EVOLUTIONARY PLATFORM FOR AGENT LEARNING

ANNA L. BUCZAK, DAVID G. COOPER, AND MARTIN O. HOFMANN
Lockheed Martin Advanced Technology Laboratories
Cherry Hill, New Jersey

ABSTRACT
This paper describes a novel methodology for agent learning. This method allows agents to learn how to better operate given the external conditions, in order not to repeat their previous mistakes. Evolutionary Platform for Agent Learning (EPAL) allows agents to change their behavior by changing the parameters of the tasks they are executing and by adding new programming constructs and tasks to their workflow. The method was especially constructed for EMAA agents developed in Lockheed Martin Advanced Technology Laboratories and proven in many military and DARPA programs. EPAL presents a general approach to agent learning, based on an extension to Genetic Programming that we developed. Our approach is suitable to learning by any agent that can be represented as a workflow that can be further decomposed into building blocks, such as operators, tasks, and parameters.

1. INTRODUCTION

Learning plays a fundamental role in most human activities. Humans learn from their experiences, both successes and mistakes. Babies learn to walk and talk, adults learn foreign languages, new skills and concepts. Learning seems to be the fundamental property that allows humans to adapt to changes in the environment and to be successful. The skill of learning is of dynamic nature in humans who continuously acquire and modify representations of the world. On the other hand, artificial systems such as software agents are usually static. For software agents to work properly, humans need to envision all contingencies and code them at the agent development time. Most agents react the same to the similar external stimuli, even when the previous reaction was a failure. We want the agent to learn different behavior in response to those stimuli.

If software agents could "just" learn from their successes and mistakes, they would be more rapidly deployable. However developing a general learning method for software agents is a difficult proposition. Several attempts have been made in the past. Most approaches use reinforcement learning. Stone [1998] proposes a layered learning approach where hierarchical task decomposition is performed that allows the learning to proceed at each level of the hierarchy. Learning performed at a lower level directly affects the learning at the higher level. A new reinforcement learning based method is introduced and applied to robotic-soccer applications.

ABLE agents from IBM [Bigus, 2002] have a library of components such as learning beans that can be used to build an agent. The learning beans implement different learning algorithms (e.g. temporal difference learning, neural networks) and a human can compose an agent that will use any of the learning beans. However this form of agent learning is limited to what a given learning algorithm does and how the human used it when configuring the agent. Conceptually in the ABLE framework there is a place for

agent's learned behavior, however the only method that exists for continuous learning is the reinforcement-learning algorithm.

A so-called "conscious learning" mechanism for software agents was proposed in [Ramamurthy, 1998]. Their agent CMattie takes a role of departmental seminar organizer. She understands past events using her built-in domain knowledge such as a description of the features of a seminar. Should a new situation occur (e.g. dissertation defense) she first treats it as a seminar concept but through e-mail interaction with the organizer learns that a dissertation is a new concept. The "conversation" between the organizer and CMattie is stored in case-based memory. CMattie performs analogy matching and learns through case-based reasoning. The new concepts learned are added to her domain knowledge and the next time she encounters them she recognizes the past experience. CMattie must also learn new behaviors when faced with a dissertation defense and unfortunately this difficult subject was not addressed in the paper.

2. AGENTS AND EXTENDABLE MOBILE AGENT ARCHITECTURE

Lockheed Martin Advanced Technology Laboratories (ATL) developed the Extendable Mobile Agent Architecture (EMAA) over the last seven years. EMAA is used in about two dozen projects covering a full range of intelligent systems, including information management for time-sensitive strike [Hofmann, 2001], situation awareness for small military units, and executing user requests entered via spoken language dialogue. Starting with the Domain Adaptive Information System [Hofmann, 1998], experimentation with prototypes in military exercises has guided our research and development towards the adaptable EMAA architecture. EMAA was used in the US Navy Fleet Battle Experiment (FBE) series as a human aiding tool. From these experiments, an Interoperable Intelligent Agent Toolkit (I2AT) was created under DARPA Control of Agent Based Systems (CoABS) funding to allow people to construct agent systems without coding the agents from scratch, but by generating a workflow of tasks and parameters in a graphical fashion. The toolkit was used to support the Expeditionary Sensor Grid Enabling Experiments where it was used for the first time by non-Java developers to create agents.

EMAA agents are designed with a workflow model for agent construction. An EMAA agent's workflow (Fig. 1) is a list of tasks, linked by execution paths that can be conditional or unconditional. Each task can take a certain number of task dependant parameters. The work-flow architecture enables a new approach to agent development called agent composition, characterized by choosing, configuring, and assembling elementary agent tasks into workflows. Typical tasks include queries to relational databases, retrieval of content from Web pages, reading and sending e-mail, etc. When a user composes an agent workflow in I2AT, the tool produces an XML mark-up of the workflow that specifies agent tasks, their interconnections, and their parameters. Another component of I2AT creates on-the-fly agent instances ready to be executed. EMAA is an environment conducive to learning applications. In the past

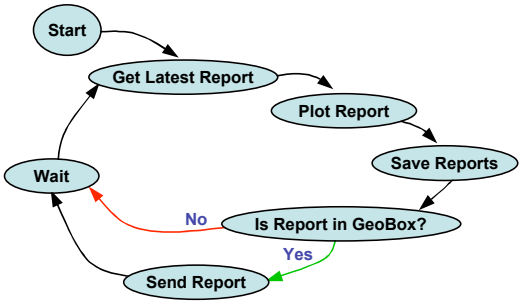

Figure 1. Example of an EMAA Agent Workflow.

years key features have been added to EMAA that constitute the support on which we are basing agent learning. The goal of EMAA's new learning capability is allowing agents to adapt to the environment in which they operate.

3. AGENT LEARNING APPROACH BASED ON GENETIC PROGRAMMING

Our novel approach to agent learning is graphically described in Fig. 2. The first step is to represent agent workflows in a Genetic Programming (GP) tree form. This new representation is described in detail in Section 4. GP individuals are translated from a tree format into an Activity Graph, using their jar file descriptions from I2AT toolkit. The Activity Graph is then used to instantiate a composable agent, which is injected into the simulated system. Fitness is collected while the agent is run in the simulated environment.

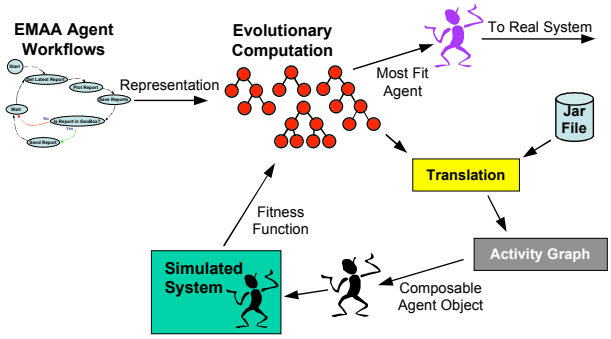

Figure 2. Evolutionary Platform for Agent Learning.

GP operators of reproduction, crossover, and mutation work on agents' genetic material to generate new agents that have learned to overcome certain problems. Once the agents have learned, the most fit are injected into the real system and take the place of their parents who did not know how to overcome the problems. The method proposed is a general method that can generate completely new agent workflows, as well as related workflows but with new parameters. As long as the tasks to add to an agent's workflow are on a task list and related jar files exist, these tasks can be added to an agent, augmenting its workflow and changing how the agent behaves.

4. GENETIC PROGRAMMING EXTENSIONS FOR AGENT LEARNING

Koza [1993] invented GP as a means of program synthesis by genetically breeding a population of computer programs using the principles of Darwinian evolution. In GP each individual is represented as a tree constructed from functions and terminal symbols and often the trees represent programs in LISP. Our EMAA software agents are much more complicated than the small software programs that GP usually evolves. In order to evolve meaningful agents in a realistic time frame our representation of GP agents needs to be at a higher level than simple Java instructions and their parameters. We have chosen as our main GP building blocks the tasks that an agent can do. The tasks in an agent's workflow can be combined by several programming constructs such as *execute-sequentially*, *loop*, *if-then-else*. These constructs define the set of our operators (functions). Each of the tasks mentioned previously has a name and an associated list of parameters that represent our terminal symbols. Therefore our GP tree has three types of nodes, not two as usual GP trees have. The highest level is the operator level, the next level is the task level and the lowest level is the terminal level. Each GP tree can be of any depth allowing both simple and sophisticated agents (see an example on Fig. 4).

Each of the operators takes always a certain number of parameters and those parameters need to be of a predefined type. For example *execute-sequentially* takes always two parameters, each being an operator or a task. The set of tasks contains all the tasks that an agent can perform. In general this set is quite large but at this point we are allowing only a small subset of tasks (about twenty tasks dealing with acquiring, filtering and reporting information, computing network statistics). The set of terminals contains all the possible parameters of the tasks in the task set. Since our tasks are diverse, their parameters vary widely in number and type. We are not in a position to obey the GP closure property that requires each function in the function set to be able to accept as its arguments, any value that may possibly be returned by any function in the function set and any value that may be assumed by any terminal in the terminal set. Since the closure property does not prevail, special steps are needed to generate feasible offspring.

Our GP agents have several syntactic restrictions similarly to syntactic restrictions existing in programming languages. For example in the GP individual tree the root can be an operator or a task but it cannot be a terminal. The largest number of restrictions concerns the task parameters. Tasks have different numbers of parameters and those parameters are of different types (e.g. latitude, proximity, e-mail address, file name). The parameter restrictions heavily affect the design of our GP tree structure and operators.

Our GP agents explore the search space by performing crossover and mutation. We use a one–point crossover that we restricted in comparison to a regular one-point crossover. The restricted crossover does not allow the formation of infeasible individuals. In regular crossover entire subtrees from the parents are swapped and because of the closure property, offsprings created are always feasible. The restricted crossover allows only swapping portions of the trees that will lead to feasible individuals. It will allow swapping two subtrees that have a task at their head but it will not allow swapping two parameters unless they are the same type.

In GP there are usually two types of mutation, each starting by randomly choosing a mutation point in the tree. The first type chooses randomly a mutation point, the subtree rooted at that point is deleted, and a new subtree is grown there using the random growth process. In the second type of mutation, a single terminal is exchanged into another terminal or a single function is exchanged into another function (with its subtrees staying the same as previously). For agent learning we use the first type of mutation described above but since it is very disruptive we use it with a low probability of occurrence. We are using mainly three more restrictive types of mutation that we defied:

a. Parameter mutation. It deals exclusively with parameters and it performs a mutation of one parameter value into another of the same type (e.g. “3” into a “7” or emaa@lmco.com into agent@lmco.com). This is a restrictive mutation operator since it cannot simply mutate one terminal symbol into another one but it needs to keep track of the range of values that a given parameter in a given task can take.
b. Task mutation. It performs the mutation of tasks and is capable of substituting at the mutation point one task for another. It differs from the usual GP function mutation because it does not keep subtrees (parameters in the case of agent learning) intact. It assigns standard or random parameters to each new task generated.
c. Operator mutation. It mutates one operator into another and it keeps most of the subtrees intact. However there are situations when the subtrees need to be changed (e.g. when mutating *if* into *execute-sequentially*, the conditional part is discarded).

An important part of GP is the fitness of an individual. Our ultimate goal is to make agents learn everything, similarly to what humans do. Consequently we do not want to give too much guidance to the GP individuals and our fitness function should be of a

general nature. The fitness should be some type of self-awareness function, describing how well an individual is performing. At this stage we want the agents to learn what to do when the network slows down/speeds up considerably. Our fitness function describes how slow/fast the network is in comparison to how it was previously.

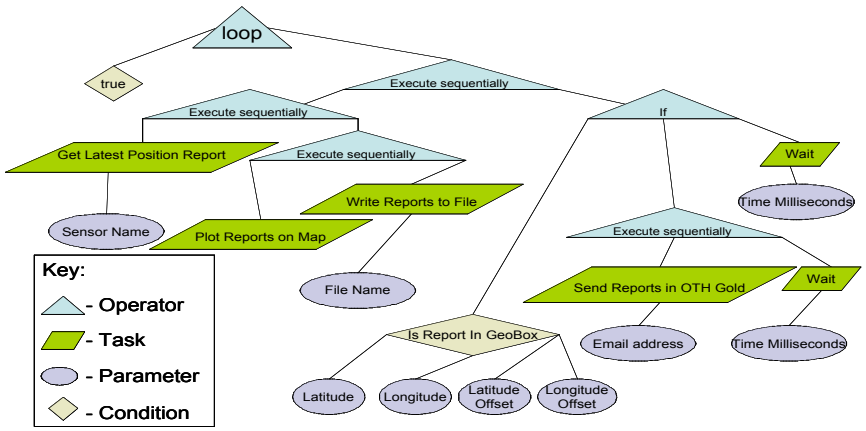

Figure 4. GP tree representation of an example agent from FBE experiments.

Our GP implementation is based on Evolutionary Computation in Java (ECJ) from University of Maryland. ECJ [ECJ] is a very flexible genetic algorithm and genetic programming freeware, written by Sean Luke et al. Its source code is readily available which allowed us to make all the modifications necessary for agent learning.

5. AGENT LEARNING EXAMPLE

One example of where learning is desirable is in the FBEs that used EMAA agents on the CoABS grid. One of the biggest challenges of this system is making sure that the network is not overloaded with data. One way to do this is to filter data, but the current filtering methods are static. In FBEs agents were used to pass relevant data to the correct participants; they stored data and kept track of who was online so that data could be refreshed when needed. While performing real-life FBE experiments, EMAA agents successfully performed all of their tasks that they were supposed to do. One type of agents was tasked to report all the tracks in a Geo Box with specific latitude-longitude coordinates. When the network was working at usual speed, the agents performed this task well but when the network became congested, the latency of reported tracks was too long. Humans observing the experiment wished that the agents decrease the size of Geo Box in order to provide tracks faster. It never happened since the agents could not learn. This is one of the simplest forms of learning that we want our agents to perform: adjust the task parameters. A more sophisticated form of learning is the one that requires agents to add/delete tasks and whole branches from their workflow. Again, while performing FBE experiments, a need for this type of learning became evident. Some of our agents were querying a database for relevant information and sending this information to the human. When unexpectedly a new database came on-line that had similar data, that database was not queried. Humans wished again that agents could figure on their own

that the new database needs to be queried as well. This type of learning would translate into adding a new branch to the workflow with a few tasks in it. The agent learning methodology that we devised will be able to address both types of learning.

Our system starts by generating an initial population of agents. Certain individuals in this population are the agents that humans generated while other members are generated randomly. The fitness function is a measure of the latency of the network in comparison to previous latency. The premise is that if the agent is getting/sending more information it gets rewarded, and if it is getting less information it gets punished. We are not telling in the fitness function what the agent should learn or how it should learn.

6. CONCLUSIONS

EPAL is a framework for software agents to automatically learn how to better perform. It is especially well suited for ATL EMAA agents but the methodology that we developed can be applied to any software agents, as long as they are described by a workflow that can be further decomposed into simple building blocks. Our method allows agents to perform two forms of learning: simple learning where only the task parameters are changed and advanced learning, where the agent's workflow can be completely changed, allowing agents to find and query new databases, send reports to new locations, or really to do anything for which a task exists in our task database. As our task database grows, agents should be able to "learn anything". We developed an extension of GP representation that allows representing agents as GP trees. This representation has three types of nodes (operators, tasks and parameters) rather than two types present in standard GP trees. For this representation, specific mutation and crossover operators have been developed in order to efficiently find agent workflows with increased performance.

The learning method developed is unique because it generates real software agents that can be executed in real military scenarios. We are not talking about toy problems and toy applications. To our knowledge the solution we formulated is the first platform in which real software agents can learn from their experience to perform standard tasks better. In the future we need to augment the set of tasks that agents can build their workflows from and to design a general self-awareness module. This self-awareness module will replace the fitness function that we are presently using.

7. REFERENCES

Bigus, J.P., Schlosnagle, D.A., Pilgrim, J.R., Mills III, W.N., Diao, Y., 2002, "ABLE: A Toolkit for Building Multiagent Autonomic Systems," *IBM Systems Journal,* Vol. 41, No. 3.

ECJ, www.cs.umd.edu/projects/plus/ec/ecj/.

Hofmann, M.O., Chacón, D., Mayer, G., Whitebread, K.R., Hendler, J., 2001, "CAST Agents: Network-Centric Fires Unleashed," *Proceedings of the 2001 National Fire Control Symposium*, Lihue, HI, August 12-30.

Hofmann, M.O., McGovern, A., and Whitebread, K.R., 1998, "Mobile Agents on the Digital Battlefield," *Proceedings of the Second International Conference on Autonomous Agents* Minneapolis/St. Paul, May 10-13, 1, pp. 219-225.

Koza, J., 1993, "Genetic Programming – On the Programming of Computers by Means of Natural Selection," Cambridge, Massachusetts, MIT Press.

Ramamurthy, U., Bogner, M., Franklin, S., 1998, "Conscious Learning in an Adaptive Software Agent," *Proceedings of Second Asia Pacific Conference on Simulated Evolution and Learning,* Canberra, Australia.

Stone, P., 1998, "Layered Learning in Multi-Agent Systems," PhD Thesis, CMU.