Concept Nodes Architecture within the Bang3 System

Neruda, Roman
2005

Institute of Computer Science
Academy of Sciences of the Czech Republic

Concept nodes architecture within the Bang3 system

Roman Neruda, Roman Vaculín
Institute of Computer Science, ASCR,
P.O. Box 5, 18207 Prague, Czech Republic

Technical report No. 947

October 2005

Pod Vodárenskou věží 2, 182 07 Prague 8, phone: +420 266 051 111, fax: +420 286 585 789, e-mail:ics@cs.cas.cz

Institute of Computer Science
Academy of Sciences of the Czech Republic

Concept nodes architecture within the Bang3 system

Roman Neruda[1], Roman Vaculín
Institute of Computer Science, ASCR,
P.O. Box 5, 18207 Prague, Czech Republic

Technical report No. 947

October 2005

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.

Keywords:
Autonomous agents, Computational Intelligence, Decision Making, BDI architecture, Vertically-layered architecture

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

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 [8, Ch. 1]. *Autonomy* is used to express that agents are able to act (to perform actions) without the intervention of humans or other system.

An *intelligent agent* is one that is capable of *flexible* autonomous action in order to meet its design objectives, where flexibility means three things: *pro-activeness* (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: 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

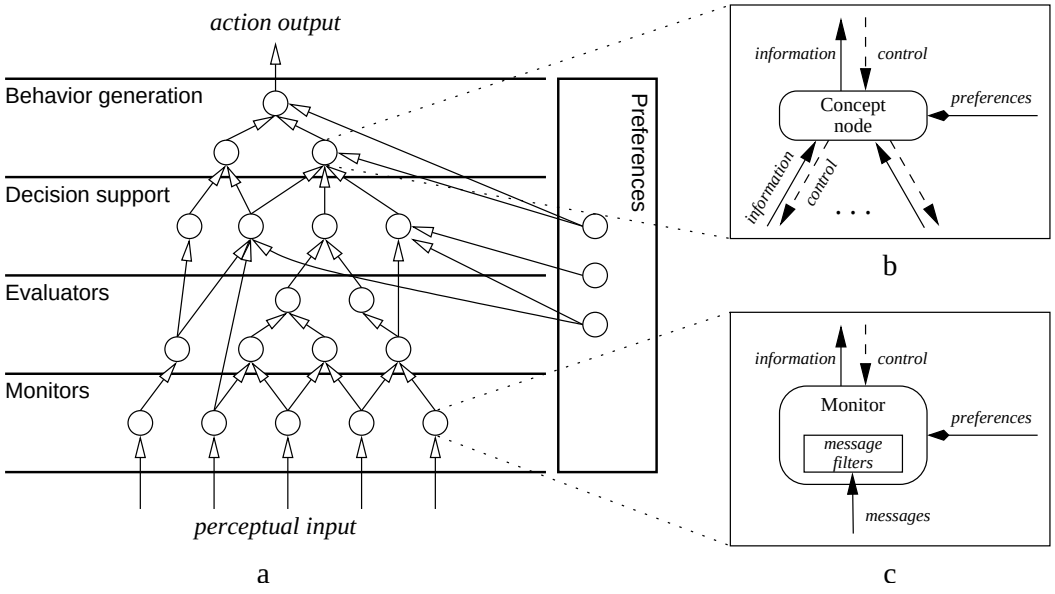

Figure 3.1: Architecture — network of concepts (a); Concept node (b); Monitor (c)

new subproblems: (1) learning (remembering) tasks the agent has computed in the past (we use the principles of case-based 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 [8, 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 3.1): 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*

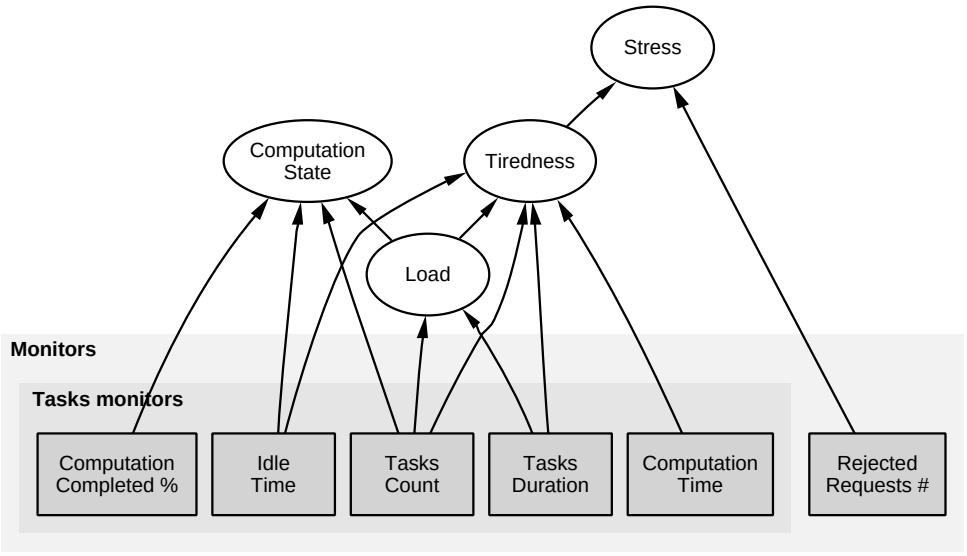

Figure 4.1: Modeling state of an agent

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 [8, 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 3.1 (a). Edges express dependences between concept nodes. This graph respects described layers. Figure 3.1 (b) shows a detailed view of a common concept node and Figure 3.1 (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.

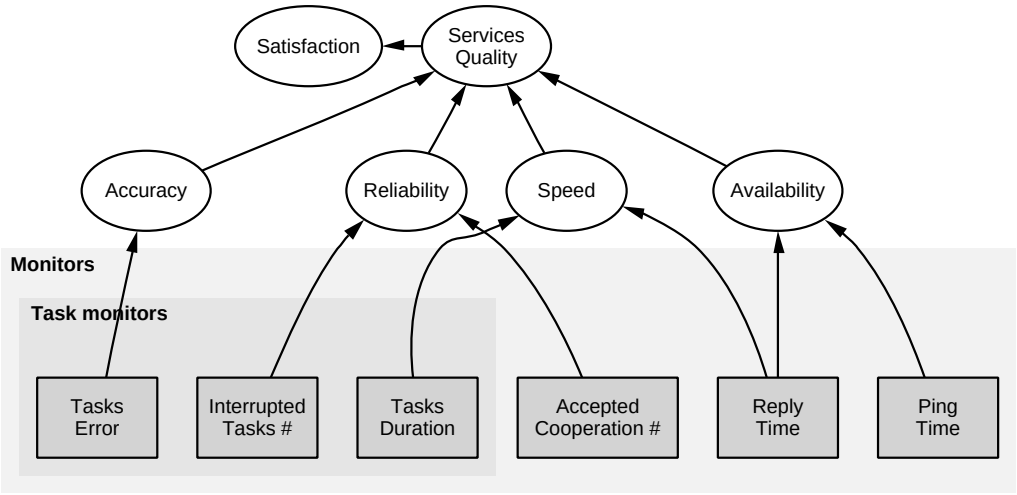

Figure 4.2: Measuring quality of services of partners

In order to describe an agent's state, we have defined four evaluators — *Load*, *Tiredness*, *Stress*, and *Computation state* (see Figure 4.1). 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 *Tiredness* evaluator reflects demandingness of agent's recent activities. Tiredness is computed as a sum of two components — instant tiredness that depends proportionally on the current load of an agent and the current computation time, and past tiredness that depends proportionally on the recent count of tasks and their average duration. When an agent is in idle state, tiredness is getting lower.

The *Stress* evaluator models nervousness and stress of an agent. It is a more experimental and fuzzy evaluator than the previous ones. The value of the *Stress* depends on the tiredness and on the count of rejected requests. If an agent is tired, it gets stress more easily. If it wants to cooperate with another agent and it is rejected, and if this happens often, then the stress coefficient grows.

The *Computation state* evaluator expresses the state of an agent. An agent can be either in the *idle* state or in the *busy* state. Besides this, some additional information may be provided. An agent is *waiting* if it is waiting for the result of some task that was assigned to another agent.

If an agent is in the busy state, the load coefficient, the count of running tasks and the progress percentage of the last started task are provided. If it is in the idle state, the idle period is provided. Finally, if an agent is in the waiting state, the count of assigned tasks is returned as a part of the state description.

Figure 4.2 shows four basic evaluators modeling different aspect of services quality — *Accuracy*, *Reliability*, *Availability* and *Speed*. Upon these evaluators, we build the aggregate

evaluator, *Services quality*, that stands for the total quality of services of a given agent. The *Satisfaction* evaluator represents an agent's satisfaction with another agent.

The *Accuracy* concept node calculates the accuracy of some agent. It depends inversely on the value of *Tasks error* monitor. The *Reliability* of an agent depends inversely on the percentage of interrupted tasks and on the percentage of rejected cooperation requests from this agent. The *Speed* of an agent depends inversely on its average reply time and on the average duration of tasks computed by this agent. The tasks duration has bigger (higher) effect on the resulting value. The *Availability* of an agent depends inversely on the ping time (instantaneous availability) and on the average reply time (long-term availability) of this agent. The *Services quality* is computed as a weighted sum of accuracy, reliability, speed and availability.

For almost all evaluators, we can specify (besides an agent for which we want to compute it) a secondary criterion that selects more precisely what exactly we want to compute. For instance, we can specify that we want to know the accuracy of an agent for a given task type (or task identifier, or instance identifier). The decision support concept nodes are used to represent particular decision problems and provide suggestions of how to solve these problems.

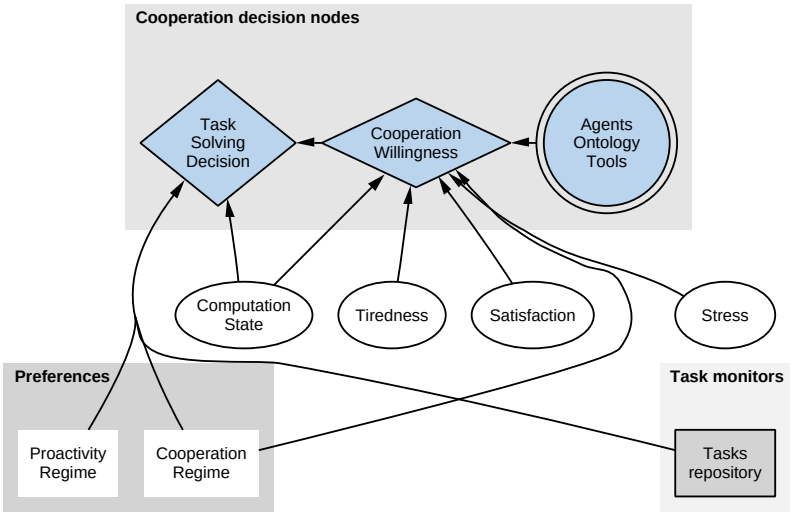

Figure 4.3: Support for cooperation

Figure 4.3 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 [8]) 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:

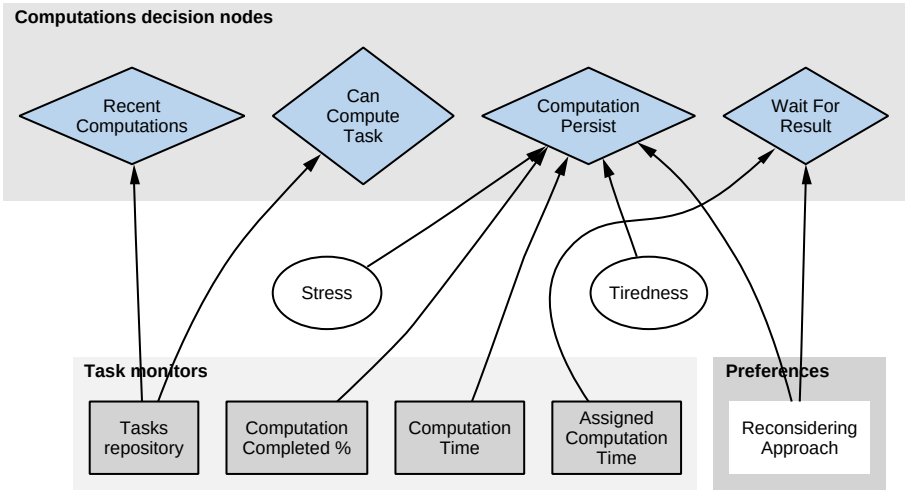

Figure 4.4: Computations processing support

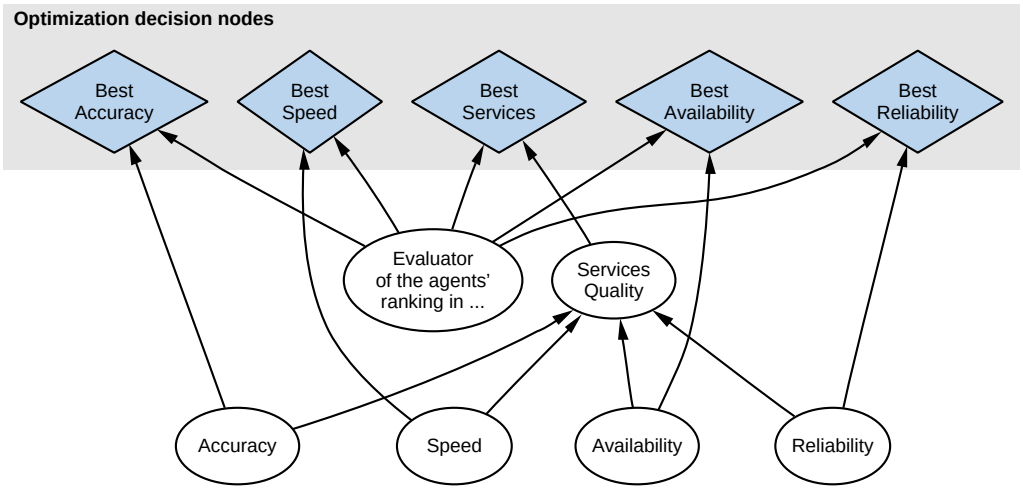

Figure 4.5: Optimization of partner choice

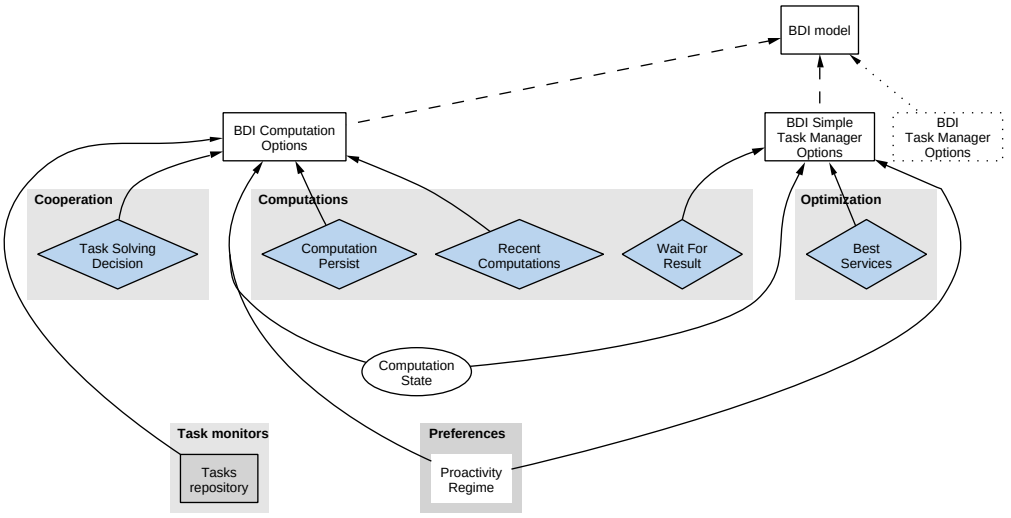

Figure 4.6: BDI architecture within the network of concepts

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 [8, 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.

Figure 4.6 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 pseudo-code in the algorithm 1 on the following page. 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:    ⟨D_i, weight_i⟩ = ONodes[i].options(B, I)
4:    if weight_i > weight_max then
5:       weight_max = weight_i
6:       D = D_i
7:       node_max = ONodes[i]
8:    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.

In different experiments we have combined these agents with other agents to demonstrate specific features of the architecture. We use the services of the *DataSource* agent, the *YellowPages* agent and the *Boss* agent. We have also developed several simple one-purpose agents that are used only for the experiment purposes.

The current state of implementation of the *Bang3* system does not allow to utilize all the features of the architecture immediately. Specifically, it is not possible to demonstrate the support for cooperation of more agents in the complex task solution scenarios since some auxiliary agents are not implemented yet (e.g., the task manager agent). *Overheads of the architecture* are summarized in Table 5.1. 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 5.1: Comparison of the agent with and without the autonomous support architecture

Table 5.2 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.

For the optimization of best service the total duration is reduced to 67246 milliseconds.

Experiments with *optimization by reusing results* are summarized in Table 5.3. We have

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

Table 5.2: Optimization of the partner choice. Comparison of choices made by different criteria.

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

Table 5.3: Optimization by reusing the results of previously-computed tasks (duration in milliseconds).

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.

The aim of the following experiment is to present the overall behavior of agents that take advantage of different capabilities of the developed architecture. In our model scenario, we use two computational agents (the MLP agent Percy and the RBF agent Rafael) and two very simple task manager agents (Manager1, Manager2) that successively assign tasks to the computational agents. In this scenario, computational agents use the behavior generation layer for regulation of their behavior. The task manager agents use the support for optimization of partner choice and the support for computations. In the real situations it can happen that some computation can take too much time to get finished. We have constructed a collection of tasks with some tasks taking much time. We conducted several experiments with different setting of parameters of particular agents. All agents benefit of the *Wait for result* and the *Commutation persist* concept nodes.

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).

Extension of the expressive power of our description language could be explored. This could be achieved by finding less restrictive limitations while maintaining the acceptable polynomial time complexity.

Bibliography

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