Towards building computational agent schemes

Gerd Beuster, Pavel Krušina, Roman Neruda[1], Pavel Rydvan[2]

[1]Institute of Computer Science, Academy of Sciences of the Czech Republic, Pod vodárenskou věží 2, 18207 Prague 8, Czech Republic, email: bang@cs.cas.cz. [2]Faculty of Mathematics and Physics, Charles University, Malostranské náměstí 25, 11000 Prague 1, Czech Republic

Abstract

A general concept of representation of connected groups of agents (schemes) within a multi-agent system is introduced and utilized for automatic building of schemes to solve a given computational task. We propose a combination of an evolutionary algorithm and a formal logic resolution system which is able to propose and verify new schemes. The approach is illustrated on simple examples.

1 Introduction

Hybrid models, including combinations of artificial intelligence methods such as neural networks, genetic algorithms and fuzzy logic controllers, seem to be a promising and extensively studied research area [1]. We have designed a distributed multi-agent system [6] called Bang 3 that provides a support for an easy creation of hybrid AI models by means of autonomous software agents [3].

Besides serving as an experimental tool and a distributed computational environment [4], this system should also allow to create new agent classes consisting of several cooperating agents. The *scheme* is a concept for describing the relations within such a set of agents. The basic motivation for schemes is to describe various computational methods. It should be easy to 'connect' a particular computational method (implemented as an agent) into hybrid methods, using schemes description. The scheme description should be strong enough to describe all the necessary relations within a set of agents that need to communicate one with another in a general manner.

Example: The genetic algorithm itself, from this point of view, consists of several parts: the *Genetics* agent, which performs the basic genetic algorithm logic and glues all parts together, the *Fitness* agent, that evaluates the fitness function for each individual, the *Operators* agent, that provides genetic operators, metrics operators, and creation operators, the *Selection* agent, that provides the selection of individuals. There are also two optional agent types that can further optimize overall performance: the *Shaper* agent, that provides global processing of population individuals fitness function values — such as sigma scaling — and the *Tuner* agent, that by exploiting information about the genetic algorithm performance (like best individual fitness, average fitness, first and second derivatives of these etc) tunes genetic operators probabilities (cf. Fig 1). It is supposed that there will exist more rival agents implementing a particular function (such as fitness evaluating) and it will be possible to choose among them.

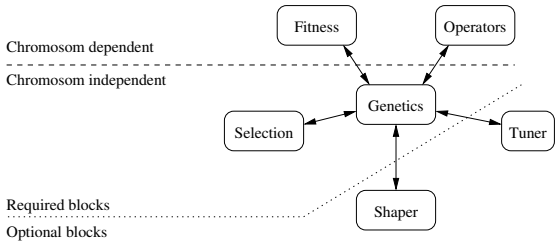

Fig. 1. Genetic algorithm as a multi-agent system.

This paper focuses on ways how to search the space of schemes representing a multi-agent system. The system consists of agents encapsulating individual computational methods, or their combinations. The behavior of the system is tested in the course of searching process by means of a given training dataset. In the case that the space is finite or 'small', variations of searching algorithms can be used (see e.g. [5]). We focused on employing an evolutionary algorithm together with the logics resolution system.

In the following section we present details on the scheme and evolutionary algorithm design. Next section treats a work on schemes as a logic constraint satisfaction problem. This approach can be used in two ways: either to generate new feasible solutions, or to verify solutions proposed by an evolutionary algorithm for their feasibility before they undergo the evolution. Such a hybrid approach neatly augments the evolutionary algorithm so it operates only on solutions that 'make sense'. Simple experiments and future work ideas conclude the paper.

2 Schemes

The scheme is a set of agents with a given topology of communication channels. The following mechanism for scheme specification has been designed.

The agents that comprise the scheme are called *building blocks*. Building blocks communicate through *incoming* and *outgoing gates*. Each agent can have any number of both incoming and outgoing gates.

One purpose of the schemes is that the hybrid computational methods designed in a form of scheme can be easily stored and used. Second, perhaps more interesting, challenge of the schemes concept is the *automatic scheme generation*. The scheme definition is a data structure consisting of the list of the building blocks and the interconnection among them. Actually, the scheme definition is a directed acyclic graph. This offers the possibility of automatic searching the space of schemes in order to find a suitable solution.

The proposed evolutionary algorithm operates on schemes definitions in order to find a suitable scheme solving a specified problem. The genetic algorithm has three inputs: First, the number and the types of inputs and outputs of the scheme. Second, the *training set*, which is a set of prototypical inputs and the corresponding desired outputs, it is used to compute the fitness of a particular solution. And third, the list of types of building blocks available for being used in the scheme.

We supply three operators that would operate on graphs representing schemes: *random scheme creation*, *mutation* and *crossover*.

The aim of the first one is to create a random scheme. This operator is used when creating the first (random) generation. The diversity of the schemes that are generated is the most important feature the generated schemes should have. The 'quality' of the scheme (that means whether the scheme computes the desired function or not) is insignificant at that moment, it is a task of other parts of the genetic algorithm to assure this. The algorithm for random scheme creation works incrementally. In each step one building block is added to the scheme being created. In the beginning, the most emphasis is put on the randomness. Later the building blocks are selected more in fashion so it would create the scheme with the desired number and types of gates (so the process converges to the desired type of function).

The goal of the crossover operator is to create offsprings from two parents. The crossover operator proposed for scheme generation creates one offspring. The operator horizontally divides the mother and the father, takes the first part from father's scheme, and the second from mother's one. The crossover is illustrated in Fig. 2.

The mutation operator is very simple. It finds two links in the scheme (of the same type) and switches their destinations. The mutation operator is illustrated in Fig. 3.

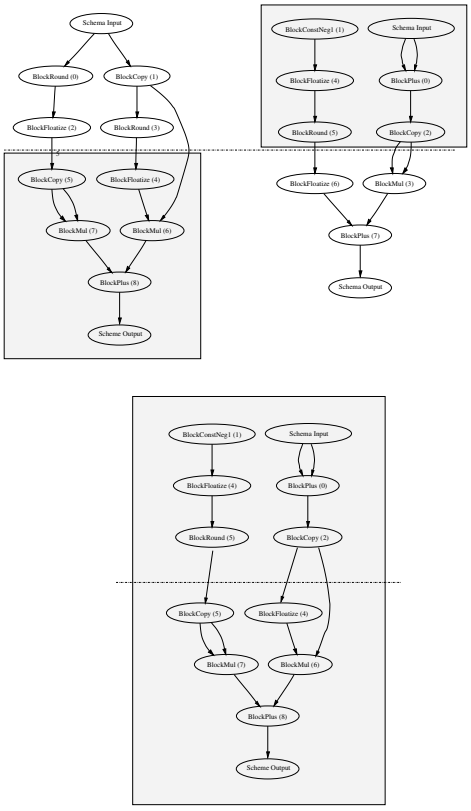

Fig. 2. Crossover of two schemes. The mother and father are horizontally divided and the offspring becomes a mixture of both.

3 Agent Constraints

Beside the genetic component, Bang also uses formal logics for the construction and evaluation of agent systems. Logics can be used both for the construction of new multi-agent systems, and for the verification of existing ones.

There are a number of applications for this:

- Sanity check of MAS

  When MAS configurations are generated automatically by a genetic algorithm, a lot of the system configuration will not work at all. Using the constraint satisfaction checking described before, it is possible to sort out these non-functioning systems without having to actual construct and test them.

- Fault Analysis

  When there are non-working part in user-constructed MAS, our constraint checking system can isolate the parts of the system that do not satisfy the constraints.

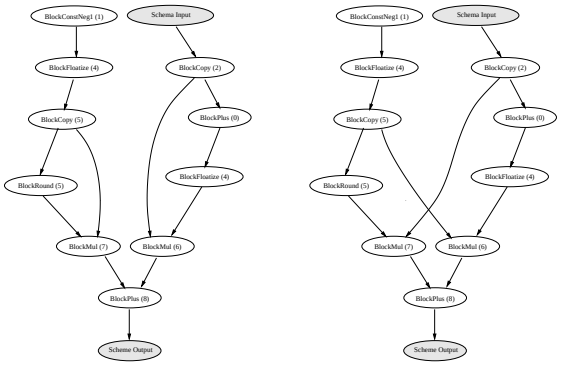

Fig. 3. Mutation on scheme. The destination of two links are switched.

- System Construction

  Given a — possible incomplete — description of a MAS and a set of constraints, our system can generate all MAS that satisfy the constraints. This can be used to automatically construct systems, or to assist the user: After the user has constructed a partial system, our system can aid the user in completing the system by showing possible extensions of the system

In order to apply formal logics to Bang, agent configuration is treated as a constraint satisfaction problem.[2] The logical description of a Bang 3 agent system consists of three parts: Descriptions of the agents, constraints on the individual agents, and constraints on the agent system as a whole.

Agent class definitions consist of a description of the agent's properties and of constraints on these properties, where agent descriptions are sets of terms and agent constraints are sets of horn-clauses over these terms:

Definition 1 (Agent Class Description) *An agent class description is a term* $\mathtt{agent\_class}(D, C)$*, where* D *is a description of the properties of the agent, and* C *is a set of constraints. The description of properties is a set of terms. Constraints are horn-clauses of the form*

$$\mathtt{constraint}(\mathtt{This}) \leftarrow C_1 \wedge C_2 \wedge \ldots \wedge C_n$$

with $C_1 \ldots C_n$ *terms, and* $\mathtt{This}$ *a free variable in* $C_1 \ldots C_n$.

Agents are derived from agent class descriptions by (partially) instantiating the terms of the agent class description.

A multi-agent system consists of a set of agents and constraints on the system as a whole:

Definition 2 (Multi-Agent System) *A multi-agent system description is a term* $\mathtt{mas}(A, C_s)$ *with* A *a set of agents,* C_s *a set of system wide constraints.* C_s *is a set of horn clauses of the form*

$$\mathtt{constraint}(\mathtt{This}) \leftarrow C_1 \wedge C_2 \wedge \ldots \wedge C_n$$

with $C_1 \ldots C_n$ *terms, and* $\mathtt{This}$ *a free variable in* $C_1 \ldots C_n$

When constraints are evaluated, variable $\mathtt{This}$ is unified with a description of the MAS. Reasoning is straight-forward: The terms can be transferred directly into a PROLOG-program. Valid configurations are generated by first attempting to satisfy the internal constrains of each agent and than satisfying the system-wide constrains. This is shown in algorithm 1.

Algorithm 1 Constraint Evaluation

Require: $\mathtt{mas}(A, C_s)$
  for all Agents $\mathtt{agent}(D, C) \in A$ **do**
    for all $\mathtt{constraint}(\mathtt{This}) \leftarrow C_1 \wedge C_2 \wedge \ldots \wedge C_n \in C$ **do**
      evaluate $\mathtt{constraint}(\mathtt{mas}(A, C_s))$
    end for
  end for
  for all $\mathtt{constraint}(\mathtt{This}) \leftarrow C_1 \wedge C_2 \wedge \ldots \wedge C_n \in C_s$ **do**
    evaluate $\mathtt{constraint}(\mathtt{mas}(A, C_s))$
  end for

So far, we have described a generic formalism for the definition of constraints onto multi-agent systems. In order to use this formalism with Bang 3, some standard terms and predicates have to be defined.

We start with a simple configuration problem, in which we are interested whether it is possible to connect agents to each other. For this, the following terms are used:

$\mathtt{name}(N)$ Where N is unique. This term is instantiated when an agent is created, and it serves as an identifier for the agent.

$\mathtt{gate}_{\mathtt{in}}(I, T)$ Where I is a name, and T is a data type. This term means that an agent has an input gate (i.e. an interface for receiving data) called I which is of type T.

$\mathtt{gate}_{\mathtt{out}}(I, T)$ Where I is a name, and T is a data type. This term means that an agent has an output gate (i.e. an interface for sending out data) called I which is of type T.

$\mathtt{float}$, $\mathtt{int}$, $\mathtt{string}$ These are basic data types.

$\mathtt{array}(T, A)$ A complex data type: Array of type T with arity A

A connection between two agents is valid if some output gate of the first agent matches an input gate of the second agent. This is expressed by the following formula:

$$\begin{aligned}&connects(A, A_{out}, B, B_{in}, C) \leftarrow \\ &\quad in(gate_{out}(A_{out}, Type), A) \wedge \\ &\quad in(gate_{in}(B_{in}, Type), B) \wedge \\ &\quad C = conn(A, A_{out}, B, B_{in})\end{aligned}$$

Here, A and B are agents, A_{out} is the name of an output gate of agent A, B_{in} is the name of an input gate of agent B, and in unifies the first argument with the appropriate term in the second argument.

A MAS is valid when all connections between input and output gates are valid. This can be checked recursively be the following predicate:

$$\begin{aligned}&check\_connections(MAS) \leftarrow \\ &\quad MAS = mas(A, C) \wedge \\ &\quad remove\_element(F, R, C) \wedge \\ &\quad F = conn([X, gate_{out}(X_{out})], \\ &\qquad [Y, gate_{in}(Y_{in})]) \wedge \\ &\quad connects(X, X_{out}, Y, Y_{in}, F) \wedge \\ &\quad check\_connections(mas(A, R))\end{aligned}$$

Extending the system to more complex relationships is straightforward. As an example, we show how a notion of trust among agents can be described. The general idea is that an agent A trusts a agent B if it either knows directly that the agent is trustworthy, or if agent A trusts a third agent C, and C trusts B.

This is captured by the following definition:

Definition 3 (Trust) *Agent X trusts agent Y if it knows the agent is trustable, or if it knows an agent M which trusts agent Y:*

$$\begin{aligned}&trust(X, Y) \leftarrow \\ &\quad agent(X, trusts(M)) \wedge trust(M, Y)\end{aligned}$$

With this additional relationship, it is easy to formulate a constraint "all agent gates should be matched, and only agents should be connected who trust each other":[1]

$$\begin{aligned}&trusted\_MAS = \\ &\quad (A, [check\_connections, \\ &\qquad check\_trust, Connections])\end{aligned}$$

$$\begin{aligned}&check\_trust(MAS) \leftarrow \\ &\quad MAS = mas(A, C) \wedge \\ &\quad remove\_element(F, R, C) \wedge \\ &\quad F = conn([X, gate_{out}(X_{out})], \\ &\qquad [Y, gate_{in}(Y_{in})]) \wedge \\ &\quad trust(X, Y) \wedge \\ &\quad check\_trust(mas(A, R))\end{aligned}$$

[1] $check\_connections$ and $Connections$ appear both in the set of constraints, because $Connections$, the actual connections of agents within the MAS, is itself a constraint onto the system.

4 Experiments

This section describes the experiments we have performed with generating the schemes using the genetic algorithm described above.

The training sets used for experiments represented various polynomials. The genetic algorithm was generating the schemes containing the following agents representing arithmetical operations: *Plus* (performs the addition on floats), *Mul* (performs the multiplication on floats), *Copy* (copies the only input (float) to two float outputs), *Round* (rounds the incoming float to the integer) and finally *Floatize* (converts the int input to the float).

The selected set of operators has the following features: it allows to build any polynomial with integer coefficients. The presence of the *Round* allows also another functions to be assembled. These functions are the 'polynomials with steps' that are caused by using the *Round* during the computation.

The only constant value that is provided is -1. All other integers must be computed from it using the other blocks. This makes it more difficult to achieve the function with higher coefficients.

The aim of the experiments was to verify the possibilities of the scheme generation by genetic algorithms. The below mentioned examples were computed on 1.4GHz Pentium computers. The computation is relatively time demanding. The duration of the experiment depended on many parameters. Generally, one generation took from seconds to minutes to be computed.

The results of the experiments depended on the complexity of the desired functions. The functions, that the genetic algorithm learned well and quite quickly were functions like $x^3 - x$ or x^2y^2. The learning of these functions took from tens to hundred generations, and the result scheme precisely computed the desired function.

Also more complicated functions were successfully evolved. The progress of evolving function $x^3 - 2x^2 - 3$ can be seen in the Fig. 4 and 5. Having in mind, that the only constant that can be used in the scheme is -1, we can see, that the scheme is quite big (comparing to the previous example where there was only approximately 5–10 building blocks) — see Fig. 6. It took much more time/generations to achieve the maximal fitness, namely 3000 in this case.

On the other hand, learning of some functions remained in the local maxima, which was for example the case of the function $x^2 + y^2 + x$.

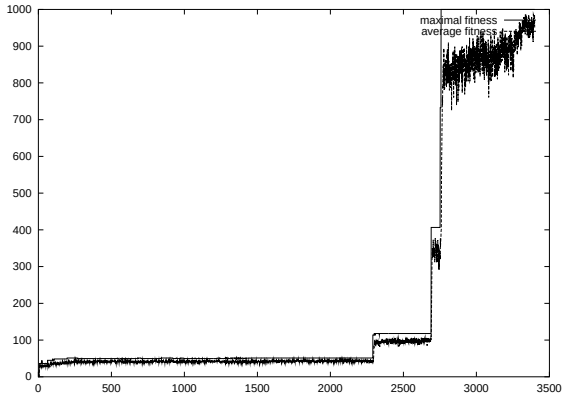

Fig. 4. Function $x^3 - 2x^2 - 3$. The history of the maximal and average fitness

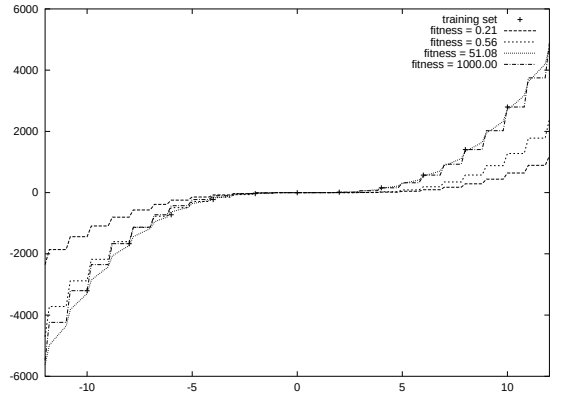

Fig. 5. Function $x^3 - 2x^2 - 3$. The best schemes from generation 0, 5, 200 and 3000

5 Conclusion

We have presented a hybrid system that uses a combination of evolutionary algorithm and a resolution system to automatically create and evaluate multi-agent schemes. So far, the implementation has focused on relatively simple agents computing parts of arithmetical expressions. Nevertheless, the sketched experiments demonstrate the soundness of the approach.

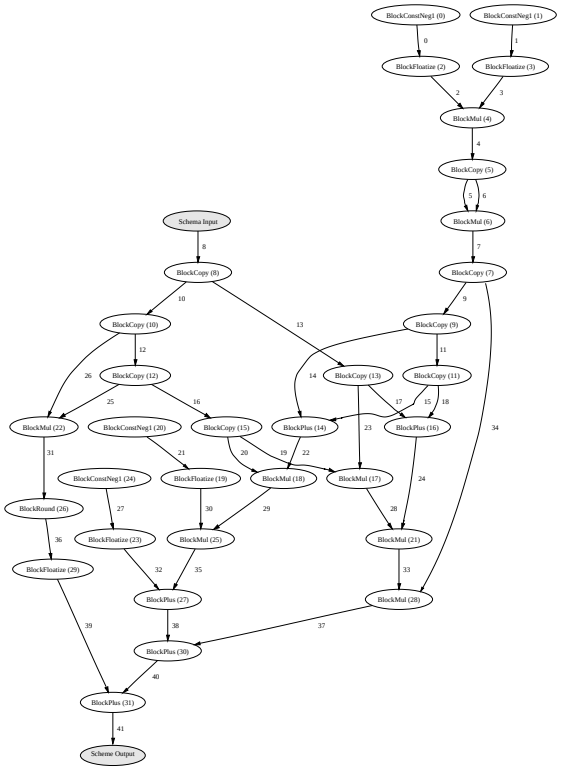

Fig. 6. Function $x^3 - 2x^2 - 3$. The scheme with fitness 1000 (out of 1000), taken from 3000th generation.

In our future work we plan to extend the system in order to incorporate more complex agents into the schemes. Our ultimate goal is to be able to propose and test schemes containing a wide range of computational methods from neural networks to fuzzy controllers, to evolutionary algorithms. While the core of the proposed algorithm will remain the same, we envisage some modifications in the genetic operators based on our current experience.

Namely, a finer consideration of parameter values, or configurations, of basic agents during the evolutionary process needs to be addressed. So far, the evolutionary algorithm rather builds the -3 constant by combining three agents representing the constant 1, than modifying the constant agent to represent the -3 directly. We hope to improve this behavior by introducing another kind of genetic operator. This mutation-like operator can be more complicated in the case of real computational agents such as neural networks, though. Nevertheless, this approach can reduce the evolutionary algorithm search space substantially.

We also plan to extend the capabilities of the resolution system towards more complex relationship types than the ones described in this paper. Our goal is to use ontologies for the description of agent capabilities, and

have the CSP-solver reason about these ontologies.

Since the computations are very time consuming, our next implementation goal is to design a distributed version of our algorithm and run it on a cluster of workstations.

Acknowledgments

This work has been partially supported by Grant agency of the Czech Republic under grant number 201/02/0428. G. Beuster has been partially supported by a DAAD postgraduate grant in the framework of the common special academia program III of the federal states and the federal government of Germany.

References

[1] P. Bonissone. Soft computing: the convergence of emerging reasoning technologies. *Soft Computing*, 1:6–18, 1997.

[2] Eugene C. Freuder Daniel Sabin. Configuration as composite constraint satisfaction. In George F. Luger, editor, *Proceedings of the (1st) Artificial Intelligence and Manufacturing Research Planning Workshop*, pages 153–161. AAAI Press, 1996, 1996.

[3] S. Franklin and A. Graesser. "Is it an agent, or just a program?": A taxonomy for autonomous agents. In *Intelligent Agents III*, pages 21–35. Springer-Verlag, 1997.

[4] R. Neruda, P. Krušina, P. Kudová, and Z. Petrová. Multiagent environment for hybrid AI models. In *Artificial Neural Nets and Genetic Algorithms. Proceedings of the ICANNGA 2001 Conference*, Vienna, 2001. Springer-Verlag.

[5] J. S. Russel and P. Norvig. *Artificial Intelligence: A Modern Approach*. Prentice-Hall International, 1995.

[6] G. Weiss, editor. *Multiagent Systems*. The MIT Press, 1999.