More Autonomous Hybrid Models in Bang^2

Roman Neruda*, Pavel Krušina, and Zuzana Petrová

Institute of Computer Science, Academy of Sciences of the Czech Republic,
P.O. Box 5, 18207 Prague, Czech Republic
roman@cs.cas.cz

Abstract. We describe a system which represents hybrid computational models as communities of cooperating autonomous software agents. It supports easy creation of combinations of modern artificial intelligence methods, namely neural networks, genetic algorithms and fuzzy logic controllers, and their distributed deployment over a cluster of workstations. The adaptive agents paradigm allows for semiautomated model generation, or even evolution of hybrid schemes.

1 Introduction

Hybrid models, including combinations of of artificial intelligence methods such as neural networks, genetic algorithms or fuzzy logic controllers, seems to be a promising and currently studied research area [2]. In our work [7] we have tested this approach in the previous implementation of our system with encouraging results on several benchmark test [7]. The models have included combinations as using genetic algorithm to set parameters of a perceptron network or fuzzy logic controller. Other example is setting learning parameters of back propagation (learning rate, decay) or genetic algorithm (crossover, mutation rate) by a fuzzy controller. Yet another example combination is using a fuzzifier or one back propagation step as a special kind of a genetic operator.

Recently we have turned our effort to more complex combinations, which have not been studied much yet, probably also because of the lack of a unified software platform that would allow for experiments with higher degree hybrid models. This is the motivation behind the design of the new version of our system called Bang^2 .

As before, the unified interface of the library of various AI computational components allows to switch easily e.g. between several learning methods, and to choose the best combination for application design. We have decided to allow components to run in distributed environment and thus to make use of parallel hardware architectures, typically a cluster of workstations. Second goal of Bang^2 design involves, beside creation of more complex models, also the semiautomated model generation and even the evolution of hybrid models.

For distributed and relatively complex system as Bang^2 it is favorable to make it very modular and to prefer the local decision making against global

* This work has been partially supported by GAASCR under grant no. B1030006.

V.N. Alexandrov et al. (Eds.): ICCS 2001, LNCS 2074, pp. 935–942, 2001.

intelligence. This lead us to the idea to take advantage of agent technology. Employing software agents simplifies the implementation of new AI components and even their dynamic changes. We also hope that some of the developed hybrid models will help the agents itself to become more adaptive and to behave more independently, which in turn should help the user of the system to build better models.

2 System Architecture

Bang^2 consists of a population of agents living in the environment, which provides support for creation of agents, their communication, distribution of processes. Each agent provides and requires services (e.g. statistic agent provides statistic preprocessing of data and requires data to process). Agents communicate via special communication language encoded in XML. There are several special agents necessary for Bang^2 run (like the Yellow Pages agent who maintains information about all living agents and about services they provide). Most of the agents realize various computational methods ranging from simple statistics to advanced evolutionary algorithms.

Our notion of intelligent agent follows the excellent introductory work by Franklin [5]. Generally, an agent is an entity (a part of computer program with its own thread of execution in our case), which is autonomous, reacts to its environment (e.g. to user's commands or messages from other agents) in pursue of its own agenda. The agent can be adaptive, or intelligent in a sense that it is able to gather information it needs in some sophisticated way. Moreover, our agents are mobile and persistent. We do not consider other types or properties of agents that for example try to simulate human emotions, mood, etc.

Bang^2 environment is a living space for all the agents. It supplies resources and services the agents need and serves as a communication layer. One example of such an abstraction is a location transparency in communication between agents — the goal is to make the communication simple for the agent programmer and identical for local and remote case while still exploiting all the advantages of the local one. There should be no difference from the agent point of view between communication to local and remote agent. On the other hand, we want to provide an easy way how to select synchronous, asynchronous or deferred synchronous mode of operation for any single communication act. The communication should be efficient both for passing XML strings and binary data.

As the best abstraction from the agent programmer point of view we have chosen the CORBA-like model of object method invocation. This approach has several advantages in contrast to the most common model of message passing. Among them let us mention the fact that programmers are more familiar with concept of function calling then message sending and that the model of object method invocation simplifies the trivial but most common cases while keeping the way to the model of message passing open and easy.

We have three different ways of communication based on the way, how the possible answer is treated: synchronous, asynchronous and deferred synchronous

(cf. 1). The synchronous way is basically a blocking call of the given agent returning its answer. In asynchronous mode, it is a non-blocking call discarding answer, while the deferred synchronous way is somewhere in between: it is a non-blocking call storing answer at a negotiated place.

Regarding the communication media, the standard way is to use the agent communication language, described in section 3. In order to achieve faster communication the agents can negotiate to use alternative binary interface (cf. 1) which does not employ the translation of binary data into XML textual representation and back.

Table 1. Communication functions properties: Sync is a blocking call of the given agent returning its answer, Async is non-blocking call discarding answer and Dsync is non-blocking call storing answer at negotiated place. BinSync and BinDsync are same as Sync and Dsync but the exchange binary data instead of XML strings. UFastNX is a common name for set of functions with number of different parameters of basic types usually used for proprietary interfaces.

Medium	XML strings	CData*	function parameters
Call	Sync	BinSync	UFastNX
Generality	High	Run-time	Hardwired
Speed	Normal	Fast	The fastest

From the programmer's point of view, an agent in Bang^2 is regular C++ classes derived from base class Agent which provide common services and connection to environment (Fig. 1). Agent behavior is mainly determined by its ProcessMsg function which serves as the main message handler. The ProcessMsg function parses the given message, runs user defined triggers via RunTriggers function and finally, if none is found, the DefaultBehavior function. The last mentioned function provides standard processing of common messages. Agent programmer can either override the ProcessMsg function on his own or (preferably) write trigger functions for messages he wants to process. Triggers are functions with specified XML tags and attributes. RunTriggers function calls matching trigger functions for a received XML message and fills up the variables corresponding to specified XML attributes with the values and composes the return statement from the triggers return values (see 3).

There are several helper classes and functions prepared for the programmers. Magic agent pointer, which is one of them, is an association of a regular pointer to Agent object with its name which is usable as a regular pointer to an agent class but has the advantage of being automaticly updated, when the targeted agent moves around.

The agent inner state is a general name for values of relevant member variables determining the mode of agent operation and its current knowledge. The control unit is its counterpart — program code manipulation with the inner state

and performing agent behavior, it can be placed in all ProcessMsg functions or triggers.

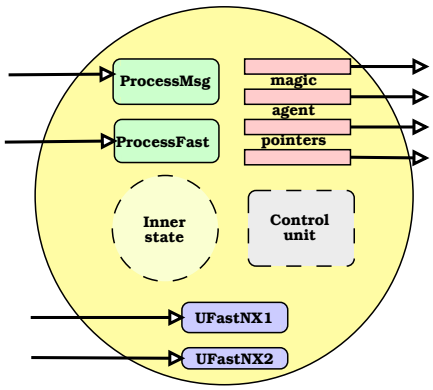

Fig. 1. The internal agent structure.

3 Communication Language

Agents need a communication language for various negotiations and for data transfer between them. The language should be able to describe basic types of communication, such as requests, acceptance, denial, queries. Also, the language should be able to describe quite wide range of data formats, such as the arbitrary data set on one hand, or the inner state of a neural network on the other. The language should also be human readable, to some extent, although there might be other tools that can provide better means of communication with the user. Last but not least, we expect reliable protocol for message delivery (which is TCP/IP in our case).

Several existing agent communication languages for agents already try to solve these problems. ACL ([4]) and KQML ([3] — widely used, de facto standard) are lisp-based languages for definition of message headers. KIF (KQML group — [6]), ACL-Lisp (ACL group — [4]) are languages for data transfer. They both came out of predicate logic and both are lisp-based, enriched with keywords for predicates, cycles etc. XSIL [8] and PMML [1] are XML-based languages designed for transfer of complex data structures through the simple byte stream.

Messages in Bang^2 have adopted XML syntax. Headers are not necessary, because of the inner environment representation of messages — method invocation — the sender and receiver are known. The first XML tag defines the type of the message (similar to message types defined in an ACL header). Available message types are:

- *request* (used when an agent require another agent to do something),
- *inform* (information providing),
- *query* (information gathering),
- *ok* (reply, no error),
- *ugh* (reply, an error occurs).

The content of the message (everything between outermost tags) contains commands (type request), information provisions, etc. Some of them are understandable to all agents (ping, kill, move, ...), others are specific to one agent or a group of agents. Nevertheless, agent is always able to indicate whether he/she understands a particular message. For illustration of agent communication language messages see figure 2.

```
<broadcast><halt/></broadcast>
<inform>
<created myid="!000000000001"
  name="Lucy"
  type="Neural Net.MLP"/>
</inform>

<ok>Agent Lucy, id=!000000000001,
type=Neural Net.MLP created</ok>

<request><ping/></request>
```

Fig. 2. Example of Bang^2 language for agent negotiation.

There are two ways how to transfer data: as a XML string, or as a binary stream. The former is human readable, but may lack performance. This is not fatal in agents' negotiation stage (as above), but can represent a disadvantage during extensive data transfers. The latter way is much faster, but the receiver has to be able to decode it. Generally in Bang^2 , the XML way of data transfer is implicit and the binary way is possible after the agents make an agreement about format of transferred data. For illustration of agent data transfer language see figure 3.

4 Conclusions and Future Work

For now, the design and implementation of the environment is complete. The support agents, including the Yellow Pages and basic graphical user interface agent (written in Tcl/Tk) are ready. We have started to create a set of agents of different purpose and behavior to be able to start designing and experimenting with adding more sophisticated agent oriented features to the system. A general genetic algorithm agent and a RBF neural network agent has been developed and tested so far, more is coming soon.

```
<query><vector row="45"/></query>
<query><vector/></query>
<ok><data separator=",">
Here are binary data
</data></ok>
<query><bin><query>
<vector/>
</query></bin></query>
<ok session="5" funcnum="1"/>
```

Fig. 3. Example of Bang^2 language for data transfer.

For experimenting with the more sophisticated agent schemes, we will focus on mirroring agents, parallel execution, automatic scheme generating and evolving. Also the concept of an agent working as the other agent's brain by means of delegating the decisions seems to be promising. Another thing is the design of load balancing agent able to adapt to changing load of host computers and to changing communication/computing ratio. And finally we think about some form of inter Bang^2 -sites communication.

In the following we discuss some of these directions in more detail.

4.1 Task Parallelization

There are two ways of parallelization: by adding an agent the ability to parallelize its work or by creating generic parallelization agent able to manage non-parallel agent schemes. Both have their pros and cons. The environment creates a truly virtual living space for agents, so the possibility for explicit inner parallel execution is there since the beginning. This approach will always be the most effective, but in general quite difficult to program.

On the other hand, the general parallelization agent can provide cheap parallelization in many cases. Consider an example of a genetic algorithm. It can explicitly parallelize by cloning fitness function agent and letting the population being fitness-ed simultaneously. Or on the other hand, the genetic algorithm can use only one fitness function agent, but be cloned together with it and share the best genoms with its siblings via a special purpose genetic operator. We can see this in figure 4, where agents of Camera and Canvas are used to automatize the sub scheme-cloning. Camera looks over the scheme we want to replicate and produces its description. Canvas receives such description and creates the scheme from new agents.

4.2 Agents Scheme Evolving

When thinking about implementing the task parallelization, we found it very useful to have a way of encoding scheme descriptions in a way which is understandable by regular agents. Namely, we think of some kind of XML description

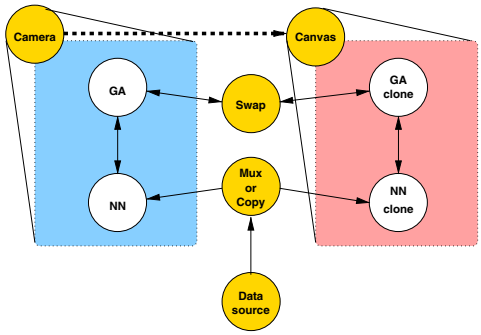

Fig. 4. Generic task parallelization.

of a scheme (which was in the previous version of the system represented in the custom Scheme Description Language). This lead to idea of agents not only creating and reading such a description, but also manipulating it.

Once we are able to represent a hybrid scheme, we can think of their automatic evolution by means of genetic algorithm. All we need is to create a suitable genetic operator package to plug into a generic GA agent. As a fitness, one can employ the part of generic task parallelization infrastructure (namely the Canvas, see fig. 5). For genetic evolving of schemes we use the Canvas for testing the newly modified schemes.

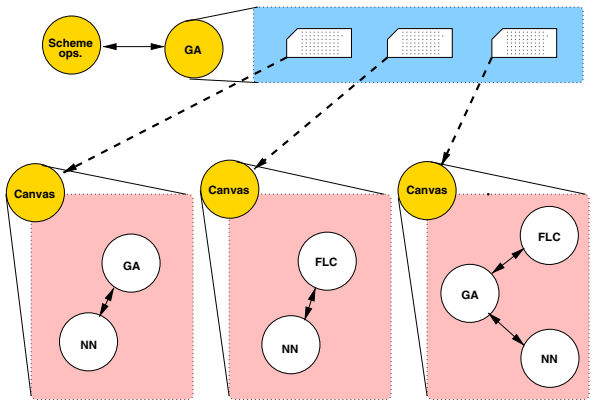

Fig. 5. Evolving a hybrid scheme by means of a genetic algorithm.

4.3 Agent as a Brain of Other Agent

As it is now, the agent has some autonomous — or intelligent — behavior encoded in standard responses for certain situations and messages. A higher degree

of intelligence can be achieved by adding some consciousness mechanisms into an agent. One can think of creating a planning agents, Brooks subsumption architecture agents, layered agents, or Franklin "conscious" agents.

Instead of hard-coding these mechanisms into an agent, we develop a universal mechanism via which a standard agent can delegate some or all of its control to a specialized agent that serves as its external brain. This brain will provide responses to standard situations, and at the same time it can additionally seek for supplementary information, create its own internal models, or adjust them to particular situations.

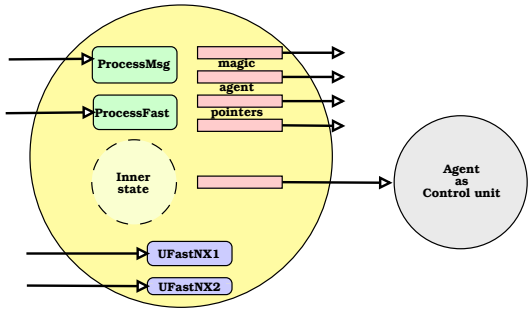

Fig. 6. Agent serving as an external brain of other agent.

References

1. PMML v1.1 predictive model markup language specification. Technical report, Data Mining Group, 2000.
2. Pietro P. Bonissone. Soft computing: the convergence of emerging reasoning technologies. *Soft Computing*, 1:6–18, 1997.
3. Tim Finnin, Yannis Labrou, and James Mayfield. KQML as an agent communication language. *Software Agents*, 1997.
4. Foundation for Intelligent Physical Agents. *Agent Communication Language*, October 1998.
5. Stan Franklin and Art Graesser. Is it an agent, or just a program?: A taxonomy for autonomous agents. In *Intelligent Agents III*, pages 21–35. Springer-Verlag, 1997.
6. Michael Genesereth and Richard Fikes. Knowledge interchange format, v3.0 reference manual. Technical report, Computer Science Department, Stanford University, March 1995.
7. Roman Neruda and Pavel Krušina. Creating hybrid AI models with Bang. *Signal Processing, Communications and Computer Science*, I:228–233, 2000.
8. Roy Williams. Java/XML for scientific data. Technical report, California Institute of Technology, 2000.