Autonomous Behavior of Computational Agents

Roman Vaculín[1], Roman Neruda[2]
[1]Faculty of Mathematics and Physics, Charles University, Malostranské nám. 25, Prague
[2]Institute of Computer Science, ASCR, P.O. Box 5, 18207 Prague, Czech Republic
Email: vaculin@cs.cas.cz, roman@cs.cas.cz

Abstract

In this paper we present an architecture for decision making of software agents that allows the agent to behave autonomously. Our target area is computational agents — encapsulating various neural networks, genetic algorithms, and similar methods — that are expected to solve problems of different nature within an environment of a hybrid computational multi-agent system. The architecture is based on the vertically-layered and belief-desire-intention architectures. Several experiments with computational agents were conducted to demonstrate the benefits of the architecture.

1 Introduction

Software agents can be seen as small self-contained programs that can solve simple problems in a well defined domain [6]. In order to solve complex problems agents have to cooperate and exhibit some level of autonomy. Autonomy, adaptivity, cooperation ability, and several other properties distinguish agents from "conventional" programs.

In this paper we present an architecture that allows simple design of adaptive, or intelligent, agents. The architecture enables the agent to solve problems of different nature within an environment of a computational multi-agent system, and thus increase its autonomy, adaptivity and the performance of the whole system. The architecture is implemented within a distributed multi-agent system *Bang3* that provides a platform for an easy creation of hybrid artificial intelligence models by means of autonomous agents (see [4]).

2 Computational agents

An *agent* is a computer system that is *situated* in some *environment*, and that is capable of *autonomous action* in this environment in order to meet its design objectives [9, Ch. 1]. *Autonomy* is used to express that agents are able to act (to perform actions) without the intervention of humans or other system.

[2]This research has been supported by the National Research Program Information Society project no. 1ET100300419.

An *intelligent agent* is one that is capable of *flexible* autonomous action in order to meet its design objectives, where flexibility means three things: *proactiveness* (goal-directed behavior), *reactivity* (response to changes), and *social ability* (interaction with other agents). Building purely *goal-directed* or *purely reactive* agents — one that continually responds to its environment — is not difficult in some environments. The problem is to build a system that achieves an effective balance between the goal-directed and reactive behavior, which strongly depends on the characteristics of the environment.

A *computational agent* is a highly encapsulated object realizing a particular computational method [5], such as a neural network, a genetic algorithm, or a fuzzy logic controller. The main objective of our architecture is to allow a simple design of adaptive autonomous agents within an environment of a computational multi-agent system. In order to act autonomously, an agent should be able to cope with three different kind of problems [8]: cooperation of agents, a computation processing support, and an optimization of the partner choice. The architecture we present is general in the sense that it can be easily extended to cope with different problems than those mentioned, nevertheless, we present its capabilities in these three areas.

Cooperation of agents: An intelligent agent should be able to answer the questions about its willingness to participate with particular agent or on a particular task. The following subproblems follow: (1) deciding whether two agents are able to cooperate, (2) evaluating the agents (according to reliability, speed, availability, etc.), (3) reasoning about its own state of affairs (state of an agent, load, etc.), (4) reasoning about tasks (identification of a task, distinguishing task types, etc.).

Computations processing: The agent should be able to recognize what it can solve and whether it is good at it, to decide whether it should persist in the started task, and whether it should wait for the result of task assigned to another agent. This implies the following new subproblems: (1) learning (remembering) tasks the agent has computed in the past (we use the principles of case-based

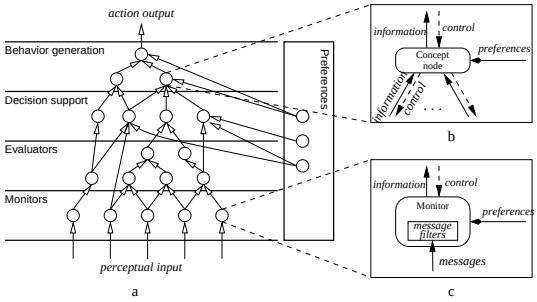

Fig. 1. Architecture —network of concepts (a); Concept node (b); Monitor (c)

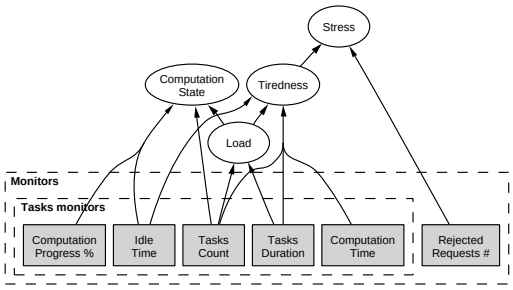

Fig. 2. Modeling state of an agent

learning and reasoning — see [2], [1] — to remember task cases), (2) monitoring and evaluation of task parameters (duration, progress, count, etc.), (3) evaluating tasks according to different criteria (duration, error, etc.).

Optimization of the partner choice: An intelligent agent should be able to distinguish good partners from unsuitable ones. The resulting subproblems follow: (1) recognizing a suitable (admissible) partner for a particular task, (2) increasing the quality of an evaluation with growing experience.

So, the architecture must support *reasoning*, *descriptions* of agents and tasks (we use ontologies in descriptions logics - see, e.g., [3]), *monitoring* and *evaluation* of various parameters, and *learning*.

3 Network of concepts

The architecture is organized into layers. Its logic is similar to the vertically-layered architecture with one-pass control (see [9, p. 36]). The lowest layer takes perceptual inputs from the environment, while the topmost layer is responsible for the execution of actions.

The architecture consists of four layers (see Figure 1): the *monitors* layer, the *evaluators modeling* layer, the layer for *decision support*, and the *behavior generation* layer. All layers are influenced by global *preferences*.

Global preferences allow us to model different flavors of an agent's behavior, namely, we can set an agent's pro-activity regime, its cooperation regime and its approach to reconsideration. *The monitors layer* interfaces directly with the environment. It works in a purely reactive way. It consists of rules of the form $condition \longrightarrow action$. *Evaluators modeling layer* is used to model more aggregate concepts on top of already defined concepts (either monitors or other evaluators). *Decision support layer* enables an agent to solve concrete problems. *Behavior generation layer* generates appropriate actions that the agent should perform, and thus controls the agent's behavior. The mechanisms for action generation and selection are provided by the BDI model (see [9, pages 55–61]).

The basic element of our architecture is a *concept node*. We can imagine a concept node as a class in some common object-oriented programming language, which defines explicitly its *dependences* on other concept nodes. The concept node is dependent on some other concept node if it needs some services provided by this other concept node in order to provide its own services. Each part of the architecture is defined as a concept node.

The network of concept nodes is a directed acyclic graph of concept nodes — see Figure 1 (a). Edges express dependences between concept nodes. This graph respects described layers. Figure 1 (b) shows a detailed view of a common concept node and Figure 1 (c) depicts a detailed view of a monitor. Each monitor can define several *filters* which represent rules as described above.

Explicitly defined dependences allow each agent to use only those those concept nodes that it really needs.

4 Modeling in the network of concepts

Evaluators are used to describe an agent's state, and to estimate services' quality of partner agents. They usually perform aggregations of several simpler concept nodes, typically the monitors. Typically, evaluators have the form of a non-linear real function that may differ in individual evaluators.

In order to describe an agent's state, we have defined four evaluators — *Load*, *Tiredness*, *Stress*, and *Computation state* (see Figure 2). For example, the *Load* evaluator depends on the count of currently running tasks and on their demandingness (complexity, etc.). We approximate the complexity of tasks by the average duration of past tasks. The load grows proportionally with the count of tasks and the average duration of tasks. The other evaluators can be described in a similar way.

The decision support concept nodes are used to represent particular decision problems and provide suggestions of how to solve these problems.

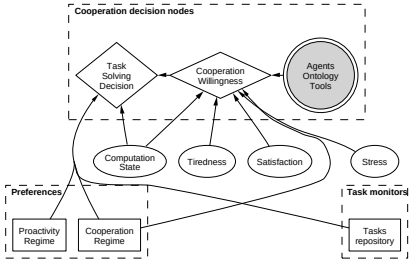

Fig. 3. Support for cooperation

Figure 3 depicts concept nodes that solve decision problems common in the area of agents cooperation. The *Agents ontology tools* concept node encapsulates reasoning services about agents' capabilities. The *Cooperation willingness* (CW) concept node suggests whether to cooperate with a particular agent or not. The *Task solving decision* concept node suggests whether to solve a particular task for a particular agent. For details about other areas of decision support see [7].

In the behavior generation layer, we use the BDI model (see [9]) to generate and choose the appropriate actions. The purpose of computational agents is to solve assigned tasks in an effective way. We distinguish two different situations:

1. If an agent does not use services of other agents in order to solve its task, it can perform the following basic actions: (a) *Accept / postpone / reject a new task,* (b) *Finish / interrupt a started task,* (c) *Evaluate task* (if there are some tasks in the tasks repository with incomplete information), d) *Find and solve new tasks.*

2. If an agent uses services of other computational agents in order to solve its own task, it acts as a simple task manager (an agent that assigns tasks to other agents), and it can further perform the following actions: (a) *Search for suitable partners,* (b) *Test and evaluate possible partners,* (c) *Distribute / redistribute task to partners.*

We base our implementation of the BDI model on the algorithm described in [9, pages 55–61]. According to it, we have to specify the implementation of a belief revision function (*brf*), an options generation function (*options*), a *filter* function, and an *execute* function. The monitors layer and the evaluators modeling layer represent the agent's knowledge about its environment, and thus stand for its beliefs. Beliefs are updated automatically by filters of monitors, which can be seen as the *brf* function.

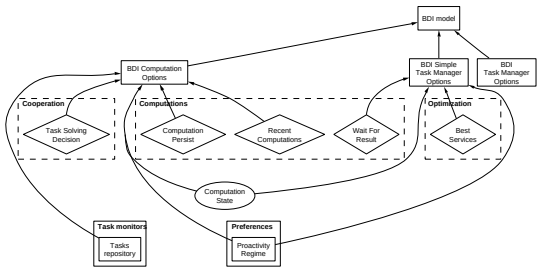

Fig. 4. BDI architecture within the network of concepts

Figure 4 shows concept nodes implementing the BDI architecture. The *BDI model* encapsulates the basic logic of the algorithm. We have further defined several concept nodes that are responsible for option generation. Each such a concept node implements its own *options* function and its own *filter* function which is responsible for filtering desires and intentions. The *action* function is implemented in the *BDI model* as defined by the pseudocode in the algorithm 1. B denotes the set of beliefs, I the set of intentions, D the set of desires, D_x the set of desires generated by the x-th option generation node, $ONodes$ a vector of all agent's option generation nodes.

Algorithm 1 *Action* function of the BDI agent.

1: $weight_{max} = 0$
2: **for** $i = 1$ to $ONodes.length$ **do**
3: $\quad \langle D_i, weight_i \rangle = ONodes[i].options(B, I)$
4: $\quad$ **if** $weight_i > weight_{max}$ **then**
5: $\quad\quad weight_{max} = weight_i$
6: $\quad\quad D = D_i$
7: $\quad\quad node_{max} = ONodes[i]$
8: $\quad$ **end if**
9: **end for**
10: **return** $node_{max}.filter(B, D, I)$

5 Experiments

We have adapted two existing computational agents embedding the multi-layer perceptron (MLP) and the radial basis function (RBF) neural network. These agents represent two different computational methods for the solution of similar categories of tasks.

Overheads of the architecture are summarized in Table 1. The creation of the agent takes 2-3 times longer since all the structures must be initialized. The communication overhead is around 30% when dealing with message delivering. However, in real-life scenario of task solving, the overhead is only about 10%.

	Without the arch.	With the arch.
Agent creation time	3604 μs	9890 μs
Message delivery time	2056 μs	2672 μs
Total computation time	8994681 μs	9820032 μs

Table 1. Comparison of the agent with and without the autonomous support architecture

	Error	Duration
Random choice	11.70	208710ms
Best speed	1.35	123259ms
Best Accuracy	1.08	274482ms
Best services	1.17	102247ms

Table 2. Optimization of the partner choice. Comparison of choices made by different criteria.

Table 2 summarizes the measured results of *optimization of the partner choice*. We simulated a usual scenario when an agent needs to assign some tasks to one of admissible partners. This agent uses a collection of different tasks and assigns them to the computational agents successively. The total duration of the computation and the average error of computed tasks were measured. A significant improvement of the efficiency can be seen.

Experiments with *optimization by reusing results* are summarized in Table 3. We have constructed several collections of tasks with different ratios of repeated tasks (quite a usual situation when, e.g., evaluating the population in genetic algorithms). We compared the total computation-times of the whole collection with and without the optimization enabled. We can see that the optimization is advantageous when the ratio of repeated tasks is higher than 20%. When more than 40% are repeated the results are significant.

Repeated tasks	Standard	Optimized
0 %	135777097	121712748
20%	94151838	90964553
40%	50704363	91406591
60%	47682940	90804052

Table 3. Optimization by reusing the results of previously-computed tasks (duration in milliseconds).

6 Conclusions

In this paper, we have described a general architecture that allows a simple design of adaptive software agents. It supports both agents' decision making and the generation of autonomous behavior. The architecture incorporates learning capabilities and support for reasoning based on ontologies which allows reasoning about agents' capabilities and activities and optimization of the performance.

The experiments have demonstrated that (1) it allows faster and more precise execution of tasks; (2) it supports a better cooperation of agents; (3) the performance drawbacks are not high.

The realized architecture provides several challenges for future work. The exchange and sharing of task cases can be a useful extension of the current implementation. We plan to perform more exhaustive experiments with groups / ensembles of cooperative computational agents. Finally, we plan to experiment with algorithms (e.g., by genetic algorithms) for automatic learning (generation) of global and local preferences of the architecture suitable for a particular situation (task).

References

[1] Agnar Aamodt and Enric Plaza. Case-based reasoning : Foundational issues, methodological variations, and system approaches. *AICom — Artificial Intelligence Communications*, 7(1):39–59, 1994.

[2] David W. Aha and Dietrich Wettschereck. Case-based learning: Beyond classification of feature vectors. 1997.

[3] Franz Baader, Diego Calvanese, Deborah McGuinness, Daniele Nardi, and Peter Patel-Schneider, editors. *The Description Logic Handbook*. Cambridge University Press, 2003.

[4] Bang3 web page. http://www.cs.cas.cz/bang3/.

[5] Roman Neruda, Pavel Krušina, and Zuzana Petrova. Towards soft computing agents. *Neural Network World*, 10(5):859–868, 2000.

[6] H. S. Nwana. Software agents: An overview. *Knowledge Engineering Review*, 11(2):205–204, 1995.

[7] Roman Vaculin. Artificial intelligence models in adaptive agents. Master's thesis, Faculty of Mathematics and Physics, Charles University, Prague, 2003.

[8] Roman Vaculin and Roman Neruda. Concept nodes architecture within the Bang3 system. Technical report, Institute of Computer Science, Academy of Science of the Czech Republic, 2004.

[9] Gerhard Weiss, editor. *Multiagents Systems*. The MIT Press, 1999.