MABE (Modular Agent Based Evolver): A Framework for Digital Evolution Research

Clifford Bohm[1,2,*], Nitash C G[2,3] and Arend Hintze[1,2,3]

[1]Department of Integrative Biology, Michigan State University, MI, USA
[2]BEACON Center for the Study of Evolution in Action, Michigan State University, MI, USA
[3]Department of Computer Science and Engineering, Michigan State University, MI, USA
[*] Corresponding Author: cliff@msu.edu

Abstract

A great deal of effort in digital evolution research is invested in developing experimental tools. Because each experiment is different and because the emphasis is on generating results, the tools that are developed are usually not designed to be extendable or multipurpose. Here we present *MABE*, a modular and reconfigurable digital evolution research tool designed to minimize the time from hypotheses generation to hypotheses testing. MABE provides an accessible framework which seeks to increase collaborations and to facilitate reuse by implementing only features that are common to most experiments, while leaving experimentally dependent details up to the user. MABE was initially released in August 2016 and has since then been used to ask questions related to Evolution, Sexual Selection, Psychology, Cognition, Neuroscience, Cooperation, Spatial Navigation and Computer Science.

Introduction

For decades researchers have been turning to digital evolution to ask questions related to biology and engineering. During that time, a number of tools have been written and successfully employed to investigate topics as diverse as sexual selection (Chandler et al. (2013)), predator-prey dynamics (Olson et al. (2013)) , antenna design (Lohn et al. (2005)) and self adapting robots (Cully et al. (2014)) among many others.

One difficulty in Digital Evolution research stems from the need to develop the software used to conduct the research; a process which can be both time consuming and costly. For example, (Frénoy et al. (2013)) investigated the evolution and maintenance of cooperation using AEVOL while (Adami et al. (2012)) asked similar questions using an Evolutionary Game Theory system designed specifically to address their particular research question. While both of these projects addressed different particular issues and had unique requirements, they also had many common requirements such as genomes, lineage tracking, file I/O and other core functionality which resulted in code duplication. Could these projects have been developed in less time if they had been able to share common components? MABE differs from its predecessors in that it was not designed to implement a particular computational system or address particular research questions, rather, MABE is fundamentally a general purpose digital evolution tool which allows for the implementation and evolution of arbitrary neural computational models. MABE does not embody a novel digital evolution methodology, rather, its novelty arises from its ability to accelerate and simplify the research process. MABE is capable of meeting the needs of many users because we do not attempt to provide complete solutions. Instead we implement those core elements which we have identified as having common utility for a majority of Digital Evolution experiments and provide rules for building other more experimentally dependent parts such that they are interchangeable.

MABE emerged from humble beginnings; there was a call to design a general purpose tool to perform research using Markov Brains (Edlund et al. (2011)), (Marstaller et al. (2013)). When we began work on MABE there was a vanilla version of the Markov Brain code. Previous work using Markov Brains would generally modify the vanilla code to add functionality needed for a given experiment. As this work was results driven, little consideration was given to reusability or extendability, and if later research required new features, it was difficult to add them. To make matters worse, if a version of the code was required that implemented features from two modified versions, it was often necessary to rewrite significant amounts of code, and in extreme cases, simply start again from the vanilla code.

The basic requirements for the tool we needed to write were clear; development time for new experiments needed to be minimized, and there needed to be some way to allow for more collaboration between users. We also wanted this tool to be accessible to users with limited programming experience. We felt that these requirements called for a modular design supported by general purpose tools. In discussions of how we could achieve our design goals, we hit upon a fundamental insight that lead to the development of MABE in its current form. We looked into a number of existing digital evolution tools (AEVOL (Batut et al. (2013)), Avida (Ofria and Wilke (2004)), EALIB (Goldsby et al. (2014)), SHARK-ML (Igel et al. (2008)), NetLogo (Tisue and Wilensky (2004)), Neat (Stanley and Miikkulainen (2002)), HyperNeat (DAmbrosio et al. (2014)), SUNA (Vargas and Murata (2016)), etc.), each of which was suited to specific domains and specific problems. We realized that many of these tools implemented similar features such as genomes, brains, lineage tracking, file I/O, data management, user parameters, etc. Consider for instance, Brains (i.e. data processing units); while ANNs, GP trees, Markov Brains, etc. are different in design and function, they all take inputs, execute some process and generate outputs. Our insight was that it should therefore be possible to design a system with interchangeable brain types. We then realized that we could apply the same rationale to genomes and for that matter other features commonly found in different systems. This resulted in an extensively modular and reconfigurable design which addressed the requirements: shareable code, accessibility to users with differing programing experience, and faster development times.

At the time of preparing this manuscript, MABE is actively being used in a number of research projects, and has already generated useful results in the fields of Evolutionary Psychology (Kvam and Hintze (2017)) and Neuroscience (Schossau et al. (2017)). MABE is written in c++11, and runs on Windows, Mac, and Linux. MABE is available as a github repository (github.com/Hintzelab/MABE), which includes the code, examples, documentation, and installation instructions.

MABE Design

Out of the Box

MABE includes tools to help users generate custom versions of the software. The default installation is configured with an example that demonstrates a typical MABE use case. In this example, Organisms will contain a Circular Genome (a simple genome type) which encodes a Markov Brain. A population of organisms (a group) is evaluated using Berry World (a resource harvesting task), which generates a score for each organism. A GA Optimizer (a method to generate a new population using roulette wheel selection) selects highly scored organisms and creates a new population. Data are tracked and saved using the Default Archivist (a module that collects, manages, and saves data). The cycle of evaluation, optimization, and archiving repeats for 20 updates.

In addition, it is easy to gain an understanding of how MABE works because MABE is capable of automatically generating configuration files with embedded parameter documentation. These files allow a user to significantly alter MABE's behavior by changing global parameters, such as population size or run duration, or selecting modules (i.e. which type of genome, brain, etc.) will be used, and managing the settings of those modules. This provides a quick entry point for new users.

Functional Overview

MABE is essentially a framework which defines how different parts (hereafter modules) of an experiment interact. The modules are: genomes, brains, optimizers, worlds and archivists (each is defined in detail in the next section) and an experiment is a collection of modules. Since each experiment is different and requires different features (many of which cannot be anticipated), MABE's modules are defined, not by their functionality, but rather, by the rules which manage their interactions. In other words, MABE is less concerned with the inner workings of the modules themselves, but rather, provides the connective tissue between the various modules. This is essentially the principle of encapsulation from computer science (the approach of keeping implementation details hidden from the rest of the system).

In addition to modules, MABE includes tools which generally exist to address issues which we have identified as common to different digital evolution systems, but which are not required to be modular. The Parameters System (explained in detail below), for example, provides a way for users to interface with configurable attributes and also manages those attributes. This functionality is found in all of the digital evolution systems that we looked at and it was implemented in effectively the same way in each of them.

To illustrate a benefit of using MABE, imagine that one set of users implements a collection of optimizers in order to test how different selection methods affect adaptation, and another set of users develops a world to test the evolution of cognitive abilities in temporally variable environments. Now let us say that you see the results of both these experiments and realize that a combination of modules from these two experiments would be useful for your research. Because both experiments were done in MABE, these modules will be compatible, and you can simply combine these modules without the need to write any additional code. As stated previously, this modularity extends to Genomes, Brains, and Archivists, allowing for arbitrary combinations.

As MABE's user base grows, there will be an ever increasing collection of modules and an ever increasing probability that some combination of existing modules will be capable of generating an experimental system for a given question, either directly or with only small modifications. It is our hope that this will optimize the time needed to progress from hypothesis to data generation, foster collaborations, and therefore make digital evolution more accessible.

Modules

Genomes

In biology, genomes are sequences of four types of nucleotides (A,C,G, and T) in the form of DNA. MABE genomes follow a similar concept, with the fundamental difference that MABE genomes are not limited to four symbols, rather they can be constructed either from discrete or continuous values. Genomes are accessed by other modules with a *GenomeHandler*, which acts like a read/write head similar to a file handle, and provides a standard interface to genomes. Each type of genome defines its own copy and mutation operators. The most common use of genomes in MABE is to provide a blueprint for constructing brains (although they can be used by other modules, e.g. in worlds

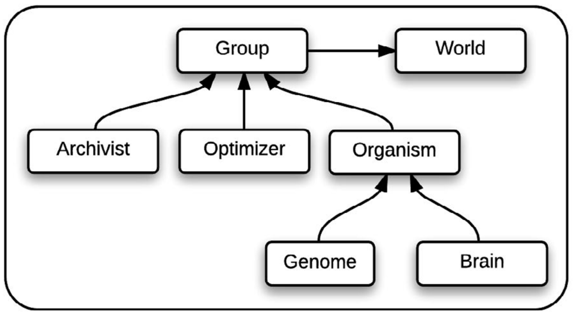

Figure 1: Depiction of core MABE modules showing their functional relationships.

to define bodies, or in optimizers to define sexes, etc.). The repository for MABE currently includes two genome types:
Circular : a simple, fast and efficient circular genome.
Multi : a polyploid genome with multiple chromosomes, which implements biologically inspired crossover and recombination.

Brains

In MABE, Brains are data processors that receive inputs and deliver outputs. What makes each brain different is how they convert inputs to outputs, i.e. their internal algorithms. Classic examples of internal algorithms are Markov Brains and Artificial Neural Networks. Inputs and outputs are kept in lists of continuous values. The internal mechanism that a brain uses to convert inputs to outputs is only limited by the computational power of the computer being used to run MABE. Brains can (and usually will) create internal structures to allow for memory or to encode state.

Brains are either a) constructed from genomes, in which case a construction method translates a genome into a brain, and variation is introduced by mutating the genome and constructing a new brain, or b) by direct encoding, in which case, a construction method that does not rely on a genome, directly creates a brain, usually by employing some random process, and variation by mutation is introduced directly into the brain.

Generally, brains have no information about the types of genomes that built them nor the worlds which evaluate them. While this may seem like a limitation, consider that the fundamental cognitive elements of biological brains almost certainly have no direct mapping to the meaning of what is being processed (sometimes referred to as the grounding problem (Harnad (1990))). As a result of this independence, any brain which is constructed from a genome can be constructed using any type of genome and any brain can be used in any world. (Note: MABE does allow for the explicit binding of a brain to a world. For an example, see Use Case: game theory and brain-world interface)

The brain interface includes functions to set inputs, update the brain's state according to its algorithm, and read outputs. The MABE code repository currently includes the following brain types:
Markov : a brain consisting of *gates* (each gate takes inputs, runs some process, and delivers outputs) which are run in parallel. The outputs from all of the gates are combined (usually by summation). (Edlund et al. (2011), Marstaller et al. (2013))
Wire : a three-dimensional evolvable implementation of Wire World, a cellular automaton (CA) (Dewdney (1990)). (see Fig 2b)
Constant Values : converts a genome into unchanging constant outputs.
GP : a tree based directly encoded genetic programing brain (Koza (1992))
CGP : a simple implementation of a Cartesian GP (Miller and Thomson (2000))
LSTM : an implementation of a Long-Short-Term-Memory artificial neural network (Hochreiter and Schmidhuber (1997)).
Human : prints the inputs of the brain to screen and then queries the user for outputs (useful for debugging and world design)

Optimizers

Optimizers take a population and use the results of the evaluation performed by worlds (see below) to select parents and generate a new population. Depending on the type of optimizer being used, the new generation may be composed of all new organisms or it may include organisms from previous generations. Optimizers can implement asexual or sexual reproduction, and can access organism's genomes (for example, a genome can be used to determine the sex type or sexual preference of an organism). The current repository of MABE includes the following optimizer types:
GA : an asexual optimizer which implements roulette wheel (i.e. fitness-proportionate) selection (Goldberg and Deb (1991)).
Tournament and **Tournament2** : optimizers that use a tournament selection method (best of n). Tournament is asexual, Tournament2 is sexual (for a more detailed review on tournament selection see (Goldberg and Deb (1991)).

Archivists

Archivists determine what data need to be saved and when (while attempting to limit memory usage). Files are generated by Archivists in *csv* format, a format which is supported by many existing tools and is human readable. Archivists generally output two types of files; temporal files which are files containing data recorded over time (such as the data associated with the highest scoring organism in a population, or the average of all organisms in a population), and snapshot files which are files containing data relating to a group of organisms at a particular instant. The current repository of MABE includes the following Archivists:
Default : a simple Archivist which saves basic population

statistics as well as whole population snapshots.
LODwAP (Line Of Decent with Aggressive Pruning) : in addition to the functionality of the Default Archivist, files are saved with information about organisms on the line of descent (Lenski et al. (2003)). This is useful for reconstructing evolutionary histories.
SSwD (Snapshot with Delay) : in addition to the functionality of the Default Archivist, additional limited population snapshot files are periodically saved which contain only those organisms whose lineages survive past some delay period. This is useful for visualizing speciation, extinction, and population diversity.

Organisms

Organisms are containers which hold genomes and brains, as well as other data needed to manage the organism (organism ID, time of birth, time of death, lineage and data tracking details, etc.). Organisms have no information about the worlds in which they are being evaluated, so any variables that relate to a world (such as score, number of inputs/outputs, time, etc.) will not be found in organisms; they must be defined in worlds.

Organisms have DataMaps (see Utilities) which can be used to communicate data between modules (e.g. a score set by a world can later be read by an optimizer).

Groups

Groups, like organisms, are containers which hold a population (a collection of Organisms), an Archivist, and an Optimizer. It is possible to have multiple (potentially interacting) Groups in MABE so Groups are necessary to bind a population to an optimizer and an archivist.

Worlds

Typically, worlds evaluate the ability of organisms to perform some task or generate a solution to some problem.

Worlds interact with brains by a) programming brain inputs based on the world state, b) calling a brain update, c) reading the brain's output, and then d) updating the state of the world. Worlds can also read from genomes. (e.g. a genome could be used to provide a phenotype, to define a body for an organism, or in the case of problems like NK fitness landscapes (Kauffman and Levin (1987)), the world may evaluate the genome directly, and not require a brain).

Like brains, the evaluations performed by worlds are only limited by the computational power of the computer being used to run MABE. Worlds can create structures to encode state and track world specific data. In addition, worlds can interface with outside libraries and tools (e.g. a physics simulator or a robotics emulator (see Use Case: evolving robotics controllers with ROS)).

Solo vs. Group Worlds
Worlds can be written to evaluate one agent at a time, a subset of a population, an entire population, or even multiple populations concurrently. Solo worlds are sufficient to ask questions relating to topics such as foraging strategies, temporal entrainment, character recognition and other vision related tasks, etc. Group worlds can model complex environments where agent's actions change the world, as is the case with frequency dependent resources. Group worlds also allow for the study of systems involving organismal interactions, which can be be used to study questions relating to cooperation, predator prey dynamics, parasite host interactions, etc.

Single Generation vs. Multi-Generation Worlds
In the Out-of-the-Box description above, a single generation world (Berry World) is used to generate a score for each member of a population and then an optimizer (GA Optimizer) uses these scores to create a new generation. Conversely, in a Multi-Generation world, offspring are produced not by an optimizer, but by the world. Typically, in Multi-Generation worlds the optimizer is replaced by some implicit method of reproduction (e.g. in order to reproduce, a world may require organisms to colocate for a specified amount of time). Multi-Generation worlds must make direct calls to the Archivist and must define their own termination cases.

The current repository of MABE includes the following worlds:
Berry : a world that defines a resource harvesting task with various types of food, where the number of types of food, reward for each food type, switch costs, etc. can be configured. Berry World provides solo and group evaluation modes.
IPD (Iterated Prisoners Dilemma) : a world designed to evolve players for two player games like the Prisoners Dilemma (Smith (1982)).
Weed : a group evaluation, multi-generational world which evolves plant like organisms.
Morris Test : tests an organism's ability to navigate to a location based on reference markers. (Morris (1984))

Utilities

Utilities are standalone code elements that are used by MABE which can be loosely divided into two categories. Core Utilities are integrated into the core elements of MABE. Helper Utilities are generally less integrated, and are tools to help with development of modules. (e.g. navigation tools for worlds).

Parameters

The Parameters system provides a simple, standardized method to define parameters which are accessible by all parts of MABE. Parameters are defined with a register function which defines the parameter name, category, type, default value, and a usage message. The parameters system can be called upon to generate configuration files from registered parameters. In other words, MABE will generate its own configuration files that also include documentation. The Parameter System also includes hierarchical namespaces (this allows for unique parameters for different populations or instances of worlds, etc.)

DataMaps

DataMaps are data collectors. Any object in MABE can be assigned a DataMap. Adding to and reading from a DataMap are handled by set and get operators which resolve data types. DataMaps include functions for saving data to files (these functions are used by Archivists when saving data, but can be accessed by developers directly). DataMaps can store data either as single values (set operator) or lists of values (append operator). DataMaps queries can return single values, lists, averages of a list, etc. For example, a world could generate a score which is then set in an organism's DataMap. This score could be used by an optimizer as part of a process for selecting parents, and also by an Archivist which could save this score to a file.

FileManager

The FileManager is the utility which manages file access. In addition to keeping track of which files have been accessed during a session, the file manager provides a single interface for file I/O. The FileManager is also necessary for the correct operation of DataMaps.

Helper Utilities

MABE includes several stand-alone utilities which can be used to build modules. Often though, modules will be written with their own utilities. Helper Utilities only include tools which we predict may be useful to many users. Examples of Helper Utilities include a Random number generator, a vector serialization method, and binary expression trees (which allow for human readable formulas in configuration files).

Standalone Tools

Standalone Tools are not part of the MABE executable; they either help create MABE code, or provide methods to analyze or visualize MABE output. Most of these tools are implemented in Python. Some examples of standalone tools include:
MBuild.py builds a MABE executable (converts the code to a working program) using a configuration file which determines which modules will be included.
MGraph.py generates simple graphs from *csv* files generated by MABE. By default, MGraph will attempt to graph all data in a given file. Command line options can be used to change what data is plotted, how it is plotted, alter visual details, etc.

MABE Design Philosophy

The single phrase that best encapsulates the philosophy of MABE is "Simple is Better." The primary reason for this philosophy is to simplify the generation of new modules. In order for MABE to succeed, users with limited coding experience must be able to design and implement new modules. A less obvious reason for the "Simple is Better" philosophy is that a growing user base may have needs that we failed to anticipate. When useful additions or changes that would require alterations to MABE's core code are suggested, we want to be able to integrate them. This goal is best served by avoiding complexity. As a result, MABE is implemented using the simplest possible coding constructs, except on rare occasions where either a significant improvement to speed or organization could be achieved.

In order to simplify configuration and minimize errors, MABE automatically initializes modules based on context. Modules provide information about their own requirements in a standardized format which can be accessed by other parts of MABE. For example, if an organism with a directly encoded brain is being used in a world which does not require a genome, MABE will identify that this organism does not need a genome and will not include one (saving the overhead related to creating and managing a genome). The automatic initialization functionality is particularly useful when it comes to the world-brain interface. The number of inputs and outputs a brain has are determined by the world within which it is being used, since it is the world that defines the meaning of the inputs (possibly relating to vision, numerical input values, or communication from other organisms, etc.) and outputs (possibly relating to organismal movement, reproduction, an answer to a mathematical problem posed by the world, etc.). This is again the Binding Problem discussed in Brains. In addition, Archivists adjust their output based on genomes, brains, and worlds being used.

Beyond simplifying use and enhancing extendability, MABE's modularity has another interesting effect, which may end up being its greatest contribution. As MABE's collection of modules expands, the number of experiments which can be performed using existing or trivially modified modules will continue to increase. Now consider the issue of substrate dependence, i.e. determining if an outcome describes an actual phenomenon, or is simply a result of the particular system being employed. Since it is trivial to swap modules in MABE, it is also trivial to rerun experiments with different modules. It is possible therefore to generate results from a single world using a collection of different types of genomes and/or brains, or to evaluate a brain in different worlds, etc. (see Use Case: substrate independence related to brains). Moreover, since all of the results are generated by MABE, i.e. using the same code, these results are directly comparable.

Developing for MABE

Since we cannot predict future MABE experiments, we have attempted to simplify the process of updating modules. Modules are isolated from each other and can only communicate though interfaces, so developers only need to understand their module and how it interfaces with other modules, but need not understand the entire code base. Once a new module has been developed it is guaranteed to be compatible with all existing modules. This does not mean that modularity comes without costs; developers may have to tailor their programming styles to fit MABE. However, we argue that the benefits of modularity outweigh those costs, even more so when one considers the framework and utilities which MABE provides.

MABE is a software package; it is not a library. A library provides a set of tools which must be assembled in the correct fashion in order to function properly. Since MABE works "out of the box", a developer starts with a working piece of code and thus can choose to either design from scratch (like you would with a library) or employ an incremental development style where a new module is created by altering an existing module. Incremental development is more beginner-friendly because if a developer makes a mistake or introduces an error they can back up to the last working version and locate the problem. In contrast, when building a project using a library from scratch, one may need to assemble a considerable code structure before they can start testing. To support either development style, MABE is designed to fail gracefully. If an illegal operation is executed, MABE stops running and attempts to provide a meaningful error message.

A note to experienced coders, or 'Why should experienced coders use MABE?' Here are the four reasons you might want to use MABE. **a)** MABE saves you time. In many cases, the necessary modules will already exist, or will need only minor modifications. This greatly reduces the time frame between experiment design and a working implementation. **b)** With MABE you do not need to develop and support code to address problems such as, tracking populations, collecting (and serializing and saving) data, or configuring and managing parameters. You can get right to work on your new brain or world or whichever module is of interest to you. **c)** The MABE framework facilitates collaboration. As the user base grows, there will be more and more modules to build from, and *your modules will be accessible to many potential collaborators!* **d)** The principle of substrate independence discussed in the design philosophy, can help to determine if some effect is the result of an actual phenomenon, or is simply the result of implementation details (providing a form of results validation). You are welcome, of course, to use MABE as a library; pick and choose the elements that you like and write the rest yourself. But, we believe that the real power of MABE comes from its modularity and the resulting collaborations.

Use Cases

Substrate independence related to brains

In Berry World organisms can navigate on a grid and are tasked with finding various resources that have associated rewards and costs. Here we consider a Berry World configured to study the evolution of patch harvesting strategies, where organisms are presented a patch of resources and must attempt to maximize the resources gathered in a limited amount of time. In figures 2a and 2b we show the paths generated by two evolved MABE organisms with different types of brains - a Markov Brain with Neuron gates (a collection of

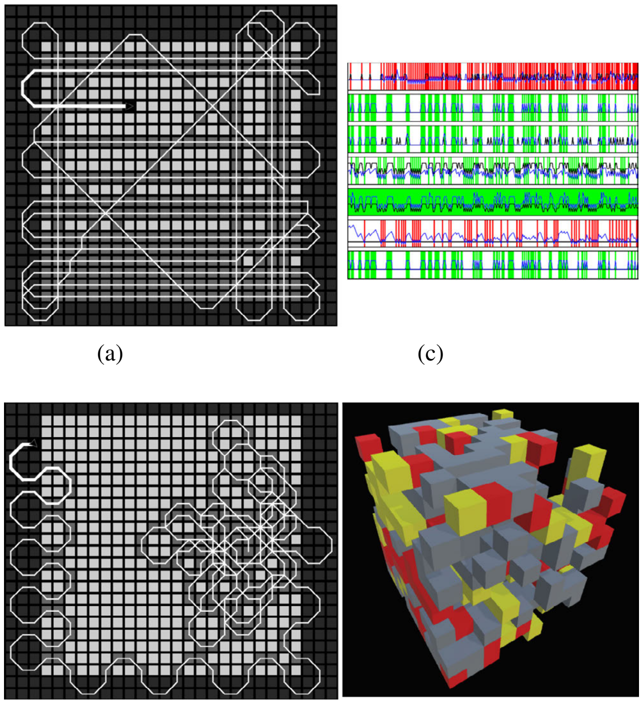

Figure 2: One of the strengths of MABE results from the ability to quickly switch modules. Shown here are results from a patch harvesting task (a and b) where two different types of brains are tested in the same world. In this world, an organism's score is based on the number of food resources which can be harvested in a limited amount of time. Light grey squares represent food resources and the organism's path is shown in white. (a) shows a harvesting strategy generated by a Markov Brain using Neuron Gates - processing units modeled on simple biological neurons. Neuron Gates accumulate input charge from one or more inputs, until the charge reaches a threshold and the gate fires, delivering an output. Neuron gates can be configured so that the brain can modulate the threshold and output value of a neuron gate while the brain is active, allowing for regulation, feedback and, we believe, lifetime learning. (c) shows a section of neuron gate activity for the seven neurons that made up this organism's brain. Here green indicates that an activating neuron is firing, while red indicates that a repressive neuron is firing. The blue line indicates current charge and the black line the current threshold. (b) show a harvesting strategy generated by a Wire Brain. Wire Brains approximate electrical current flowing though wires contained in a cubic virtual space. The wire in a Wire Brain becomes charged if a neighboring wire is charged, but then must undergo a reset period (decay) before returning to the uncharged wire state. The reset period results in directional movement of charge. Inputs are delivered to a Wire Brains work by charging input wires on one side of the brain, allowing charges to radiate through the brain for some number of internal updates and observing 'output' wires on the opposite side of the brain to determine the brain's output. (d) shows a visualization of a wire brain in mid-update, where grey is uncharged wire, red is charged wire and yellow is wire in decay.

modulated threshold units) and a Wire Brain (a type of cellular automaton). Figures 2c and 2d illustrate how the internal processes of these two brains are significantly different.

WeedWorld, a multi-generational world

WeedWorld simulates the evolution of simple sexually reproducing plant like organisms. These organisms have roots, a stalk and leaves. The size of the roots determines nutrient and water uptake. Nutrients are used to grow roots, and nutrients and water are used to grow leaves and the stalk. Leaves allow for more effective use of nutrients and stalk height affects pollination success and seed dispersal distance.

Organisms in WeedWorld do not reproduce by means of an Optimizer, rather, mating and reproduction is managed by WeedWorld directly. That is, mates are selected, offspring are produced, and death is addressed in the world as these events occur (i.e. lifetimes are variable and can result in overlapping generations). Consequently this world does not need to generate scores for organisms, since fitness is implicit. Figure 3 shows the dynamics of a WeedWorld experiment.

Game Theory and Brain-World interface

Iterated Prisoners Dilemma is a well studied problem in Game Theory (Smith (1982)). We are working with researchers who are interested in asking questions related to how different brains and genomes affect the evolution of strategies. Obviously due to its modular design MABE is a good fit. However, this research also requires brains with fixed strategies, which in turn requires brains that have information about the worlds in which they are running. We have previously stated that, in general, brains do not have information about the worlds in which they are running, but if the meanings of the inputs and outputs that a brain will receive and deliver are known, a brain can be designed to make use of this knowledge.

In Iterated Prisoners Dilemma (IPD) World, two inputs are provided (input 0: is this the first move?, input 1: other players last move - either cooperate or defect) and one output is expected (output 0: the brain's move - either cooperate or defect). IPD Brains are brains which are specifically implemented to be used in conjunction with IPD World. IPD Brains contain hand coded strategies and an internal switch (which can be genetically determined) that chooses between them.

Note that while IPD Brains will function with any world, they will almost certainly not be able to generate meaningful outputs when that world is not IPD World . This sort of brain-world binding is counter to the concept of interchangeable modules, and so should be avoided except in cases like this because it breaks the ability to perform substrate independent tests.

Evolving Robotics Controllers with ROS

One of the complexities encountered in robotics is interfacing between many distributed components (e.g. sensors, actuators, and controllers), which often have dissimilar interfaces, data formats, and rates of communication. The Robot Operating System (ROS) is an open-source framework used for robot control that solves these problems by providing standardized synchronous and asynchronous communications protocols (via a topic-based publish/subscribe and service-based request/reply communication model) (Quigley et al. (2009)).

A MABE world was developed to evolve ROS compatible brains. This world was designed to function in two modes: 1) a simple spatial environment that provided a fast ROS-like interface to quickly evolve brains and 2) a mode that interfaced MABE directly with ROS such that ROS inputs could be fed directly into brains and brain outputs could be redirected back into ROS, allowing for the control of either simulated or actual robots.

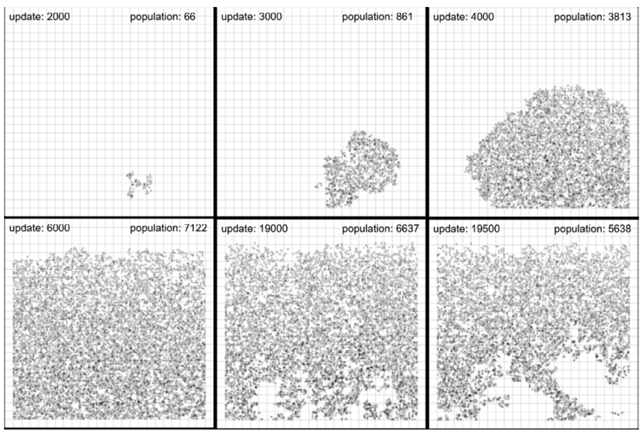

Figure 3: This figure shows an evolving population of weeds where nutrients are distributed on a vertical gradient increasing from top to bottom. The individual frames are labeled with the time of the frame (update) and the size of the population. Frames 2,000 through 6,000 show steady growth as the population fills the available space. The situation in frame 6,000 holds steady until shortly before frame 19,000, where it is apparently disturbed by the appearance of a new strain, which is fast moving, short lived and destructive.

Outlook

We are convinced that the fundamental concepts outlined in this paper (interchangeable parts, reconfigurability, and substrate independence), have the potential to significantly accelerate advancements in the field of digital evolution research. To the best of our knowledge, MABE is the first tool to implement these concepts in a comprehensive fashion. Certainly, the idea of developing a single platform that can be used to investigate significantly different research questions is ambitious, but we feel that it is not only possible, but in fact critical to moving past the current state of affairs, which is plagued by inefficiencies resulting from repeated work, and the difficulty in comparing results generated by different systems.

The functionality and modules presented here only represent the current state of MABE. But MABE is an organic project, which has been designed to be able to grow and adapt to an ever expanding user base. As the community of users grows and matures we are committed to providing support with improved features and documentation. We look forward to many collaborations in the years to come with researchers from different disciplines and backgrounds.

Contributions

CB and AH conceived of the project which resulted in MABE. CB designed MABE's framework and implemented the code base. AH supervised the project. CB and NCG prepared the manuscript.

Acknowledgements

This work is supported by the NSF BEACON Center for the Study of Evolution in Action Cooperative Agreement No. DBI-0939454., and start-up funds from Michigan State University for the Hintze lab. We would like to thank Doug Kirkpatrick, Alex Lalejini, and Jory Schossau for their input in preparing the manuscript.

References

Adami, C., Schossau, J., and Hintze, A. (2012). Evolution and stability of altruist strategies in microbial games. *Physical Review E*, 85(1):011914.

Batut, B., Parsons, D. P., Fischer, S., Beslon, G., and Knibbe, C. (2013). In silico experimental evolution: a tool to test evolutionary scenarios. *BMC bioinformatics*, 14(15):S11.

Chandler, C. H., Ofria, C., and Dworkin, I. (2013). Runaway sexual selection leads to good genes. *Evolution*, 67(1):110–119.

Cully, A., Clune, J., Tarapore, D., and Mouret, J.-B. (2014). Robots that can adapt like animals. *arXiv preprint arXiv:1407.3501*.

Dewdney, A. (1990). Column" computer recreations": Wireworld. *Scientific American, January*.

DAmbrosio, D. B., Gauci, J., and Stanley, K. O. (2014). Hyperneat: The first five years. In *Growing adaptive machines*, pages 159–185. Springer.

Edlund, J. A., Chaumont, N., Hintze, A., Koch, C., Tononi, G., and Adami, C. (2011). Integrated information increases with fitness in the evolution of animats. *PLoS Computational Biology*, 7(10):e1002236.

Frénoy, A., Taddei, F., and Misevic, D. (2013). Genetic architecture promotes the evolution and maintenance of cooperation. *PLoS Comput Biol*, 9(11):e1003339.

Goldberg, D. E. and Deb, K. (1991). A comparative analysis of selection schemes used in genetic algorithms. *Foundations of genetic algorithms*, 1:69–93.

Goldsby, H. J., Knoester, D. B., Ofria, C., and Kerr, B. (2014). The evolutionary origin of somatic cells under the dirty work hypothesis. *PLoS Biol*, 12(5):e1001858.

Harnad, S. (1990). The symbol grounding problem. *Physica D: Nonlinear Phenomena*, 42(1-3):335–346.

Hochreiter, S. and Schmidhuber, J. (1997). Long short-term memory. *Neural computation*, 9(8):1735–1780.

Igel, C., Heidrich-Meisner, V., and Glasmachers, T. (2008). Shark. *Journal of machine learning research*, 9(Jun):993–996.

Kauffman, S. and Levin, S. (1987). Towards a general theory of adaptive walks on rugged landscapes. *Journal of theoretical Biology*, 128(1):11–45.

Koza, J. R. (1992). *Genetic programming: on the programming of computers by means of natural selection*, volume 1. MIT press.

Kvam, P. D. and Hintze, A. (2017). Rewards, risks, and reaching the right strategy: Evolutionary paths from heuristics to optimal decisions. *Evolutionary Behavioral Sciences, invited submission for the Special Issue on Studying Evolved Cognitive Mechanisms.*

Lenski, R. E., Ofria, C., Pennock, R. T., and Adami, C. (2003). The evolutionary origin of complex features. *Nature*, 423(6936):139–144.

Lohn, J., Hornby, G., and Linden, D. (2005). An evolved antenna for deployment on nasas space technology 5 mission. *Genetic Programming Theory and Practice II*, pages 301–315.

Marstaller, L., Hintze, A., and Adami, C. (2013). The evolution of representation in simple cognitive networks. *Neural computation*, 25(8):2079–2107.

Miller, J. F. and Thomson, P. (2000). Cartesian genetic programming. In *European Conference on Genetic Programming*, pages 121–132. Springer.

Morris, R. (1984). Developments of a water-maze procedure for studying spatial learning in the rat. *Journal of neuroscience methods*, 11(1):47–60.

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

Olson, R. S., Hintze, A., Dyer, F. C., Knoester, D. B., and Adami, C. (2013). Predator confusion is sufficient to evolve swarming behaviour. *Journal of The Royal Society Interface*, 10(85):20130305.

Quigley, M., Conley, K., Gerkey, B., Faust, J., Foote, T., Leibs, J., Wheeler, R., and Ng, A. Y. (2009). Ros: an open-source robot operating system. 3(2):5.

Schossau, J., Albantakis, L., and Hintze, A. (2017). The role of conditional independence in the evolution of intelligent systems. In *Proceedings of the 2017 on Genetic and Evolutionary Computation Conference Companion*. ACM.

Smith, J. M. (1982). *Evolution and the Theory of Games*. Cambridge university press.

Stanley, K. O. and Miikkulainen, R. (2002). Evolving neural networks through augmenting topologies. *Evolutionary computation*, 10(2):99–127.

Tisue, S. and Wilensky, U. (2004). Netlogo: A simple environment for modeling complexity. In *International conference on complex systems*, volume 21, pages 16–21. Boston, MA.

Vargas, D. V. and Murata, J. (2016). Spectrum-diverse neuroevolution with unified neural models. *IEEE Transactions on Neural Networks and Learning Systems*.