - An Extendable Distributed Experimentation System for Large-Scale Agent-Based Alife Simulations

M. Scheutz, P. Schermerhorn, R. Connaughton, A. Dingler
Artificial Intelligence and Robotics Laboratory
Department of Computer Science and Engineering
University of Notre Dame
Notre Dame, IN 46556, USA
mscheutz@cse.nd.edu

Abstract

We introduce SWAGES, a distributed agent-based Alife simulation and experimentation environment that uses automatic dynamic parallelization and distribution of simulations in heterogeneous computing environments to minimize simulation times. SWAGES allows for multi-language agent definitions, uses a general plug-in architecture for external physical and graphical engines to augment the integrated SimWorld simulation environment, and includes extensive data collection and analysis mechanisms, including filters and scripts for external statistics and visualization tools. Moreover, it provides a very flexible experiment scheduler with a simple, web-based interface and automatic fault detection and error recovery mechanisms for running large-scale simulation experiments.

Introduction

Agent-based simulations have become increasingly important in the study of complex systems. While many simulation environments for the study of Alife models have been proposed over the last decade (e.g., (Mössinger et al., 1995; Collier, 2003; Minar et al., 1996; Minar et al., 1999; Ray, 2001; Ofria and Wilke, 2004)), with their different individual strengths and weaknesses, there is currently no simulation environment that supports the automatic parallelization of agent-based simulations and dynamical distribution of simulations over a set of heterogeneous, dynamically changing hosts. Moreover, most simulation environments limit users in the way they can design simulations (either a special language is provided, or the programming language the simulation environment is written in has to be used). Finally, most simulation environments come with built-in components (e.g., simulation and graphics engines, statistics packages, etc.), which can typically not be replaced.

In this paper, we present our SWAGES environment which addresses all three problems. First, it implements a novel algorithm for automatically parallelizing and distributing agent-based simulations. Second, it allows for agent definitions in a variety of programming languages and provides standard interfaces for calling external functions (either directly or via socket connections). Third, it provides a standard plug-in interface, which can be used to link in any combination of external physics engines (such as ODE) and graphical visualization tools (such as OGRE), and simulation output can either be analyzed and visualized with SWAGES or saved in formats appropriate for external tools (e.g., R or Scilab).

A Brief Overview of SWAGES

SWAGES consists of several heterogeneous, distributed components that cooperate closely at different levels to achieve maximum resource utilization in a dynamic computing environment. It can be generally divided into *server-side* and *client-side* components, where the *server-side* components provide the distributed computation infrastructure, and the *client-side* components provide the communication components and the simulation platform SimWorld. Currently, all server-side components are run on a single host–the *grid server*[1]–while client-side components run on individual *simulation servers*.

Server-side Components

The grid server is the central locus of control of a SWAGES system, running various server-side components to schedule, distribute, start, and monitor the execution of distributed simulations and recover from failures.

The experiment server is responsible for setting up *experiment sets* (possibly consisting of large numbers of individual experiments). Important factors here are generation of initial conditions (unique or identical) across different experiments in a set, priorities of experiments and scheduling parameters, levels of supervision and recovery parameters, format of data collection and location for data storage, statistical analysis of results and output formats, and user notification of progress.

The scheduler is responsible for taking experiments from several priority-based queues (in which new experiments are

[1]Since grid server components are largely autonomous, it is possible to distribute them as well. We are currently on a prototype version with distributed server components to improve performance.

submitted by the experiment server) and starting them on remote hosts. The experiment scheduler will dynamically create experiment data structures for large-scale experiment sets (to avoid memory shortage), and only schedule a new experiment when new hosts are available that are not needed for other experiments to finish.

The client server represent a remote simulation and maintains an open communication channel with the simulation instance, keeping track of the simulation's progress, state, update, and degree of parallelization. It is critical for error detection and recovery: when a simulation crashes (e.g., due to OS problems on its host), is not responding (e.g., due to network problems), or cannot be continued (e.g., because its current host does not meet user-defined criteria for running simulations anymore), the client server can resume the simulation elsewhere based on saved state.

The watchdog implements a second level of supervision which is particularly important for dynamic computing environments where hosts can "disappear" from a pool of usable machines. It regularly checks all clients for progress, terminates clients that are stuck or not responsive, and reschedules simulations either from scratch or from saved states.

The web server provides a simple web-based interface to `SWAGES` that can be used to submit experiments, check their status, perform simple statistics and view the results.

The simhost is the server-side representation of a remote simulation host. It keeps track of the simulations running on it, and is responsible for monitoring the availability of the host computer for simulations based on user-defined criteria (e.g., a restriction might be that simulations can only be run if no console user is logged on).

Client-side Components

The client-side components are responsible for running the actual simulation and communicating its state to the server-side components.

The simclient represents the simulation to the (server-side) client server to which it communicates updates about the simulation. It is responsible for saving the state of the simulation and checking the host it is running on for availability according to user-defined criteria.

`SimWorld` is the default simulation environment used in `SWAGES`. It interacts with the simclient to report simulation statistics and agent data that client servers can use to parallelize simulations (based on host availability).

`SWAGES` works in homogeneous fixed clusters (e.g., Beowulf clusters) and heterogeneous ad-hoc clusters (e.g., individual workstations that can only be used if nobody is logged in) alike. Moreover, it requires no set-up procedures on the host participating in simulation experiments (other than the standardly installed *secure shell tools* for secure, remote login and file transfer) and will run on all operating systems that support the JAVA virtual machine and the Poplog environment (for `SimWorld`). Note that it is possible to use simulation environments other than `SimWorld` in `SWAGES` by simply adapting the *simclient* to the new simulation engine (the server-side interaction stays the same). That way `SWAGES` is not tied to `SimWorld` (and Poplog), but can be used with a variety of simulation environments and thus improve their usability.

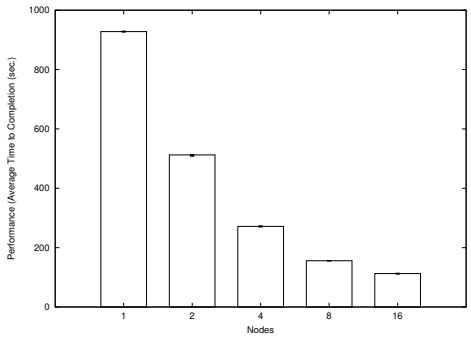

Figure 1: Average time to completion for 1, 2, 4, 8, and 16 nodes for a *swarm task*, where agents must find and gather at the nearest checkpoint as quickly as possible (see (Scheutz and Schermerhorn, 2006) for details). Error bars denote confidence intervals for $\alpha = 0.05$.

The `SimWorld` Environment

`SimWorld` is built on top of the `SimAgent` agent toolkit, a general purpose agent toolkit based on the Poplog environment with the built-in OPS5-style rule interpreter `poprulebase`. It can run in "GUI mode" (an interactive mode for agent development and testing that provides a 2D graphical user interface) and "batch mode" (a non-interactive mode for large-scale simulations) and be used for simulations of any kind of agent in any kind of environment, from simple, grid-based cellular automata to complex environments with deliberative agents, including evolutionary investigations. `SimWorld` provides extensive support for periodic or event-driven data collection (virtually any system parameter can be recorded via a simple specification language, see Step 2 in the next Section). Over the last five years, it has been used successfully in several diverse projects, ranging from the study of the utility of affect for agent control (e.g., (Scheutz and Schermerhorn, 2002)), to various evolutionary investigations (e.g., (Scheutz and Schermerhorn, 2005)), to the study of conflict resolution strategies (Scheutz and Schermerhorn, 2004), and others (e.g., UAV swarms (Schermerhorn and Scheutz, 2005)).

Automatic parallelization is now included in `SimWorld` based on a novel algorithm that distributes simulations over a set of available hosts (Scheutz and Schermerhorn, 2006). The algorithm can either run in lock-step mode (i.e., updating all parallel simulations one cycle at a time), or in asynchronous mode where individual

Figure 2: Simple reactive ant-like agents on their hunt for food in OGRE and in the simple 2D `SimWorld` GUI (brown circles with black dots) superimposed on the upper right.

simulations update independently for as many cycles as possible until information from other simulations is needed. The asynchronous algorithm utilizes spatial information available about the "sphere of influence" of agents in spatial agent-based models such as SWARMS, ANTS, and many others, where agents can affect their environment only within a given range. By exploiting the information about the "potential impact" an agent can have on its environment, the algorithm can automatically split a given simulation S with n agents into k parallel simulations $S_1, S_2, \ldots, S_k$ with $n_1, n_2, \ldots, n_k$ agents, distribute them over k hosts (via the `SWAGES` server-side components) and update them asynchronously.[2] In the current implementation, each parallel simulation S_i will send updates of the states of its agents to the client server, which stores them in a data structure shared by all client servers that belong to the same parallelized simulation. The time-savings of the algorithm in cases of SWARM simulations where good splits of agents can be computed (e.g., because different groups of swarms are located far apart, so that they cannot influence each other) are shown in Fig. 1. The results reported are averages over 20 simulation runs of 100 cycles each. Each of the 20 initial conditions was simulated using 1, 2, 4, 8, and 16 nodes in a dedicated Linux cluster of dual 2.4GHz Xeons with 1GB RAM. The times reported include all overhead of starting and finishing `SWAGES`, as well as distributing the simulations when more than one node is used.

[2]For space reasons, we cannot elaborate on the details of determining the split of n agents into $n_1, n_2, \ldots, n_k$ agents, which ideally will take overlaps of their sphere of influence into account, compute the transitive closures of the respective overlaps, and attempt to distribute each transitive closure onto a separate host.

Support for external "plug-ins" (e.g., to link in external simulation and visualization components such as external physics and graphics engines) is supported via an open "plug-in architecture". We briefly mention two engines that have been connected to `SimWorld`: the *Open Dynamics Engine* (ODE), a physics engine used to provide efficient collision and friction detection as well as realistic rigid-body motion, and the *Object-Oriented Graphics Rendering Engine* (OGRE), a three-dimensional graphics rendering package. The integration of external engines is achieved via segments of shared memory (together with inter-process synchronization mechanisms) that allow `SimWorld` and external engines to share basic agent features. In the case of a physics engine, for example, the information passed to the engine at each cycle are force vectors representing the forces generated by the agent's controller (via its body); the physics engine then simply returns the updated position and orientation of the agent based on the newly generated force and all other forces that apply. For a graphics engine, location and orientation of each agent (plus additional information about the appearance of the agent, etc.) are shared (see Fig. 2 for a comparison of `SimWorld` and OGRE visualization). Both physics and graphics engines can be included at the same time, and moreover, designers can selectively choose which agent to update and render via the external engines (thus allowing for non-physical information gathering agents that might not need to be displayed).

Simulation Experiments in `SWAGES`

We briefly demonstrate the utility of the `SWAGES` environment by stepping through the process of defining, running, and analyzing a large-scale set of simulation experiments.

Step 1: Developing an agent model. Agents can be defined in any of the available programming languages in Poplog (i.e., Pop11, Prolog, ML, Scheme, C-Lisp) or *via* condition-action rules in `poprulebase`. Additional support for external function calls to code written in other programming languages is provided via a C-function call interface as well as JAVA socket serialization mechanisms that can serialize and de-serialize Pop11 and JAVA objects, to the extent that they are similar in nature.

Fig. 3 shows an example of parts of the code for a simple reactive agent that can obtain energy from food items. `SimWorld` supports interactive development of agent code by virtue of the incremental Pop11 compiler that allows for dynamic code modifications while the simulation is running. For example, it is possible to replace the above "eating method" with an "empty method" at cycle 34 in "GUI mode" (e.g., to study the effects of refusing to eat):

```
34 ? pop11 'define :method eating(a:simple_agent); enddefine'
```

Similarly, it is possible to define new agent types, add and remove agents from a running simulation, and track

```
define :class thing; is sim_object; /* base class */
  slot geometry == undef;
  slot position = undef;
  slot heading = undef;
  slot mass = undef;
enddefine;

define :class simple_agent; is thing; /* derived class */
  slot speed = 3;
  slot energy = 1000;
  slot maxenergy = 2000;
  slot food = undef;
  slot intake = 50;
enddefine;

define :ruleset eating_ruleset;    /* rules for eating */
  RULE start_eating
    [ingest food] [NOT halting_for eating]
        ==>
    [do halting_for eating]
    [do eating]
    [STOP]

  RULE end_eating
    [NOT ingest food] [halting_for eating]
        ==>
    [NOT halting_for eating]
    [STOP]

  RULE continue_eating
    [ingest food] [halting_for eating]
        ==>
    [do eating]
enddefine;

define :method eating(agent:simple_agent);
  if energy(agent) >= maxenergy(agent) then
    remove([halting_for eating]); /* sated, stop eating */
  else /* add energy to agent, remove from source */
    energy(agent) + intake(agent) -> energy(agent);
    energy(food(agent)) - intake(agent)
       -> energy(food(agent));
  endif;
enddefine;
```

Figure 3: The object-oriented structure of agents in `SimWorld`: a base class for "things" and a derived class for a simple agent, together with a rule set for "eating behavior" and a method that implements the "eating" action (defined in Pop11).

```
[[ /*** set up simulation environment ***/
  [initfile 'reactive.p']       /* include user definitions */
  [quitif 10000]                /* stop after 10000 cycles */
  [world WIDTH 1000 HEIGHT 1000]/* limited world */
 [ /*** set up agents ***/
  [simple_agent                 /* use this entity */
    [startup [variate AGENT from 1 to 10 step 1]]
    [record [at death [sim_x][sim_y]]
            [at end [energy]]]
  [food                         /* use this entity */
    [startup [variate FOOD from 10 to 200 step 10]]]]
 [ /*** schedule events ***/
  [random [[variate PROB from 0.1 to 0.5 step 0.05] [food]]]]
 [ /*** swages setup ***/
  [name 'simple' 'resultsdir']  /* where to store */
  [user 'airolab']              /* run as this user */
  [priority 5]                  /* run at medium priority */
  [parallelize]                 /* allow parallelization */
  [ranseed 29187]               /* same initial conditions */
  [replicates 40]               /* across all replicates */
  [watch 120 30]                /* supervise execution */
  [email 'mscheutz@cse.nd.edu'] /* notify user when done */
  [copyimages '/tmp/']          /* temp space for sim state */
  [copystats 'statsdir']]]      /* where to store stats */
```

Figure 4: The four-part setup of an experiment set.

and record the values of their slots (e.g., their energy levels throughout a run).

Step 2: Defining and running experiments. Once a simulation (with its agent models) is ready for experimentation, large-scale experiments that systematically investigate

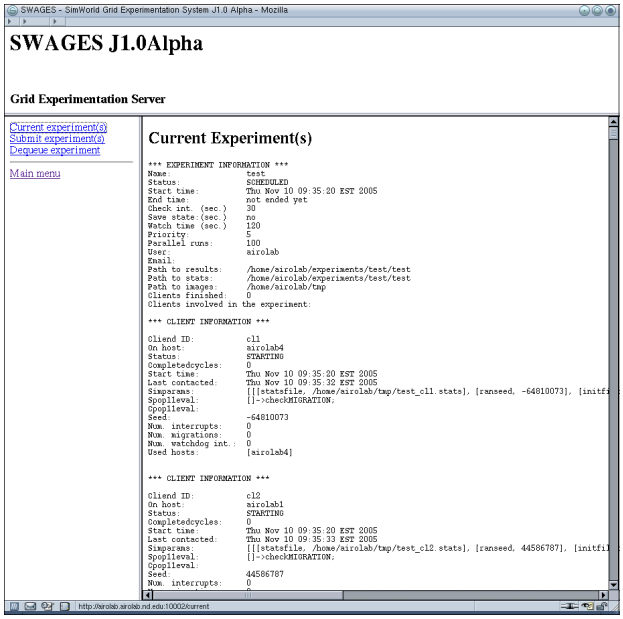

Figure 5: A scheduled experiment set and simulations running concurrently on several hosts.

multi-dimensional parameter spaces can be defined quickly in a simple experiment definition language and scheduled in `SWAGES` *via* a web-based interface. Fig. 4 shows the setup for an experiment set that investigates the survivability of agents. The "variate" keywords specifies the variations of setup parameters (the number of initial agents "AGENT", food items "FOOD", and the probability "PROB" with which new food items are placed in random locations within a 1000 x 1000 area in the environment), for a total of $10 \cdot 20 \cdot 10 \cdot 40 = 80000$ individual experiments based on the three-dimensional parameter space AGENT x FOOD x PROB and the number replicates with different random inital placements of agents and food items (as specified by "startup" and "replicate" keywords). The "record" keyword is used to specify the variables that should be recorded (and when), in this case the location of agents when they run out of energy and the energy level of agents at the end of the simulation (i.e., after 10000 cycles as given by the "quitif" keyword). Experiment definitions can be entered in a web-based interface to schedule experiments in `SWAGES` (Fig. 5 shows the interface depicting scheduled and running experiments).

Step 3: Analyzing the resultant data. `SWAGES` provides basic built-in statistical analysis tools (e.g., for performing simple significance tests), and libraries for data extraction and combination (in various formats including HTML, TeX, and plain text, see Fig. 6 for typical statistical operations that can be performed within the `SWAGES` web-interface). `SWAGES` also provides various export filters and scripts to

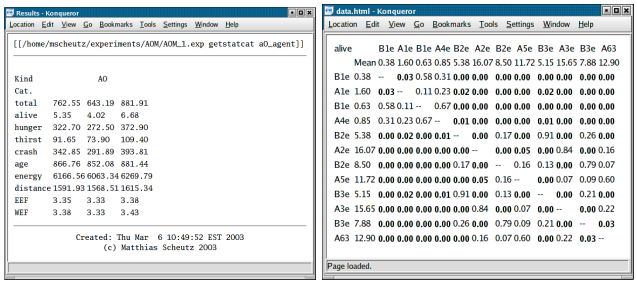

Figure 6: Various result parameters of an experiment with a single agent A0 (left) and summary results comparison (*p*-values) of the average number of survivors among several agents kinds (right), where bold numbers depict significant differences.

interface common open-source statistics, visualization, and scientific computing software (Fig. 7 shows output files for R, results from dynamically generated scripts for gnuplot, and data files for Scilab).

Related Work

Many software environments have been proposed for agent-based and artificial life simulations within the last decade. Here, we can only selectively name a few to show that there is no system like SWAGES that combines (1) the flexibility of designing agents and recording any data in the course of simulation runs in flexible ways, (2) the open plug-in architecture (that allows to link in external simulation and graphical engines and produce output for a variety of open-source statistical analysis and graphical visualization packets), with (3) the automatic parallelization and distribution of large-scale simulations that can be easily defined, scheduled, and observed through a simple web-based interface. General toolkits that support distributed computing (e.g., MACE3J (Gasser and Kakugawa, 2002) and Hive (Minar et al., 1999)) do not typically provide a readily usable simulation environment that supports easy definitions of agents and events, data collection, statistical analysis. On the other hand, agent-based and Alife simulation environments with such support (e.g. XRaptor (Mössinger et al., 1995) do not provide support for distributed computing and/or parallelization of simulations.

And while some extensions to existing simulations add support for distributed computing (e.g., HLA_REPAST (Minson and Theodoropoulos, 2004), which uses HLA to distribute simulations based on the *RePast* toolkit (Collier, 2003)), the distribution is not automatic and proceeds in lock-step, whereas in SWAGES agent designers do not have to include any provisions for parallelization in their code (simply adding the experiment setup keyword "parallelize" (as in Fig. 4) is sufficient for SWAGES to parallelize simulations whenever possible based

```
"N" "Repl" "Spread" "Females" "Time" "Males" "Distance"
1   1      0.8      1         147.83 0.00000 274.512
1   2      0.8      1         175.15 0.05000 276.763
1   3      0.8      1         201.63 0.13333 281.357
1   1      0.8      2         226.10 0.21250 290.448
1   2      0.8      2         245.87 0.25200 300.530
...
```

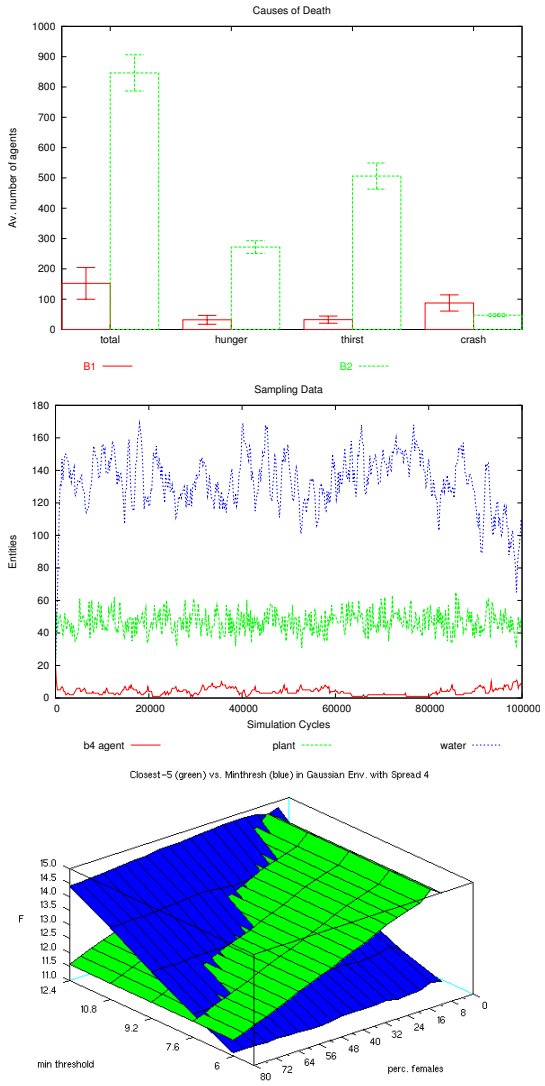

Figure 7: R output file (top), comparison of two agent kinds in terms of causes of death (upper middle), including 95% confidence intervals, population dynamics of an evolutionary experiment showing numbers of all entities at each simulation cycle (lower middle)–both middle graphs were generated via gnuplot–and a performance space in Scilab (bottom) comparing two agent kinds (green and blue) along several dimensions.

on available computational resources).

While most simulation environments provide mechanisms for recording and filters for exporting data (e.g., Swarm (Minar et al., 1996), Hive (Minar et al., 1999),

Repast (Collier, 2003)), they do not reach the flexibility of SWAGES where *any* simulation variable defined in SimWorld (all slots of agents, global SimWorld variables, etc.) and all of the underlying Poplog and OS variables (e.g., memory consumption or system time) can be recorded in multiple ways, possibly performing operations on the data before recording it (e.g., as part of the recording process by virtue of dynamically compiled Pop11 functions specified as part of the experiment start parameters). Moreover, these environments do not provide easy mechanisms for defining, scheduling, and running large-scale experiment sets, with mechanisms for fault-detection and automatic recovery from errors, statistical analysis and data visualization, and export filters for various open source software tools (all of which is supported in SWAGES), nor do they provide an architecture for linking in and utilizing external physics or graphics engines.

Finally, agents designers are limited in the ways in which they can specify agents (typically in the underlying programming language of a toolkit, e.g. C in Tierra (Ray, 2001), C++ in Avida (Ofria and Wilke, 2004), and Java in Hive (Minar et al., 1999), or some interpreted meta-language, e.g., as in Mozart (Peter Van Roy, 1999)), while SimWorld provides many different ways of defining agents (from any of the languages supported by Poplog, to external function calls via libraries or sockets).

Conclusion

We presented a brief overview of our SWAGES system, which provides a flexible, integrated platform for systematic, large-scale agent-based Alife simulation experiments. SWAGES has been successfully employed in at least two dozen research projects over the last five years and has reached a level of maturity which we believe will make it a useful research tool for the Alife community.[3]

References

Collier, N. (2003). RePast: An extensible framework for agent simulation. http://www.econ.iastate.edu/tesfatsi/RepastTutorial.Collier.pdf.

Gasser, L. and Kakugawa, K. (2002). Mace3j: Fast flexible distributed simulation of large, large-grain multi-agent systems. In *Proc. of AAMAS 2002*, pages 745–752.

gnuplot. http://www.gnuplot.info/.

Minar, N., Burkhart, R., Langton, C., and Askenazi, M. (1996). The Swarm simulation system, a toolkit for building multi-agent simulations. http://www.santafe.edu/projects/swarm/overview-/overview.html.

Minar, N., Gray, M., Roup, O., Krikorian, R., and Maes, P. (1999). Hive: Distributed agents for networking things. In *First International Symposium on Agent Systems and Applications (ASA'99)/Third International Symposium on Mobile Agents (MA'99)*, Palm Springs, CA, USA.

Minson, R. and Theodoropoulos, G. (2004). Distributing repast agent based simulations with HLA. In *Proceedings of the 2004 European Simulation Interoperability Workshop*, Edinburgh, UK.

Mössinger, P., Polani, D., Spalt, R., and Uthmann, T. (1995). Xraptor: A synthetic multi-agent environment for evaluation of adaptive control mechanisms. In *EUROSIM*, pages 1229–1234.

ODE. http://www.ode.org.

Ofria, C. and Wilke, C. (2004). Avida: A software platform for research in computational evolutionary biology. *Journal of Artificial Life*, 10(2):191–229.

OGRE. http://www.ogre3d.org.

Peter Van Roy, S. H. (1999). Mozart: A programming system for agent applications. In *International Workshop on Distributed and Internet Programming with Logic and Constraint Languages*.

R-Project. http://www.r-project.org.

Ray, T. (2001). Overview of tierra at atr. In *Techn. Inf. 15, Technologies for Software Evolutionary Systems*.

Schermerhorn, P. and Scheutz, M. (2005). The utility of heterogeneous swarms of simple uavs with limited sensory capacity in detection and tracking tasks. In *IEEE Swarm Intelligence Symposium 2005*.

Scheutz, M. and Schermerhorn, P. (2002). Steps towards a theory of possible trajectories from reactive to deliberative control systems. In Standish, R., editor, *Proceedings of the 8th Conference of Artificial Life*. MIT Press.

Scheutz, M. and Schermerhorn, P. (2004). The more radical, the better: Investigating the utility of aggression in the competition among different agent kinds. In *Proceedings of SAB 2004*. MIT Press.

Scheutz, M. and Schermerhorn, P. (2005). Predicting population dynamics and evolutionary trajectories based on performance evaluations in alife simulations. In *Proceedings of GECCO 2005*.

Scheutz, M. and Schermerhorn, P. (2006). Adaptive algorithms for the dynamic distribution and parallel execution of agent-based models. *Journal of Parallel and Distributed Computing*, page forthcoming.

Scilab. http://www.scilab.org.

[3] SWAGES is available freely at *http://www.nd.edu/~airolab/software*.