A Framework for Evolutionary Computation in Agent-Based Systems

Robert E. Smith and Nick Taylor
The Intelligent Computing Systems Centre
The University of The West of England
Bristol, UK
email: rsmith@ics.uwe.ac.uk, nick.taylor@ics.uwe.ac.uk

Abstract

For agent-based systems to reach their full potential, an important capability for individual agents is adaptation. An adaptive technique that is particularly well suited to the agent-based paradigm is provided by evolutionary computation (EC). EC systems have been shown to develop complex groups of coevolved structures. In fact, Holland's original vision of artificial adaptive systems was more like an agent-based system than the typical centralized genetic algorithms (GAs) used today. Moreover, the EC techniques employed today are naturally distributable to an agent-based system. However, no standardized agent-based framework that includes EC capabilities is currently available. This paper introduces such a framework, based on Java, and IBM's Aglets. This framework provides a foundation for giving general agents EC capabilities. These capabilities are demonstrated in a basic optimization application, illustrating that the decentralized agents have emergent behavior which is equivalent to that of a centralized GA. The paper will also discuss the range of applications made possible by a standardized EC capability for agents.

EC's Potential Impact On Agent Based Systems

The key qualities that agent-based components and systems exhibit are: autonomy, reactivity, proactivity, and social behavior. Moreover, agents have the possibility of mobility in complex network environments, putting software functions near the computational resources they require. Agents can also explicitly exploit the availability of distributed, parallel computation facilities (Franklin & Graesser, 1997; Wooldridge & Jennings, 1996)

However, these qualities ultimately depend on the potential for agent adaptation. For instance, if an agent is to operate with true autonomy in a complex environment, it may have to react to a spectrum of circumstances that cannot be foreseen by the agent's designer. Autonomous agents may need to explore alternative reactive and proactive strategies, evaluate their performance online, and formulate new, innovative strategies without user intervention. Moreover, for systems of agents to behave in this manner, social interactions between agents may also need to adapt and emerge as conditions change.

Areas where agents could benefit from adaptation are addressed by active research in machine learning (e.g., classification of unforeseen inputs, strategy acquisition through reinforcement learning, etc.). However, many machine learning techniques are focused on centralized processing of databases to formulate models or strategies. In contrast, EC techniques are inherently based on a distributed paradigm (natural evolution), making them particularly well suited for adaptation in agents.

Motivation For Melding Agents And EC

As is illustrated in past genetics-based machine learning (GBML) systems (Smith & Dike, 1995), complex, innovative, multi-component adaptive systems can emerge from EC processes. Moreover, these EC processes implicitly exploit parallelism, while remaining trivial to explicitly parallelize (Holland, 1975). Therefore, EC methods are one of the most natural machine learning techniques to transfer general-purpose adaptive capabilities to agent-based systems.

Although there is a large body of extant work on the application of parallel and distributed EC algorithms (Kapsalis, Smith & Rayward-Smith, 1994), these studies differ substantially from the agent-based work introduced here. Specifically, past parallel GAs:

- usually consist of a number of centralized GAs running on separate computational nodes
- are restricted to optimization, rather than coevolutionary (GBML) applications, and
- are often designed for particular parallel computer configurations, and
- are not standardized to fit in with other distributed software systems.

There is no currently available, generalized, agent-based EC system. The work introduced here seeks to provide and test such a system.

Design of the Framework

To design an agent-based EC system, one must turn the typical GA software design on its head. In common GA software, a centralized GA program stores the GA individuals as data structures, and imposes the genetic operations that generate successive populations (Figure 1).

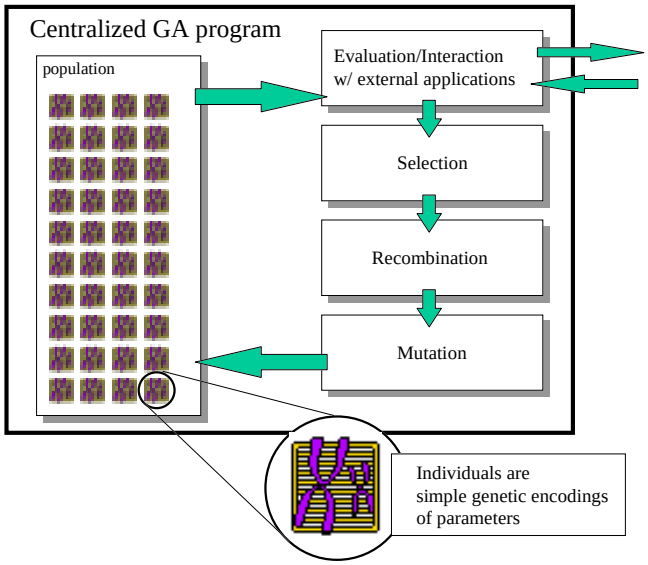

Figure 1. Structure of a typical, centralized GA.

In natural genetics, the equivalent of the centralized GA does not exist. Instead, the evolutionary effects emerge from individual agents. This makes designing an agent-based EC system a matter straight forward analogy (see Figure 2).

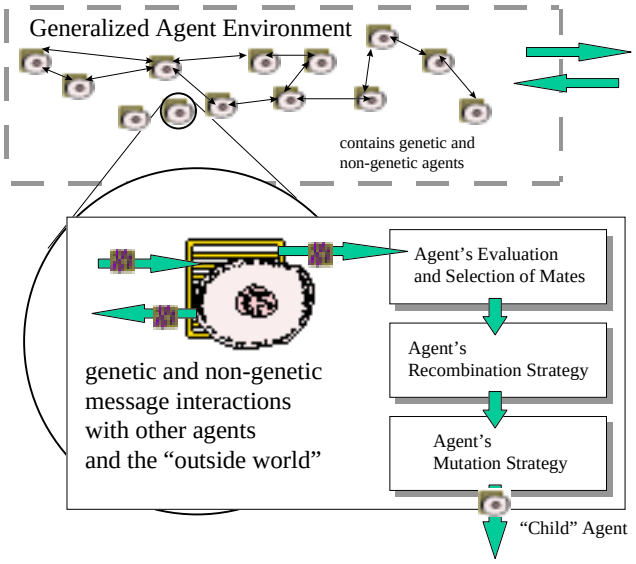

Figure 2. Structure of the EC-enabled agent system.

However, in designing such a system, one must make sure to allow for extensibility and broad utility. The framework introduced in this paper is a tool that can be used within broader agent-based, centralized, and hybrid systems. By including EC agents in broader frameworks, the advantages of genetic adaptation can be coupled with controlled agent behavior.

Structure of the Framework

The framework introduced here is called *Egglets*. The `Egglet` framework is built in Java, for its machine independence, and on IBM's Aglets framework, for basic agent capabilities (e.g., agent messaging, agent storage and recall, etc.). The Aglet framework also allows for agent mobility. Using these system's standards will allow for consistency with other software systems built on these standards.

The first element of the framework is a package of Java classes and interfaces called *ec.basics*. The intention of this package is to provide basic EC capabilities for general Java objects. The first keystone of the package is

```
interface ec.basics.Gene
(extends Cloneable, Serializable)
```

A class implementing this interface is required to implement a *Mutate(Object parameters)* method, which takes a set of parameters as its argument. When an object's *Mutate* method is invoked, the single "gene" represented by the object is altered. For instance,

```
class ec.basics.BooleanGene
(implements ec.basics.Gene)
```

represents the binary gene often used in GAs. As expected, this object's *Mutate* method flips the boolean value of the object. However, the Gene interface can be used in other ways as well, for instance,

```
class ec.basics.FloatGene
(implements ec.basics.Gene)
```

has a *Mutate* method that adds a Gaussian random variable to the object's value (so-called "creep" mutation).

The second keystone of ec.basics is

```
interface ec.basics.Genotype
(extends Cloneable, Serializable)
```

Classes implementing this interface must provide methods *ToSperm* and *ToEgg* for producing new instances of classes that implement the interfaces

```
interface ec.basics.Gamete
(extends Cloneable, Serializable)
  interface ec.basics.Sperm
  (extends ec.basics.Gamete)
  interface ec.basics.Egg
  (extends ec.basics.Gamete)
```

The `Gamete` interface requires methods *getSegment* and *setSegment*, which return segments of their objects, suitable for recombination methods. The `Sperm` interface is a wrapper for `Gamete`. The `Egg` class requires a

ToGenotype method that takes an object implementing the `Sperm` interface (and parameters) as arguments, and returns an object implementing the `Genotype` interface.

Each of these interfaces are intended for extension of existing Java data structures. For instance

```
class java.util.Vector is extended by
  class ec.basics.VectorGenotype
   (implements ec.basics.Genotype)
  class ec.basics.VectorGamete
  (implements ec.basics.Gamete)
  is extended by
    class ec.basics.VectorEgg
    (implements ec.basics.Egg)
    class ec.basics.VectorSperm
    (implements ec.basics.Sperm)
```

These classes provide the basic "string" data structures used in typical GAs. However, the `ec.basics` interfaces could be applied to tree structures (as in genetic programming (Koza, 1992)), or `Arrays`, or other data objects. The motivations for some of the divisions of these interfaces (`Gene, Genotype, Egg`, etc.) are more than simple "wishful mneumonics". They provide for natural organization and functionality in the agent-based system, as is discussed below.

The second element of the framework is the *`ec.gaglet`* package. The intention of this package is to provide interfaces and classes that give Aglets EC capabilities, using the structures outlined in ec.basics.

The base EC capabilities are divided into two interfaces

```
interface ec.gaglet.GeneticInitAgent
  interface ec.gaglet.BreederAgent
  (extends ec.gaglet.GeneticInitAgent)
```

Classes that implement that `GeneticInitAgent` interface must have several methods. Key amongst these are the methods *`GeneticCreation(GeneticInitAgent MyParent)`* and *`GeneticCreation(GAgletPhylum MyPhylum)`*. The first method allows the object to "create" its genetic structure based on a similar parent. The second allows the object "create" its genetic structure based on a "phylum" object, which will be further discussed below. The `BreederAgent` interface extends the `GeneticInitAgent` interface, to provide methods that find mates and create children agents. These functions are divided across two interfaces to allow for hybrid agent systems, where some agents may want the have genetic initialization capabilities, without the overhead of mate-seeking and child-creating behaviors.

These interfaces are meant to extend Aglets, for instance

```
class aglet.Aglet
  class ec.gaglet.GAglet
  (implements ec.gaglet.GeneticInitAgent)
    class   ec.gaglet.Egglet
    (implements ec.gaglet.BreederAgent)
```

An `Egglet` is a fully-functional genetic agent that implements mate-seeking and child-creating policies. To implement such policies, `Egglet`s needs a standard class for communicating information needed to make mating decisions. This functionality is provided in the class

```
class ec.gaglet.Plumage
```

This class includes a variety of methods, but chief amongst them is the *`getSperm`* method, which tells an agent desiring mating with the agent represented by a `Plumage` object how to obtain that agent's `Sperm` object. Another key function is *`getSelfDeclaredFitness`*, which gets the fitness of the agent, as declared by that agent. The agent receiving a `Plumage` object has a choice as to whether to trust this information or not.

The final class necessary for EC functionality is the abstract class

```
class ec.gaglet.GAgletPhylum
```

This is used to initialize first generation agents, thus avoiding the "chicken and egg" problem. The `GAgletPhylum` provides the agent with its primary mating policy intentions. Its key method is *`comparePlumages(Plumage p1, Plumage p2)`*, which tells the agent how to judge `Plumage` objects from other agents in making mating decisions. `GAgletPhylum` also provides a *`FormPlumage`* method, which constructs a `Plumage` object for the agent, and (amongst other things) sets the `Plumage`'s *`selfDeclaredFitness`*. To implement agents with a particular set of genetic intentions, one initializes the agents through *`GeneticCreation(GAgletPhylum MyPhylum)`*, where `MyPhylum` is a subclass of `GAgletPhylum`. For instance

```
class ec.gaglet.GAgletPhylum
is extended by
  class ec.gaglet.OneMaxPhylum
```

The *`comparePlumages`* method of `OneMaxPhylum` compares the self-declared fitness values of `Plumage` objects, favoring high values. `OneMaxPhylum` specifies

that an agent have a `VectorGenotype` of length 32, associated `VectorSperm` and `VectorEgg` objects, and `BooleanGene`s. The *`FormPlumage`* method simply sets the *`selfDeclaredFitness`* in the agent's `Plumage` object to the number of true values in the agent's `Genotype`.

Preliminary Results

To obtain emergent GA-like effects from the asynchronous behavior of agents, one must design their mating strategy. As an initial trial, the following strategy was employed:

- If an agent has less than *`CourterThreshold`*=5 `Plumage` objects from other agents (`Courter`s), it sends out a `Plumage` object of its own to another, randomly selected agent.
- If the agent receives a `Plumage` object, it replies with a `Plumage` object of its own.
- `Plumage` objects (both solicited and unsolicited) are asynchronously added to a list of `Courters`, sorted based on the `Phylum`'s *`comparePlumages`* method.
- If an agent has `CourterThreshold` (or more) `Courters`, it gets the `Sperm` object of the best `Courter`, creates a child with this `Sperm`, and dies.

Note that this is similar to tournament selection in a typical GA, but not identical in its effects. In a variant of this strategy, the agent will only add a `Courter` if its `Plumage` as good as or better than its own. This is similar to elitist selection in typical GAs, but, once again, no identical. In both cases, the selection effects are emergent from single individuals. Results from a population of 20 agents from `OneMaxPhylum` are shown in Figure 3. Crossover and mutation rates (defined in the `VectorEgg` object) are 1.0 and 0.01, respectively.

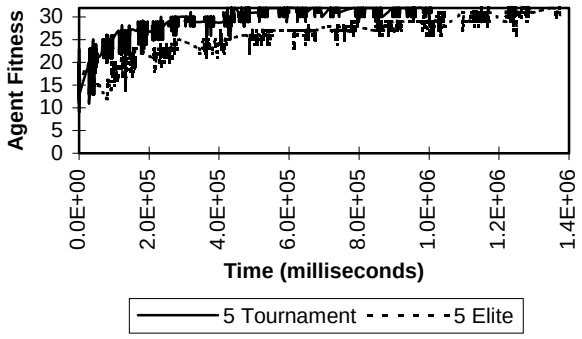

Figure 3: 32-bit `OneMaxPhylum` results.

Note that the notion of "generation number" does not exist in this system, so results are plotted versus time. Time values here are for unoptimized code, running on a Pentium 120MHz with 16Mbytes of memory. Both graphs show "GA-like" emergent behavior, as expected. Surprisingly, in real-time the non-elitist strategy has faster convergence. Further investigation of agent mating strategies is certainly necessary. Moreover, the optimization task addressed here is certainly trivial. However, the intention of these results is simply to illustrate that emergent GA-like effects can be obtained in the standard, agent-based framework presented here.

Final Comments

The agent based EC framework presented here suggests several areas for future investigation. These include

· collaborative EC-adaptive work (search, problem solving, etc.) between distributed users and agents (without a priori knowledge of the collaborations), and

· a variety of EC-scientific experiments, including emergent speciation, self-adaptive GA operators, etc.

The Egglet framework design also allows for diverse agents that can use a variety of genetic representations and operators, while interacting in the same distributed computational environment.

Finally, the framework presented establishes a way of using EC techniques within agents that are based on emerging system standards. This should allow interaction with standard architectures (e.g., CORBA) which are gaining industrial acceptance. This will hopefully increase appropriate and productive use of EC techniques in real world systems.

References

Aglets Workbench. Http://www.trl.ibm.co.jp/aglets/

Franklin and Graesser (1997). Is it an agent, or just a program? : A taxonomy for autonomous agents. In Proceedings of the Third International Workshop on Agent Theories, Architectures, and Languages. Springer-Verlag. pp 21-35.

Goldberg, D. E. (1989). Genetic algorithms in search, optimization, and machine learning. Addison-Wesley. Reading, MA

Holland, J. H. (1975). Adaptation in natural and artificial systems. The University of Michigan Press. Ann Arbor, MI.

Kapsalis, A., Smith, G. D. and Rayward-Smith, V. J. (1994). A unified paradigm for parallel genetic algorithms. In T. Fogarty (ed.) Evolutionary Computing AISB Workshop. Springer-Verlag. 131-149.

Koza, J. R. (1992). Genetic Programming: On The Programming of Computers By Means of Natural Selection. MIT Press.

Smith R. E. and Dike, B. A. (1995). Learning novel fighter combat maneuver rules via genetic algorithms. International Journal of Expert Systems. 8 (3). 247-276.

Wooldridge and Jennings (1996). Software agents. IEE Review. January 1996. 17-20.