CLWAR/EURON Workshop on Robots in Entertainment, Leisure and Hobby
December 2 – 4, 2004, Vienna, Austria

BUILDING CONTROL SYSTEM OF MOBILE ROBOT WITH AGENT-SPACE ARCHITECTURE

Andrej Lúčny

Faculty of Mathematics, Physics and Informatics, Comenius University, Bratislava, Slovakia
MicroStep-MIS

Abstract

Agent-Space architecture was proposed for building of complex systems [8]. According to this architecture, a control system is implemented as a set of reactive agents which communicate indirectly through another entity called *space*. It is derived from subsumption architecture and related philosophy of R. Brooks and M. Minsky but it employs terms of modern approach of multi-agent systems. However, agents do not represent cooperating robots here, but units of a control system of a single robot. The presented architecture can be considered as an application of agent-oriented programming in domain of mobile robotics. Its most interesting features are dataflow which supports a many:many relationship, implicit sampling, high ability to modify and ability to model competition among internal structures. The key idea of the architecture is to allow only indirect communication among agents. It is realized by reading and writing named data. The data are stored as blocks in the space which also controls their time validity. Each such a block realizes a dataflow among several agents. It can be read and/or written by several of them. How do agents know about presence of particular data (which should be processed by them) in the blocks? There are two solutions: an agent can regularly read blocks (and process what is stored there regardless it is something new or not) or it can receive a notification about their changes.

We demonstrate capabilities of the architecture by implementation of a control system for a mobile robot which follows a ping-pong ball. An incremental development of the control system is presented. We start with a chain of reactive agents which model a traditional pipe-line structure of image processing and action selection. Even this structure is not a pure equivalent of the traditional approach. But advantages appear just when we try to improve capability of the robot to operate under various lighting conditions. We clone a part of the control structure to run its several instances in parallel, each using a different configuration for individual condition. Unlike usual approaches these instances do not cooperate but they compete. However, there is no negative impact of the competition to the resulting behavior of the robot. The only result is that it works under more general conditions. We introduce some further modifications of the control system.

Keywords : reactive agents, indirect communication, multi-agent systems, agent-oriented programming, subsumption architecture

1. Agent-oriented programming

Subsumption architecture [1,2] invented in eighties and its later derivates are still inspiring for domain of mobile robotics. However, the original framework for design based on augmented finite-state automata, wires, suppressors and inhibitors is not attractive nowadays and we are able to use terms of more modern approaches. The most suitable approach - in our opinion – is agent-oriented programming (AOP). We can consider AOP as a next generation of programming which is coming after structured and object-oriented programming. The key difference among these three is a kind of transferring real world entities into the computer. The oldest approach is based on transferring of passive entities. They are represented by records and they can be just manipulated. The next approach provides transfer of reactive entities - objects, which are able to act but only when they are called. AOP provides a next step and transfers proactive entities – agents [3], which are constantly active and need not to be called. The difference is demonstrated in Fig. 1.

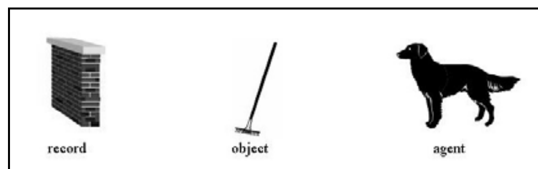

Fig. 1. A funny example of a passive, reactive and proactive entity. (A wall cannot injure you; just you can injure yourself on a wall. A rake can injure you, but at first you have to step on it. However, a dog can suddenly injure you without any your activity)

2. Agent-Space Architecture

We use AOP to define an own modern derivate of subsumption architecture. Within the architecture,

systems consist of many *reactive agents* and one *space*. *Space* is a server with an ability to contain named data units (called blocks) with a given time validity. *Reactive agents* are its clients which can write to blocks, delete them and read them as well as they can get a notification about their change (called a *trigger*). The notification service is not inevitable since agents are usually woken up by the notification from a *timer* as displayed in Fig. 2a. However, it can be very useful when an agent has to process some data swiftly, the data is coming in irregular instants of time and there is not enough power to cover the redundancy of analogical regular processing (Fig. 2b). Unlike in most of other AOP architectures, direct communication among agents is not allowed and we rely on indirect (sometimes called stigmergic) communication exclusively.

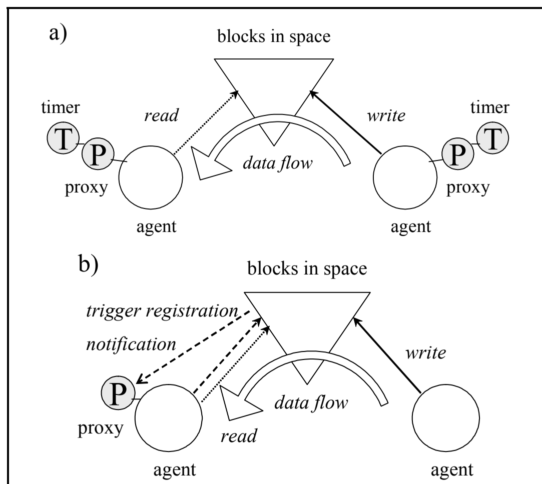

Fig. 2. A dataflow among agents. a) A dataflow from an *agent* A through *space* to an *agent* B which is activated regularly by a timer. b) An analogical dataflow to the *agent* B which is activated by a trigger, i.e. immediately when data are written to the *space.*

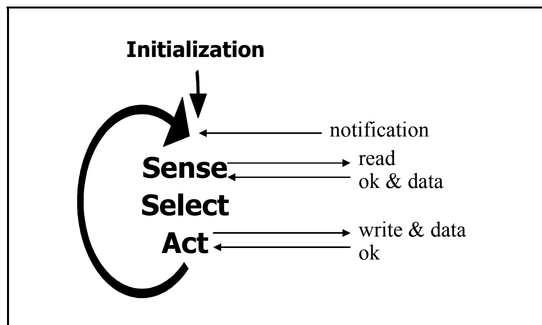

Fig. 3. A typical structure of a code of reactive agents

The *reactive agents* perform a simple sense-select-act cycle (reading from *space*, computing a reaction, writing to *space*). Any course through the cycle is activated by a notification from a *timer* or a *trigger* (and in special cases also by a device or by a user). There is no requirement to use a specific algorithm during the select phase like planning or learning. We prefer to use an ordinary code there and try to achieve a more complex behavior rather by interaction among agents than due to a clever "cognitive" component put into agents (therefore we attributed to the agents the adjective: *reactive*). The usual code structure of a *reactive agent* is depicted in Fig. 3.

This architecture exhibits two main features. The first one is a free communication among producers and consumers. Unlike other concepts of indirect communication, e.g. LINDA tupple space [4], we do not require calling an operation for creation of a block before its use; the block is created by the first write operation. Blocks can also become empty when their validity expires. Thus agents have to handle situation when they read a non-existing or empty block; e.g. each agent specifies an individual default value which is used instead of the missing value. Due to these arrangements we are able to realize dataflow, which supports a many:many relationship, in a comfortable way (Fig. 4).

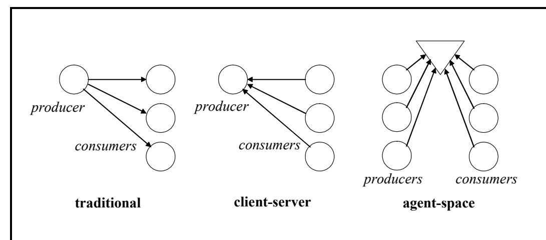

Fig. 4. A comparison of dataflow which is typical of the agent-space architecture and other approaches. The same data can be produced by many producers and processed by many consumers.

Consequently it is possible to restart a producer without any impact on consumers (useful for the recovery from errors), or to replace a producer by another one (useful for the ability to configure).

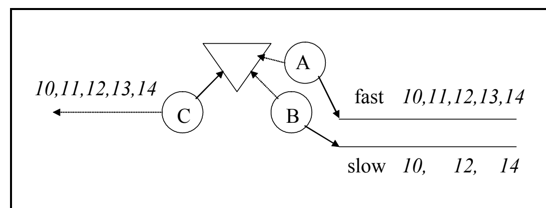

Fig. 5. An implicit sampling: *Agents* A and B perform the same code engaged in data distribution through a network. However the *agent* A, working on a fast medium, distributes all the data written by the agent C, while the *agent* B, working on a slow medium, is able to process just every second data.

The second main feature is implicit sampling. It is based on the fact that producers overwrite data in blocks regardless of whether consumers have read

them or not. Additionally, if a consumer receives more notifications during single course through its cycle, it recognizes them as a single notification. Implicit sampling appears when a consumer is too slow to undertake all data generated by their producer (Fig. 5). Therefore the implicit sampling is important for the real-time operation: the data, which cannot be processed quickly enough, are automatically lost (sampled). This feature is unacceptable for some applications (like counting money) but very profitable for other domains (like control systems of mobile robots).

Additional benefit of the agent-space architecture resides in the separation of a domain-dependent code from a data-exchange code and in the unification of data-exchange interfaces (reusability).

Regarding development process, agent-space architecture is designed for systems which have tens, even hundreds of versions. It enables us to consider the modification ability as a crucial factor of development. A simple modification is usually realized by one of the following ways:

- by adjustment of certain agent codes
- by development of new agents which alternate some former ones
- by development of some additional agents which profit from the former agents but have no impact on their operation (they implement additional features only)

On the other hand, we are also able to use the same strategy as subsumption architecture:

- to develop new agents which would be able to influence behavior of the former ones in a proper manner (without an adjustment of their code).

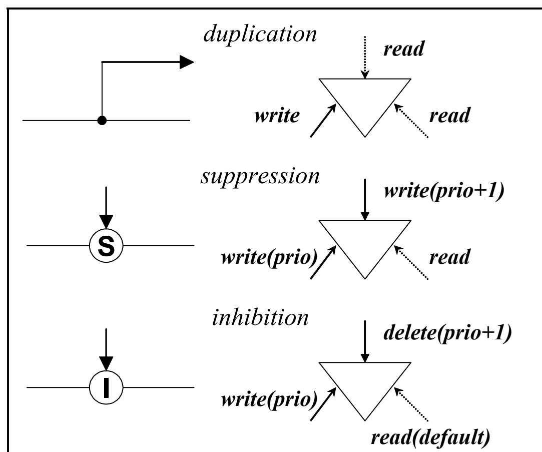

Fig. 6. Implementation of subsumption mechanisms in agent-space architecture

This influence can be realized by writing to former blocks to which some former agents are "sensitive" (Fig. 6). However, unlike Brooks' *suppressor* and *inhibitor*, our kind of influence is not based on strict priorities. Despite writing to a block by a new agent, the former producers remain active and compete with it. Of course, the priorities can be added, e.g. we can define a priority for any block in *space* and discard those write operations which try to overwrite the current value by a value with a lower priority. (Thus priority will be an additional parameter of the write and delete operations.) However, we are still able to model competition among agents (which operate on the same level of priority), while subsumption architecture is not able to provide the same.

3. Incremental development and competition

In nature, the competition among internal units seems to be an important creative principle. Sometimes we can even observe its presence in global behavior of living systems, mainly when we let them operate under exceptional or artificial conditions. Therefore it is acceptable also for artificial systems which follow ideas of biomimetics. However, can it be also profitable? Well, we are not sure – it is not possible to answer this question without many projects which will try to use competition for building of artificial systems. Anyway, we can offer an architecture which supports this elaboration. Further we are able to demonstrate several simple examples. We present one of them in the following.

We have the following task: to develop a mobile robot which follows a ping-pong ball. The robot is equipped with a camera scanning a scene in the front of it and motors which allow robot to rotate on the spot, to move forward and backward or to stop.

Fig. 7. A simple robot following a ping-pong ball

The traditional control system of such a robot is based on a pipeline which connects input images with output moves through several phases of image processing:

– color image is converted into greyscale
– Sobel's operator is applied
– points with intensity higher than a given threshold are selected to represent thick edges
– thick edges are turned to thin ones
– points of thin edges are represented as a set of segments
– potential centers of circles are recognized and

completeness of their associated circular lines is checked; position and size of a circle is recognized

- if the position is too on the left, we rotate to left, if it is too on the right, we rotate to right. Otherwise, if the recognized circle is too small, we move forward, if it is too big, we move backward.

If we use agent-space architecture, we can start from the same organization. However, we have to model the pipeline to a sequence of agents which exchange data through intermediate blocks (Fig. 8).

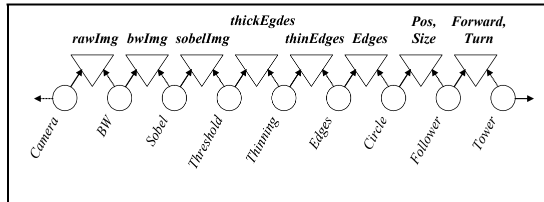

Fig. 8. A pure pipeline realized by agent-space architecture

At this stage of development, our control system is a pure pipeline realized in an unconventional manner. The only difference is that its computation does not run sequentially through the individual phases of processing but each phase is running in an own thread – as it is represented by an individual agent. Such an agent is running a cycle which is invoked by a timer with a given frequency or by a trigger indicating changes of some particular blocks. For example, the agent *Camera* is running it cycle with a frequency which is equal to the refresh rate of the camera, while the agent *Sobel* employs a trigger on the block containing the grayscale image. In principle, all agents except *Camera* could be invoked by trigger, however we have to take into acount that - in comparison with the refresh rate of the camera - some of them are too slow. Therefore we use timers also for agents *Threshold* and *Follower* and we set them to lower frequencies. Also we use a timer for the agent *Tower* and we set it to a frequency which is appropriate for physical capability of communication between the control system and motors. Although it is not necessary to use these timers – due to implicit sampling a slow agent simply looses some notifications and treats as much as possible – it is better to let the system to be sometimes idle.

Such a control system follows the ball quite successfully. However, we will find out soon that its success depends on lighting conditions. The system requires an improvement and at this moment we start to get a profit from the employed architecture. The pipeline can be easily turned into a more complicated structure which reflects that the optimal threshold for the edge detection is varying (Fig. 9). We are able to realize this improvement without any modification of the formerly developed agents except the agent *Threshold*. It is necessary to modify this agent because the threshold value was originally a part of its internal state, i.e. hidden to other agents. When we would like to prevent similar complications, we have to put such parameters into blocks from beginning. In general we can require revealing any information which can persist from one course through agent cycle to the next one. Agents which follow this rule are so-called purely reactive.

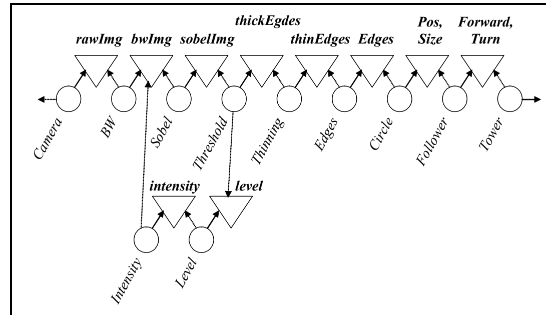

Fig. 9. An example of profit from pure reactivity

However, our solution is still not perfect since it is difficult to calculate the optimal value of threshold as a global parameter of image: the left part can be light and the right part dark – thus we will need two different thresholds, not their average. We need to let several different thresholds to compete. Thanks to our architecture we can realize it easily: we just launch agents from *Threshold* to *Circle* several times in parallel (Fig. 10).

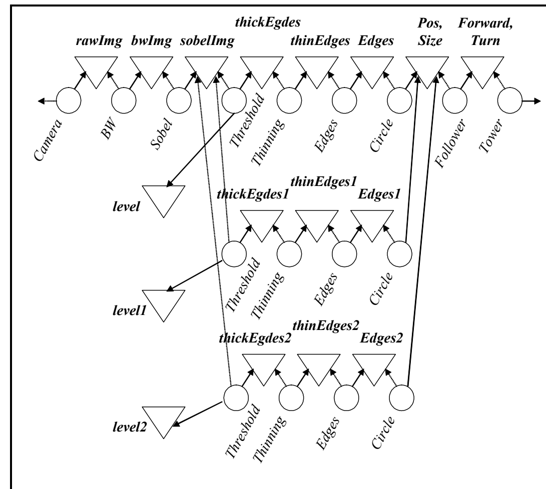

Fig. 10. An example of competition

In this way we implement the competition by writing of various recognized positions and sizes into a common block. Of course, this can work well only if the agent *Circle* write nothing into the block when it is not able to recognize something reasonable – it must not write a “bad value” there. As a result, the reasonable value should have certain time validity - otherwise we still recognize a ball after it is removed from the scene. Then - of course – the agent *Follower* needs to handle the problem of reading an empty

block, for example by using "no ball" as a default. In general, this arrangement (writing nothing instead of "bad value", specifying certain time validity and reading with a default) is the best kind of manipulation with blocks within agent-space architecture.

Another rule, which we can derive from this situation, is that the particular names of blocks which agents manipulate should be submitted as their parameters. Then we can easily re-use the agents and let them to operate over blocks which have the same meaning but different names.

Anyway, now we have implemented quite a robust system which operates under various lighting conditions (Fig. 11).

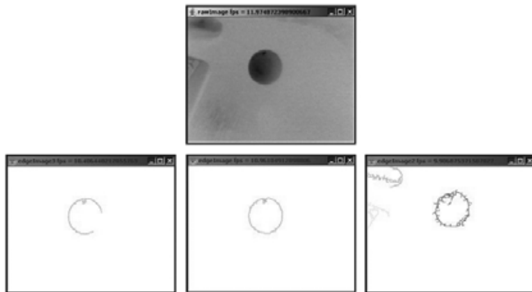

Fig. 11. An original image (negative) and results of competing recognizers with various thresholds

Since there is no priority and the "winning" proposal of position and size is anyone which is undertaken by the agent *Follower*, it can easily happen that the acquired values vary. This variation can be even significant – e.g. when there are two balls in the scene. Of course, this variation could be suppressed by adding priorities. However, much better solution can be realized by adding a new purely reactive agent which represents concentration (Fig. 12). That agent undertakes varying values, but it selects one of them and remembers it. Then the agent ignores any value which differs too much from the remembered one. On the other hand, if the undertaken value differs just a little, it is taken as a new selection. Thus the agent filters the significant variation and provides concentration on one ball.

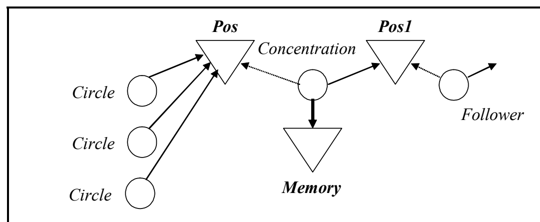

Fig. 12. An example of variation filtering

Since the remembered value has a persistent character, we store it in a block. Thus agent *Concentration* has to read the same value which it has written during the previous course through its cycle. This seems to be redundant but it brings two advantages. At first, we can use time validity to define such a timeout that its expiration indicates loss of the selected ball and the agent should look for another one. Secondly, we are able to remove content of this block when we want to force the agent to deal with another ball.

In this way, we could follow incrementally to more and more complicated solutions which successfully handle more and more conditions.

4. Conclusion

We introduced an architecture which can be considered as a modern reformulation of inspiring but old-fashioned subsumption architecture. The reformulation was based on agent-oriented programming with focus on indirect communication among agents. We discussed main features of the architecture. We presented how mechanisms of subsumption can be expressed within the architecture, thus we found that any solution based on subsumption architecture can be transferred into our architecture. Additionally, we demonstrated that our architecture is able to model not only priority-based cooperation but also competition among internal units. Some advantages and important details of this approach were demonstrated on a particular example.

5. References

[1] Brooks, R (1991). "Intelligence without representation", **Artificial Intelligence** 47, pp. 139-159

[2] Brooks, R. (1999). **"Cambrian Intelligence"**. The MIT Press, Cambridge, Massachusetts

[3] Doran, J. (1992). "Distributed AI and its Applications". In: **Advanced Topics in Artificial Intelligence** (Mařík V., Štepánková O., Trappl R.) Springer-Verlag, Berlin, pp. 368-372

[4] Gelernter, D. (1985). "Generative Communication in LINDA". **ACM on Transactions on Programming Languages and Systems**, Volume 7(1), pp. 80-112

[5] Jennings, N. (2000). "On agent-based software engineering". **Artificial Intelligence** 117, 2000, pp. 277-296.

[6] Kelemen, J. (2001). "From statistics to emergence - exercises in systems modularity". In: **Multi-Agent Systems and Applications**, (Luck, M., Mařík, V., Štepánková, O. Trappl, R.), Springer, Berlin, pp. 281-300

[7] Lúčny, A. (2004). "Refinement and Emergency versus Design and Predetermination". In: **Proceedings of IWES 2004** (Ueda K., Monostori L., Márkus A.), Budapest, pp. 13-18

[8] Lúčny, A. (2004). "Building complex systems with Agent-Space architecture". **Computing and Informatics**, Vol. 23, pp. 1001-1036