Computing and Informatics, Vol. 23, 2004, 1–36

BUILDING COMPLEX SYSTEMS WITH AGENT-SPACE ARCHITECTURE

Andrej Lúčny

Institute of Informatics, Faculty of Mathematics, Physics and Informatics
Comenius University, Mlynská Dolina, 841 42 Bratislava, Slovakia
e-mail: lucny@fmph.uniba.sk

Manuscript received 28 January 2003; revised 17 May 2004
Communicated by Jozef Kelemen

Abstract. Building complex systems requires a specific kind of modularity as well as incremental development. Here we introduce an architecture where basic modules are reactive agents and the data exchange among them is based on the so-called stigmergic communication through space. In this way we have connected ideas coming from the multi-agent systems and the coordination languages on the one side and ideas of the behavior-based systems on the other side. We demonstrate that this architecture manifests several interesting features which are useful for engineering of real-time systems and modeling of biological creatures or their parts. We advocate for so-called purely reactive agents which are stateless entities usually taken as too weak building blocks of systems. However, their features enable us to use a special method of incremental development (so-called subsumption method).

Keywords: (purely) reactive agent, space, stigmergic communication, incremental development, subsumption, multi-agent systems

1 INTRODUCTION

Over the years, artificial systems controlled by software have increased in complexity to approach the natural ones. As systems get more and more complex, their development becomes more and more difficult and therefore new techniques are needed. In this article, being inspired by mobile robotics – the first area, in which problems related to complexity appeared [5], we introduce one such technique for the complex software development.

Though the complex system is quite a vague concept, we can define it here as a system with such extensive behavior that we are not able to make its clear comprehensive description and design its detailed structure before starting the process of implementation. It means in practice that such a system must evolve from a much simpler basis. However, in our approach we do not suppose (in conformity with [4]) that this evolution must be based on an adaptive approach only (using techniques like artificial neural networks). On the contrary the evolution is enabled by such a system architecture which is opened for potentially endless modifications provided by a developer. This enables us to approach complex systems in a symbolic and algorithmic way (Fodor's ideas) rather than by principles like connectionisms.

Complex systems are typical mainly for a domain of artificial intelligence. Here we are trying to approximate behavior of living creatures which is not only extensive but also partially unknown and hidden. Moreover, if we trust (and we do trust) that intelligence does not reside in a presence of some special components (like inference mechanism of expert systems or genetic algorithm or adaptive mechanism of neural networks) then we can interprete intelligence as our subjective binary categorization of behavior complexity related to a particular problem. For being accepted as intelligent, every problem requires a system producing adequate behavior within a certain set of conditions. Thus we unintentionally define its own threshold of intelligence for every problem and we cannot overcome this threshold without a kind of architecture which provides sufficient scale of the behavior complexity. Therefore achieving complexity is necessary (though not sufficient) for achieving intelligence.

On the other hand, it is not clear why arbitrary architecture is not able to overstep any threshold of complexity. Theoretically, we can develop any kind of systems by coding in assembler. However, it is taken for granted that there are many natural limits such as capacity of human brains, communication skills, money issues, lack of time, probability of errors, ... etc. That is why every architecture has its limit of complexity which cannot be overcome. Nevertheless, some architectures can provide higher complexity than others. Unfortunately, we are not able to evaluate it in a theoretical manner, therefore we will examine their usefulness for building particular applications of complex systems. Considering today's technology, we focused on such applications, which are situated in a real or simulated world and operate in an interactive manner. In such systems, information is continually being sensed and behavior is continually being generated [13]. From those, we select the following application domains:

1. *engineering of real-time systems*, i.e. industrial systems like monitoring systems or process control systems
2. *modeling of biological systems or their parts*, e.g. modeling of artificial movement (mobile robots inspired by insects, various 2D and 3D simulators), models of mind which explain some psychological experiments (like [18] explains Piaget's experiments).

Therefore we will take into consideration two main criteria for evaluation of our architecture: its usefulness for real-time software engineering and its biological relevancy (usefulness for modeling of biological systems).

Our architecture is ideologically inherited from the domain of behavior-based systems (BBS), like subsumption architecture [4] and its later derivates. However, we do not express its structure by definition of a special behavioral module as is usual for BBS, but we use the modern terminology of multi-agent systems (MAS). MAS are a rapidly growing area which could become crucial for many domains of industry, e.g. for manufacturing systems. Unfortunately, the agent metaphor is too wide [17]. As a result, MAS are an umbrella domain. Its mainstream is represented by the so-called deliberative agents [9]. Our approach shares some features with the mainstream, mainly those which are based on the same kind of modularity. However, our approach belongs to a branch of the so-called reactive agents [10] which is inspired by R. Brooks. Anyway, terminology of MAS provide us more general framework than particular architectures originated in BBS.

2 SYSTEM STRUCTURE

Concerning the system structure, our architecture is based on a multi-agent system. Our system consists of a set of **reactive agents** which communicate indirectly through another entity called **space**. Agents cannot communicate directly and they are not able to refer to each other by an ID or a name, they can just read and write named data in the space.

2.1 Space

Space can contain potentially unlimited amount of named data buffers, called blocks. All the agents can write some data into a block and read them. The *read* and *write* operations are services provided by the space for agents. In that way an agent can leave a message in a block for another agent (Figure 1), but it cannot direct it to a particular agent. It is up to the agent whether the message is noticed or not. A block can be read and written by several agents, the appropriate coordination is intended to be provided by a designer. No agent can hide such message, any agent can read or even rewrite it. Sometimes such a message is called stigma for its simplicity and that is why this kind of data exchange is referred to as **stigmergic communication**.

Each block can be empty and does not have to be created – it is physically formed by the first *write* operation. Consequently, it is possible to read it even before it is written. These blocks are not intended to be queues, so the next *write* operation rewrites the data written during the previous *write* operation. *Read* operation does not take data from the block, reading does not change the space. Data contained in each block can have limited time validity or can be written forever. When validity of a block expires, the data disappears – such a block becomes empty – no matter whether the data has been read or not.

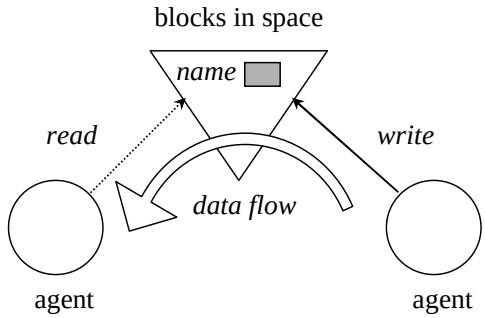

Fig. 1. A dataflow between two agents provided by space

Our architecture does not require any special form of data which is stored in the space, although a form could be defined for a particular application. The data can be compared to a knot on a handkerchief, the reader must know what they mean to understand. (So-called ontology problem is not addressed here.) Usually, the content of a block is a record, but it is read and written as a pure buffer of a specified size. It is even enabled that two agents read it with different size – therefore providing a modification, it is not suitable to remove an item from a record or to add an item on other place than at the end of the record.

There is an important detail that an agent can perform a sequence of *read* and *write* operations in space without interruption by the operations of other agents. This also enables us to associate some auxiliary operations with the fundamental ones, for example the specification of time validity with the *write* operation.

The *read* and *write* operations are the most important services provided by space, but not necessarily exclusive. An example of another fundamental operation is *delete* which provides deletion of a block. Another important operation is *freeze* which is useful for disabling other agents to change a block content during a specified period. *Freeze* is an auxiliary operation – it is associated with the *write* operation. It is dedicated to enable us to define priorities of agents which write to the same block. When an agent calls the *freeze* operation, it specifies a priority which is a positive real number. Then – during the period given by validity of the written blocks – *write* operations of a lower priority do not change the content of the block. (The *write* operations, which are not associated with *freeze* operations, are concerned to have a zero priority.)

In principle other services are possible as well. Because of physical limits, they are required mainly in real applications, but they are not inevitable from the theoretical point of view. The first of these services is the registration of triggers which provide a notification for an agent when a block is changed. This is particularly important when we wish the reading agent undertakes written data immediately. An agent can ask for this service during its initialization. It is possible to ask for a notification of any block here, regardless of whether it has been created or not.

When a block is changed, all agents which registered its notification are triggered (the exact meaning will be explained later).

The other possible services are the following: reading of many blocks whose names match with a pattern, reading all notified blocks which have not been read, turning a block into a queue, etc. Nevertheless, this concept is quite simple as within applications that we have developed, space provides from five up to ten services.

2.2 Reactive Agents

All the application codes are concentrated in relatively small and simple processes called reactive agents. A reactive agent is a process which, after its initialization, performs endless cycle *sense-select-act* (Figure 2). The cycle is still in operation at given frequency and there is a pause – *sleep* after each course of the cycle. In the *sense* phase, the agent reads data from the space. Then it selects what has to be changed to achieve a certain goal. Finally, in the *act* phase, it writes the calculated changes into space.

It is important to emphasize that the period of *sleep* can vary from agent to agent. However, there is a global watch which provides regularity of agent cycle timing (provided that the period of the real execution of the cycle is less than the period of sleep). On the other hand, the courses of cycles of individual agents are not synchronized or ordered.

Usually, the period of sleep is specified during the agent initialization. Exceptionally, the sleep can wait for a notification that some blocks have been changed, mainly when the period happens to be so low that it causes a live-lock (because of the physical limits). However, we prefer not to turn such an agent into an event-driven program – this agent is just woken up and it starts to observe not the changes, but the current state of space. We dislike to let it wait for a notification endlessly, we just want to wake it up from its sleep urgently.

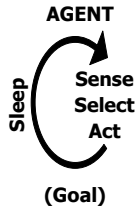

Fig. 2. Reactive agent

The presented entity is called an **agent**, because its activity does not depend on an activity of another entity and as such is proactive. It is not a consumer, but a generator of activity. Therefore, it can easily happen that during its activity it selects an idle action, as at the beginning of its activity it cannot know whether something in space is changed so much to act.

The adjective **reactive** is added to emphasize that the code providing the selection is quite simple. No planning is used here and the pursued goal is not represented in an explicit form. The code is just like a branched if-then-else command (a decision tree) and the goal is only implicitly inbuilt in particular selections. Therefore the response of such an agent is a pure reaction to the state of space and to its own internal state.

We can have a special case when the agent has no internal state, i.e. during the current cycle it cannot use any internal data from the previous cycle. Such stateless agent is called a **purely reactive agent**. Though its abilities are very poor when it is isolated, we will advocate later that it is the most progressive building block within our architecture.

2.3 Implementation

Implementation of our architecture can be founded on any kind of inter-process communication (IPC) in which we can implement client-server model[1]. In fact, space can be understood as a special kind of server and an agent as a more specific client.

We have used real-time OS QNX4 and we have implemented the architecture over the *Send-Receive-Reply* blocking message passing in C language. Space was implemented as a server which expects requests for services in *Receive* and responds by *Reply*. Agents were implemented as separated processes which expect a proxy triggered by timer or by space (in case of notification) and consecutively perform *read* and *write* operations by *Send*. These *Send* commands have been enwrapped into functions of communication library linked to each agent. The library contained functions as the following:

```
AgentRead(char *block_name, void *buffer, int size,
   struct block_status *status)
AgentWrite(char *block_name, void *buffer, int size,
   struct block_status *status)
AgentDelete(char *block_name, struct block_status *status)
AgentSetValidity(int seconds, int nanoseconds)
AgentFreeze(float priority)
AgentDefTrigger(char *block_mask, pid_t proxy)
```

A simultaneous execution of a sequence was provided by queuing of the requests on the agent's side and their common realization by calling `Agent(char *SpaceName)`. So the library was implemented as follows:

```
struct space_msg msgto, msgfrom;
```

[1] An alternative approach can be based on network programming (middleware) or component programming (run-time component). Here we prefer IPC, i.e. concurrent programming.

```
void AgentRead(... parameters ...) {
    struct space_request rq;
    ... fill rq with parameters ...
    ... add request rq into msgto ...
}

int Agent(...) {
    Send(space_id, &msgto, &msgfrom, sizeof(msgto), sizeof(msgfrom));
    ... fill parameter with returned values ...
}
```

Owing to technical reasons it was not forbidden to mix various function calls. It can be useful for instance to realize a lock (perception of a free item in space) in the following way:

```
lock = 0; locked = 1;
AgentRead("lock", &lock, sizeof(lock), NULL);
AgentWrite("lock", &locked, sizeof(locked), NULL);
Agent("SPACE");
if (lock == 0) printf("locked\n");
else printf("wait\n");
```

Consequently the code of agents had approximately the following form:

```
void main () {
    ... initialization ...
    for (;;) {
        Receive(proxy, 0, 0);                      // timer (or trigger)
        AgentRead("a", &a, sizeof(a), NULL); ...   // perception
        Agent("space");
        ... Compute b form a ...                   // selection
        AgentSetValidity(s, ns);                   // action
        AgentWrite("b", &b, sizeof(b), NULL); ...
        Agent("space");
    }
}
```

Finally, space looked like:

```
void main () {
    struct space_msg msgin, msgout;
    ... initialization ...
    for (;;) {
        pid = Receive(0, &msgin, sizeof(msgin));     // receive a request
        do {  // go through requests
            switch (msg.action) {
                case ACTION_READ:
                    ... compute msgout from msgin
```

```
                    and representation of space ...
                break;
            case ACTION_WRITE:
                ...
                break;
            ...
        }
    } while (...);
    Reply(pid, &msg, sizeof(msg));              // respond with data
  }
}
```

Since the QNX4 is not common outside the real-time community and it is possible to implement the architecture almost on any modern platform, we prefer not to discuss more details here. We intend just to point out that in comparison with the so-called pyramidal client-server architecture usually used for system development under QNX4, our architecture has many advantages – not only as non-blocking inter-process communication. Other advantages resulting from inner features of the architecture are discussed below.

This architecture is suitable mainly for single concurrent systems or distributed systems within the local area networks. Its usage for systems distributed in the wide area networks is limited by time latency – it depends on application domain if this approach is usable with today's hardware.

2.4 Transducers

When we are building a system operating in the real world, we need to provide an interaction between agents containing application code and devices. This can be done easily through blocks in space which correspond to input or output of a device. This correspondence is realized by an agent with an inbuilt specific knowledge about the device. Of course, the block contains only significant information, which depends only on the nature of the device, not on its particular type. Therefore it is similar to a shared library with normalized application interface, but it is not passive. Typically such an agent provides a stream of data regularly coming into space or it controls the device in a default way even when there is no requirement in space to do something with the device. Of course, agents can use any present resource of operating system, therefore the above mentioned information is related to such external devices like sensors and motors, and not to file opening on hard disk or similar standard devices.

Analogically, we can provide an interaction between the system and its user in such a way that the user will use a utility program to get or change the status of some blocks in space.

The blocks which serve for data exchange between space and devices or a user will be called **transducers**. For both kinds of transducers we will use the same graphic representation presented on the right side of Figure 3.

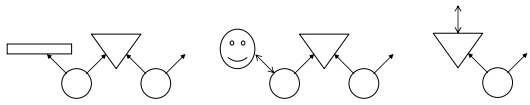

Fig. 3. Transducers

2.5 Name

Because of the structural character of the presented architecture we propose to call it **the Agent-Space Architecture**.

3 ADVANTAGES AND DISADVANTAGES

Now we will go through several issues related to the presented architecture. Of course, the architecture brings both advantages and disadvantages and it depends sometimes only on the point of view how a particular feature is evaluated. We will discuss and clarify typical features of our architecture and we will outline how to use them to be beneficial for software enginnering of real-time applications (our first criterion).

3.1 No Deadlocks

The first trivial profit of the agent-space architecture is its absolute resistance to deadlocks. In fact, this is a trivial feature, since there is no unit whose activity depends on another unit. From the inter-process communication point of view, no process can be blocked endlessly, because we established and used exclusively non-blocking message passing. It is not possible to design agents in such a way that some of them get stuck. However, this is true only if we use a notification in such a way as described above, i.e. the notification must not be used as exclusive trigger of agent's activity – only to make it active urgently. So if we follow this rule then no deadlocks appear.

Of course, in this way it is guaranteed only that there will be a certain activity in the system, not that the activity will be reasonable. So a robot controlled by such a system can still get stuck – just its building modules can not.

3.2 Normalization

In principle, it is possible to normalize everything everywhere; however, norms differ in quality. Here we have normalized the communication interface of the space and partially the form of the agent codes. The main feature of the norm is its simplicity, since the space provides only few services. On the other hand, the norm is general enough and independent from application domain.

Although the communicated data are not normalized (principally they could be – for example if the semi-structured data are used), the norm is dedicated to usage protection from the so-called enwrapping functions. Usually these functions enwrap data transfer by composition of the transferred structure from application-oriented parameters and backward decomposition of the replied data. In our approach, we dislike such functions as `GetDistance(float distance)` calling internally `AgentRead()` and we prefer to use `AgentRead("Distance", &distance, sizeof(distance), NULL)` instead. Regarding the terminology of object oriented programming, we prefer to transfer objects instead of calling their methods remotely – only methods called remotely are the methods of space.

3.3 Data Flow

Unlike the both traditional ways of data exchange, where a producer sends data to consumers or consumers request data from the producer (client-server), we have applied another method of data flow (Figure 4). In our approach, neither the producer nor the consumer knows who is on the other side. They know the block only – the place where they should put data or where they could get data. This allows better distribution of codes related to application domain into individual processes. From this point of view, space is similar to a database containing operational data only. From the other point of view, we can understand it as a news-server among human agents applied to the realm of the software agents.

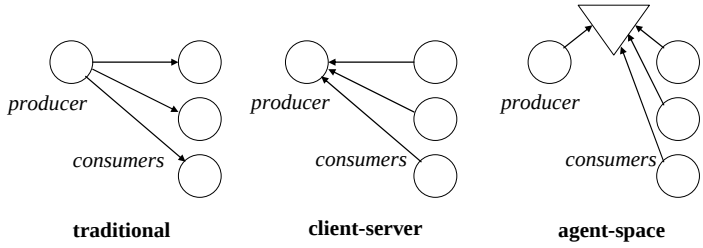

Fig. 4. Data flow

3.4 Recovery from Errors

For the sake of the type of data flow, it is possible to restart any agent whenever it is required, without any impact on other agents. This feature is crucial for system recovery when an agent crashes. In this situation we can simply start it again. Agents communicating with the restarted agent do not need to refresh any id to recover data flow. Even if the data stored in the space has no relation to the internal state of the restarted agent, the cooperating agents will not discover at all that something special has happened.

Space is independent from the application domain, therefore its stability and reliability can be on the same level as stability and reliability of the operating system. Space is not intended to be restarted or to have errors. It must be simply reliable, stable and powerful enough. As space is the same for all applications we are able to focus on it and provide its reliability.

Another consequence of the type of data flow is that system can be easily started. Any agent can be started in spite of the fact that other agents, which it communicates with, have not started yet. From the communication point of view, the order is not important.

From the implementation point of view, for the system recovery and initialization, we need of course a special process, which launches agents and provides restart if some of them exit in failure. Unlike the watchdog, the restart here is local only.

3.5 Ability to Configure

There are several reasons why our approach supports the ability to configure. First of all, it enables agents to share configuration parameters split into many individual blocks in space. In this way we are able to avoid copying of the parameters from configuration sources among many agents and the necessity to keep the copies equal.

Secondly, we can configure such a system by launching different agents which write data to the same blocks – this is similar to the power of normalized application interfaces. Typical example of this method is a set of agents which are drivers of different input devices which produce the same data: just one driver will be launched – that one which responds to a particular configuration parameter.

Finally, we are able to propagate a change of a configuration parameter through the system during an operation. As we have already mentioned, under usual conditions we can restart any agent whenever we want. In this way we can change a parameter and restart all agents which read the parameter during its initialization. Another way how to provide that is to read the changing parameter during each cycle of the involved agents. In this way we can spread a change even without restarts, of course at the cost of redundancy while the parameter is stable.

3.6 Hot Backup

As a block can be read by several agents, it can also be written by several of them. Although, at the first glance, it does not seem to be reasonable, as it results in double production of the same data; this trick allows us to implement hot backups into our system in a very natural way.

Imagine two different methods calculating the same result from different inputs which are not reliable (some of them can be missing). Any agent calling one of these methods consequently writes no data into space when the method fails. Then it can happen that the resulting block is written by both the agents (one result is overwritten by another one) or only by one of them or even by none of them. But since blocks do not have nature of queues, from the consumer's point of view the

value of the block is present or missing. (In the case of failure of both the methods, there is no danger that the consumer finds an older result in the block as the block validity is limited.)

On the other hand, we can have two methods producing different results which are written in the same block. One of the methods can produce a very fast, but not very accurate estimation and the second an exact but slow calculation. In this case, consumers, which have a shorter sleep period, process at first the estimation, and then revise their results using the exact value. Consumers with a longer sleep period can process the correct value only.

When we employ a backup of data which are transmitted to a consumer's block via several different paths, we have to concern that transmitted data could be delivered in an invalid order. In such a case, we can attach a timestamp to the transmitted data and ensure that the space overwrites older data only. Thus the consumer receives the data in a consistent order and even triggers are generated (if they are used) in such an order as if we had the data flow without any backup.

3.7 Decentralization

Although the space is a bottleneck of the whole system, there is no scheduling program which specifies whether an agent has to act and when. Agents are completely autonomous. They are not interdependent. Their cooperation is not defined explicitly in their code, it must emerge from interaction among agents.

Of course, for acting reasonably an agent can have a need of data produced by other agents. However, (normally) there is no special agent which has an exclusive position, no "control agents", no "upper agents" and so on.

Therefore we are able to declare that our architecture is as suitable for development of decentralized systems as it is improper for centralized ones. Decentralization is simply natural for our approach with all resulting advantages and disadvantages. The main disadvantage is that we lose the perfect control over the system. On the other hand, we are able to handle most of details locally without any overall plan.

It is important that though the space is a bottleneck, it does not depend on application domains. Therefore it is as acceptable as a database or a message queue.

3.8 Data Inconsistency

A typical example of losing control over the system can be observed when there is a change of a block that has both direct and indirect influence on many other blocks. When a value of such a block is stable, the dependent values are mutually consistent. Suddenly, when the block is changed, consistency of these values is violated during a short period while the change is spread through the system as a wave. During this period, even an undesirable combination of data, like previous and current, can be processed together and inappropriate results can appear. However, after some time, the blocks become stable again and their mutual consistency is renewed.

The main reason for the inconsistency period is that the direction of propagation is reverse. The source of a change neither propagates it, nor sends a notification to its target. Instead, the target waits for its next loop when it detects the change occurred and performs one step of its propagation. For the sake of this reverse propagation, we almost do not have to take care of propagation during development of agents. So the disadvantageous inconsistency is interconnected to another faculty from which we make profit. Moreover, it is acceptable anyway, if the inconsistent values are continuous. In fact, calculating e.g. the average from `10` and `10`, and then from `12` and `12`, we have no harm if we catch the inconsistent state `10` and `12` – we get a continuous sequence of reasonable averages `10, 11, 12`, although the `11` is undesirable. However, when we process logical values like `ball-in-the-left-hand` and `no-ball-in-the-right-hand` and then `no-ball-in-the-left-hand` and `ball-in-the-right-hand` we could get a wrong result `ball-in-both-hands`, although only for a moment. Principally, it is the art of design to avoid such situations: we can prevent such cases, or we can design all agents in such a way that they wait for stable input, or we can even let inconsistency as it is and consider its impact.

3.9 Redundancy

Another tax which we have to pay to employ the reversed propagation is redundacy. Basicaly, instead of a single propagation of a change from a source to a target when the change appears, the target has to check regularly whether the change has appeared in the source or not, i.e. potentially a lot of time per one change.

Therefore such a system is active regardless whether a change has occurred or not and consecutively the system performance is decreased by redundancy. Potentially, it can execute plenty of instructions to produce none behavior.

3.10 Data Loss

Data loss is even worse than redundancy. It can appear easily when an agent produces an output faster than other agents are able to undertake it. A designer has to pay attention to this possibility. Unfortunately, when we use the only block, it is sometimes really difficult to carry out such operations as for instance money calculations. However, we can design any data flow without such losses when we use a different block for each instance of data. Of course, we need to use a generic name for all these blocks to recognize them from others. Further, each such name has to contain a timestamp or another identification.

3.11 Real Time Support – Implicit Sampling

Redundancy and data loss are usually recognized as undesirable properties. However, we can refuse this opinion when our system operates in real time. Real time

means for our architecture that courses through an agent loops are distributed over time exactly according to the sleep periods – only a limited difference can occur.

Consequently, the occurring redundancy is acceptable, since the system has to be prepared to propagate a change at any time. Therefore there must be constantly enough free capacity to provide the propagation in the limited period. Thus the redundancy is just exploitation of the capacity which must be available anyway.

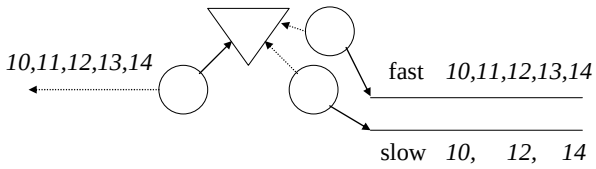

Fig. 5. An implicit sampling

Similarly, the data loss can be even positive within real-time systems. If one part happens to produce values faster than the other part can process, these values will be sampled and no overload will appear. Moreover, faster consumers will sample data at higher frequency than slower ones and each consumer will sample data at the highest frequency possible (Figure 5). This sampling is an inner ability of the architecture and as such it does not require any special care. A designer has just to know that such data can be sampled. Owing to this implicit sampling we are perfectly protected against live-lock.

Generally, we can say that the operation in real time and our architecture support each other.

3.12 Ability to Modify

Most of the already mentioned features of our architecture is related to the system structure and they are manifested during its operation. However, for building complex systems, the phase of the system development is even more important. Therefore we intended to design our architecture in the way that the developer's job is simplified as much as possible – even at expenses caused by operational disadvantages like redundancy. We focused predominantly on improvement of an ability to modify, because of the inevitable incremental development. Modifications are usually related only to further improvement of a developed system, but principally also each phase of the incremental development can be recognized as a modification. Though the main idea of these modifications will be discussed later, some other issues will be mentioned here, namely those which require no additional conditions.

As for the extension of the system, it can be easily provided by adding some new agents and interconnecting them to the already implemented ones by reading the already implemented blocks (Figure 6).

Further, concerning an ability to configure, we have also an opportunity to provide easily such modifications which extend the configuration parameters of the

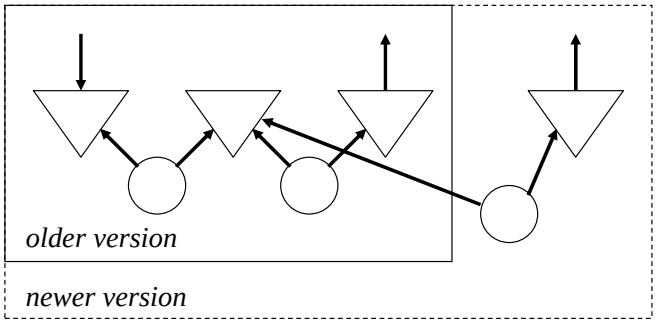

Fig. 6. Modification by adding new agents

system. In fact, we can add new agents which alternate those already implemented. During the initialization of the system, only such an agent will be launched from a set of the alternations, which corresponds to configuration parameters.

Functionality change of the already implemented parts of the system is more difficult. It is tolerable when this change can be provided without changes of blocks, since the code of agents is quite simple. On the other hand, we would like to change functionality even without changing codes of agents. This possibility will be discussed later.

Sometimes a modification requires to extend a block by additional information. Of course, this can be realized by adding a new block with this information. However, in our approach also an original block can be extended, if we add the information at the end of the stored data buffer (Figure 7). This is enabled by an extra parameter of the read operation which specifies the expected size of the read data.

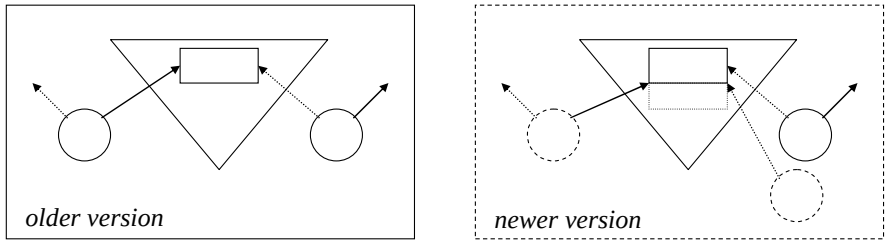

Fig. 7. Modification by block extension

Of course, this strategy works only when the data in space are recognized as ordinary buffers. Since within our approach this is not an indispensable condition, this strategy is not universal. However, for other kinds of data a similar strategy can be found. For example, when the data are XML documents, the older agent can use an older version of XSD and the newer agent a newer version. Thus both extract that part of data which is relevant to them.

3.13 Example: A Simple Monitoring System

In Figure 8, we demonstrate a simple monitoring system.

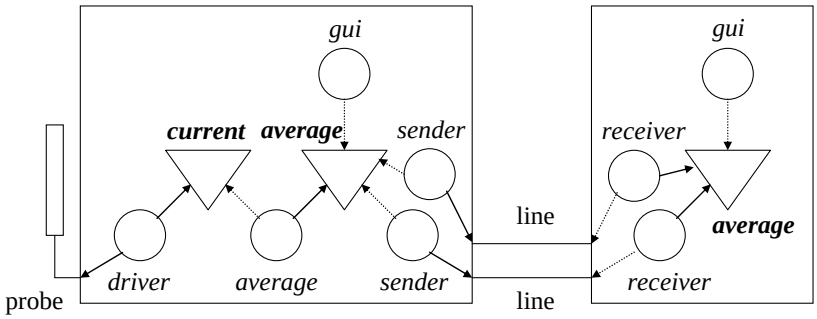

Fig. 8. Example of simple monitoring system

There are two places where a floating average of a measured quantity should be displayed. We put a probe on the first place and interconnected the two places by two lines (to increase reliability) and we developed five agents (some of them are used more times) to provide separated activities:

- *driver* agent provides measurement from the particular probe
- *average* agent calculates an average of coming values
- *gui* agent displays a particular value
- *sender* agent provides transmission of a value to a line
- *receiver* agent provides reception of a value from a line.

These agents have been organized to cooperate through the two following blocks of space:

- block **current** contains a current measured value
- block **average** contains an average which is intended to be displayed

Our architecture enables us to decompose the system into independent modules, of which only the *driver* and *gui* are tied to application domain. We are able to use some agents without any change at both the places and even for other systems. Connection backup between the two places is achieved by running the same pair of agents – *sender* and *receiver* – twice; thus it is free of charge. If the connection is too slow, the block **average** at the second place is automatically sampled (according to the block **average** at the first place) at the highest possible frequency – thus real-time is supported. Though the system is small, it is quite smart and opened for further modifications. Its ability to be modified can be demonstrated on the following examples:

- It is easy to use another type of probe – we can provide that by an exchange of the agent *driver* for another agent which produces the same output into block **current**.
- We can implement a double or triple probe when we use more *driver* agents which write into the same block (**current**). When a probe gets a failure, the *driver* agent does not write any value to the space. Thus while at least one probe is working, the other agent *average* receives a value. It has no knowledge whether there are one, two or three operational probes, it works in the same manner regardless of whether there is a backup of probe or not.
- We can easily add new places, even a network with circles can be implemented. Such distribution will be finite due to timestamps. When some connections are broken, but the target is accessible, automatically the distributed values use a suitable route.
- If we wish to implement an additional quantity to the system, we will establish new blocks **current2** and **average2**. We will use a new probe and its driver agent, we will have to modify the *gui* agent and we will re-use the agent *average* to produce **average2** (we will use the same program with different parameters). Further we will extend parameters of *sender* to distribute also **average2**.
- If we want to display different data at different places, we can implement another agent *gui2* and we will add a new parameter (type of display) of a scheduler which launches the system.
- When we need to adjust a parameter during the course of the system, e.g. we would like to stop distribution of **average2**, we have two possibilities. Firstly, we can adjust parameters of the agent *sender* and restart it. As an alternative, we can put the parameters into a block in the space and re-implement the *sender* to read this parameters regularly, during each loop.
- We can implement entrance of subsidiary values (e.g. for the case when a user finds that a probe is measuring incorrect values and wishes to enter a correct value manually) in such a way that we add an agent which writes subsidiary values into the same block as the probe driver writes measured values. However, it has to use the *freeze* operation to disable the driver to rewrite a subsidiary value by an incorrect measured value. Thus we achieve that the driver agent has no impact on the system if there is no need to deal with it.

4 BIOLOGICAL RELEVANCY

In this section we will discuss the above mentioned features of our architecture according to their similarity with the living systems. These features are interesting mainly for modeling living systems (our second criterion).

4.1 Structure

It is typical for the living systems that there is a general principle which describes how all modules are interconnected and organized, but the purposes of the modules are realized in specific ways. We could say that the modules are similar by syntax but different by semantics. We have implemented something similar in our architecture by the level of normalization. In fact, only those aspects are normalized which are independent form the application purpose: the communication with space and the code structure of agents. However, there is no norm for knowledge representation, no norm for employed algorithms, no norm for coordination, etc. We think this is the right level of normalization for following Minsky's idea [19] that it is not a goal to find the best kind of knowledge representation, but a system manipulating various representations all together.

A very nice feature of our approach (which is unfortunately very rare within multi-agent architectures) is its ability to express an agent as a multi-agent system of lower description level (Figure 9). This faculty is crucial for building (or modeling) a hierarchical system which contains several levels. This trick is based on substitution of an agent acting in space by such a group of agents, which cooperate through another space, whereas at least one of them provides mutual mapping of some blocks between the two spaces. Within this substitution, such group of the lower-level agents acts in the higher-level space as one higher-level agent. (Of course, technically, we don't need two spaces here, just two separated groups of blocks in one space. For this purpose, it is recommended to have an order in block names – e.g. different prefixes.)

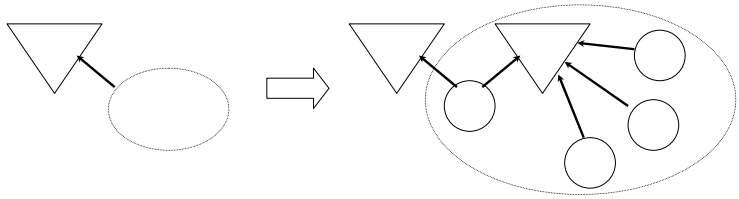

Fig. 9. Realization of hierarchy

So within our architecture, hierarchy can be modeled by encapsulation. Of course, the system is developed in the expanded form, but it can represent the hierarchical form thanks to the above mentioned trick. Moreover, such a hierarchy is recursive, so there is no limit for the number of levels and no qualitative differences among them.

4.2 Activity

Now, let us look at the deadlock feature. In fact, it is typical for living systems that when a critical situation appears, in which they are not able to behave reasonably,

they do not stop their activity and they act in a non-reasonable way. They do not get stuck, they do something anyway. None of their modules exits or throws an exception, it keeps trying to do something. This is another inner faculty of our architecture. On the other hand, this nice feature is at the cost of our control. Adding more and more agents, the result of their cooperation is more and more difficult to be understood.

Of course, although the activity of agents is permanent, it does not mean that blocks (mainly the output blocks) are constantly changing. Some agents cannot even perform *write* operations at all. However normally, having nothing to do, the system tends to the regular dispersion of the *write* operations. This regularity is disturbed when a non-redundant activity appears. A change of outside conditions is propagated through the system as a wave. This wave can reach a block along several different paths at different time, so the first reaction of the system can be very quick and it can be gradually improved by better reactions whose calculations require more time (Figure 10).

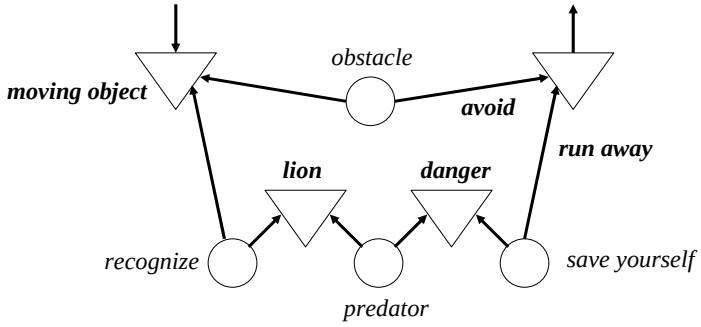

Fig. 10. An example of a fast but not sound reaction later improved: when a lion is met, the system at first tries to avoid it as an obstacle, only later the lion is recognized as a dangerous predator and the system tries to escape

4.3 Fluidity

Another typical feature of our architecture, which we can also observe on living systems, can be likened to fluidity. In fact, when different agents write to the same blocks and they compete whose value will be chosen for the further processing, then the behavior resulting from these interactions can be fluky (Figure 11). This quality can be used as a design principle in such a way that a designer adjusts system (by setting up periods of *sleep* for agents which share same blocks) in the way that it produces almost stable behavior which sometimes fluctuates to an alternative. If such an alternative is profitable, it can be strengthened, otherwise the system

returns to the stable operation. In such a way, the fluidity enables the system to move away from a wrong track.

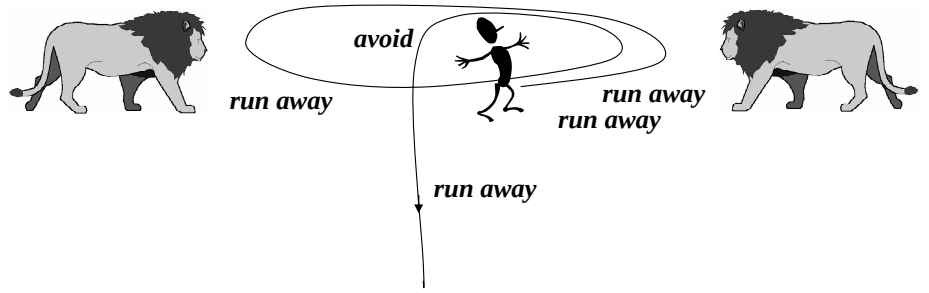

Fig. 11. An example of fluidity: The system in Figure 10 is not able to escape from the presented situation as it is attacked by two lions. In this case the *run away* command turns it still back – from one lion to the other. However, the *obstacle* agent, which tends to turn the system to the left, can help here although it is not responsible for getting away from lions. This is possible because both agents are still active. An agent cannot be passive, only its activity can have no effect

If a designer allows, the fluidity can also cause a short-term appearance of inadequate global behavior. In such a case we can at least observe richer behavior than inevitable on the system, even if it is sometimes completely useless. This is interesting especially when we are modeling a living system and our solution based on fluidity has a simpler structure than alternative solutions without fluidity. In nature we can find a lot of examples of strange behavior which is apparently ineffective or even harmful. In many such cases the behavior can be modeled and explained as a competition among more modules operating in parallel. We could say that nature prefers here more effective implementation at the cost of less effective behavior.

4.4 Example: Modeling of Mind Failure

The discussed kind of activity and fluidity is possible due to sharing of one block in space by more producers. This feature is usually considered to be questionable, because it permits solutions which are not completely correct. Now, we would like to present that nature sometimes prefers such solutions. In fact, natural systems work fine under ordinary conditions, but sometimes we can establish such special conditions that they produce a failure. Such a failure should not appear if the systems were constructed under the laws of logic. However, the failures can be modeled by agent-space architecture and explained by the presence of fluidity. We demonstrate it on the following example, a well-known joke from Internet[2]:

`Read the sentence and count the number of letter F's there are:`

[2] `http://www.grassyknoll.homestead.com/5307WorldFinishedFiles.html`

```
FINISHED FILES ARE THE RE-
SULT OF YEARS OF SCIENTIF-
IC STUDY COMBINED WITH THE
EXPERIENCE OF YEARS
```

When this task is performed by a group of people, approximately one quarter of them answers that there are three F's, the next quarter answers four, the next one five and the last one six. The correct answer is six, but those F's which are part of the word OF are almost invisible. Usually it is explained by a phonetic argument: the invisible F's are read as V's. However, when we let to count the letters V, nobody finds them. Moreover, we have tested that knowledge of English has no impact on the results. (On the other had, e.g. age has a significant impact: young children, mainly those who are just learning to read, find all the F's. Also positioning has an impact.) In our opinion, the invisible F's are omitted because there is a special module (a recognizer of OF) in our mind which moves our focus by two letters. This recognizer acts along with another module which process letters sequentially, i.e. moves the focus by one letter. Thus sometimes the module which is counting letters catches F and sometimes not. (We do not know why it is just OF that produces such an effect. We assume that each word could cause the same, but it is more probable that it is caused by short words without a separate meaning.) The relevant part of mind is modeled by our architecture in Figure 12. There is fluidity in the block **eye-mover**: *reading-timer* and *OF-recognizer* are competing whether to move focus by one or by two letters. They regularly write their commands into the block and it is given only randomly which value is undertaken for processing. Out of an unknown reason, the frequency of recognizers related to longer words is low and the frequency of the *OF-recognizer* is high, even close to frequency of *reading-timer*. Therefore recognizers of other words have no impact and the *OF-recognizer* causes that the F is lost with the probability of approximately 50 %. Thus the probability of all the four answers is about the same. (Besides these competing agents, we have also self-correcting mechanisms which provide – for example – that our vigilance has an impact. Finally, there are also memorizers in our mind which cause that our mind adapts to this test and produces the correct result the next time. Therefore it has no meaning to repeat such a test.)

By this experiment we tried to demonstrate that nature employs also potentially incorrect building blocks and rectifies them by patches. For the modeling of cooperation among these competitive modules, an appropriate architecture is inevitable. The agent-space architecture can serve this purpose well.

4.5 Summary

In this section we have outlined that our architecture could serve well for modeling or simulation of living systems. Mainly it can respond to those situations in which profit or failure are explained by sharing of communication resources by several

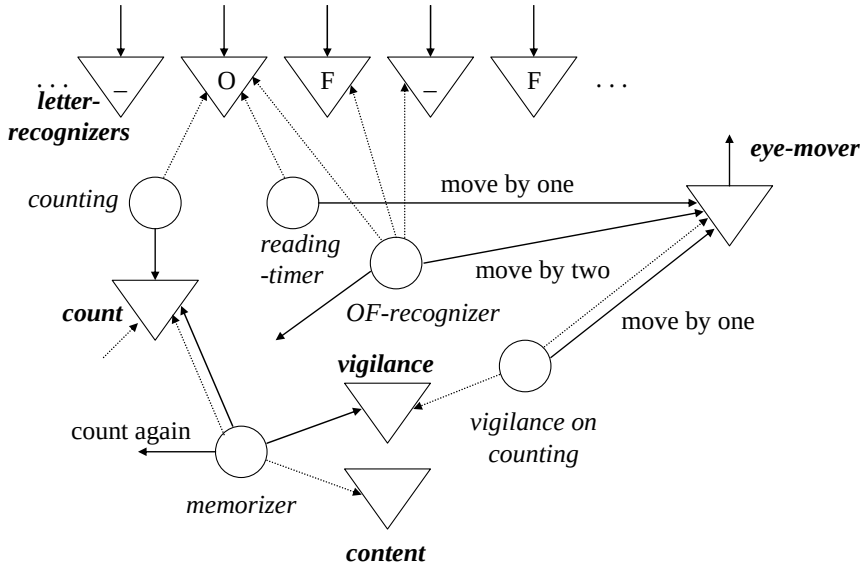

Fig. 12. Modeling of failure

different modules. Thus we observe profit when the communicated agents cooperate or we observe failure when they compete. The most interesting situation happens when the agents do the both, i.e. they mainly cooperate, but (because of simplicity of implementation) the cooperation is not perfect.

We have used our architecture to simulate behavior of insects ([15]) and behavior of humans during one psychological experiment ([16]). On the basis of this experience we can affirm that the introduced architecture is biologically relevant. We regard this quality as a very important evaluation parameter, though it is based only on our belief that similarity to the living systems is crucial for building complex systems in general. Anyway, regardless the biological oriented features, we have successfully used the same architecture for building systems in practice (namely for real-time monitoring systems).

5 INCREMENTAL DEVELOPMENT BY SUBSUMPTION

Although we have noticed that a system created on the basis of our architecture tends to do something rather that nothing, we have still not mentioned one issue which is much more important: How to provide that our system will do what we have intended? Though in most of our experiments we simply relied on our art and experience, we were also looking for a method, which could guarantee validity of results. We have found the key inspiration to this problem in the so-called subsumption architecture created by R. Brooks ([4]).

5.1 Subsumption

It is taken for granted that any larger system must be developed incrementally. It is preferred to design the whole structure of the system in details by top-down decomposition, then to implement its core and sequentially add further parts. We should use such an order that we are able to integrate each increment with the already developed parts and to test them all together. Usually this method works well, although the quality of validation is sometimes decreased by employment of trivial replacements of those parts which have not yet been developed, but we need them to provide integration. However, building really large and complex system, we can just pretend that we have designed all details before implementation. In such a case each validation shows that design (and consecutively the already developed structures) must be modified. These modifications are handled in such a way as if we designed the system from scratch. Of course, in reality we do not throw away all what we have already done and rather we modify than design again. Thus, using this method the start is very rigid but the finish can be – because of many modifications – pure art.

At the end of the eighties R. Books invented the so-called subsumption architecture, in which he overcame the lack of ability to design all details before implementation. He postulated modifications as the main principle of design and proposed to design at the beginning just the structure of the system behavior (practically the test criteria for each stage of the incremental development), i.e. what to do (and how to test that it is done) instead of how to do. The stages of development were organized to activities (the so-called decomposition by activity). After test procedures for each stage had been prepared, development started with design of the first stage and its implementation. Then the system was tested and modified until it passed all the test criteria for the first stage. Then the structure for the second activity was designed, implemented, tested and modified until it passed the criteria for both the first and the second activity (also the first stage had to be tested again because it could be damaged by addition of the second stage). This procedure was repeated through all the stages (Figure 13).

Of course, before putting this strategy into operation, some particular problems have to be resolved. R. Brooks focused on the following ones:

1. How to find out from a higher stage what lower stages do?
2. How to employ (re-use) on a higher stage what we have already implemented on lower stages?
3. How to block activity of a lower stage when neither we can employ the stage nor we can let it operate as it is?

It is obvious that all these problems originate from our capability to influence lower stages from higher ones. In fact, this is a problem, since the lower stages were not designed to be interfaced by some higher stages – the higher stages were not concerned during its development at all. This problem is typical for bottom-up

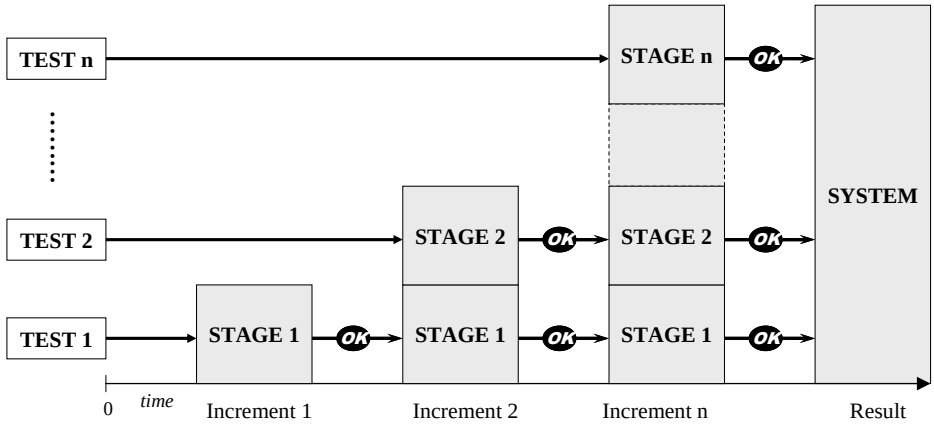

Fig. 13. Incremental development by subsumption

development in general. Principally, it can be overcome; however, the solution depends on the type of structural units and on the data communication among them. R. Brooks has used so-called augmented finite state automata which sent messages via fixed communication wires, therefore it was possible to solve the above mentioned problems by the following tricks (Figure 14):

Wiretapping: All wires can be monitored from any higher stage and the message transmitted through the monitored wire is duplicated and received also on the higher stage.

Suppression: Any wire can be equipped by a special device called suppressor. It receives a message from a higher stage and provides that this message rewrites any message coming from the stage where the suppressor is placed.

Inhibition: Any wire can be equipped by a special device called inhibitor. It receives a message from a higher stage and provides that any message coming from the stage, where the inhibitor is placed, is blocked. (Principally inhibition is suppression by empty message).

Because of technical reasons, the duration of each suppression and inhibition was limited by a parameter of the suppressor and the inhibitor devices.

Within our architecture we have different but similar structural units. Principally, our blocks in space resemble the communication wires and our reactive agents resemble the augmented finite state automata. Therefore we can handle the above mentioned problems in the following way (Figure 15):

Wiretapping: a block designed for a lower stage is read by an agent designed for a higher stage.

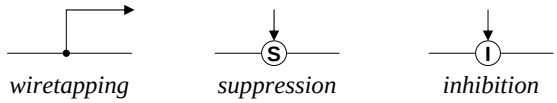

Fig. 14. Means of subsumption: wiretapping, suppression and inhibition

Suppression: an agent designed for a higher stage writes to a block designed for a lower stage.

Inhibition: an agent designed for a higher stage deletes (or writes a default value into) a block designed for a lower stage.

If we accept a competition between the stages, the duration of suppression and inhibition can be realized by validity of blocks. Otherwise, a strict priority can be provided by the *freeze* operation, which disables block changes for a given period.

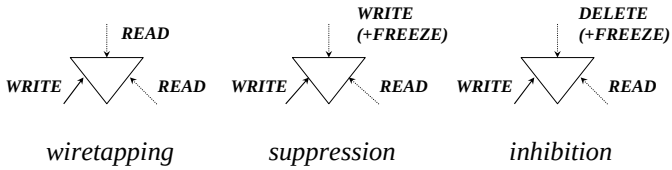

Fig. 15. Subsumption in the Agent-Space Architecture: wiretapping, suppression and inhibition

Thus, we are able to influence any lower stage from a higher stage in spite of the fact that the lower stages are not designed to be influenced. In this way, main disadvantages of the bottom-up development are overcome.

5.2 Example: Reimplementation of ALLEN

ALLEN is a mobile robot developed under Brooks' subsumption architecture [2]. Now we will use this robot to demonstrate our architecture. We will copy the original ideas one by one, in the way that in the end it will be possible to compare our solution with the original one – except small differences caused by potential misunderstanding of the original solution, because of lack of details. We emphasize that all ideas of the robot construction are taken from the original work and we hereby only show how they can be expressed within our architecture.

ALLEN's task is quite simple: to traverse natural space inside a building which contains walls, doors, obstacles and moving objects. It is equipped with sonar sensors which measure distance of the nearest object in a certain direction and wheels which provide forward and backward motion and rotation.

At first, we have to specify a set of tests, which should be passed at the end of development. In this way we define the behavior, which we intend to develop. Although the order of the tested activities is not important for the global behavior, it is crucial – because of reusability – for the development process. We have to start with tests focused on the simplest – atomic – activity and continue with activities which are more and more difficult. Therefore, in our example the set of tests could be as follows:

1. A. When the robot moves to a wall, it must not bump into it.

   B. When it moves to an obstacle, it must avoid it. (Because of the strict logic involved, we have to affirm that no trivial fulfilment of this test is allowed – and the robot must move.) (AVOID activity)
2. The robot must not tend to remain on a periodic path. (WANDER activity)
3. The robot must not remain inside a local region for a longer period, but it must tend to explore the whole space. (EXPLORE activity)

Now we can start with implementation. Because of lack of space, we present just the final result on Figure 16. However, the system is implemented incrementally, from left to right, in such an order as described below.

- **Forward** is a block, whose content is interpreted as forward or backward motion or stop. If it is empty, forward motion is taken as default.
- *Sonar* is an agent which processes input from sonar sensors stored in the *sonar* block. It produces a **map** block which contains distances of the nearest objects for each sector. (The robot has no compass and its vicinity is split into sectors according to its current heading.)
- *Collide* checks such a part of a **map** block which represents space in front of the robot. If it detects a danger of collision, it puts a command to **forward** to provide backward movement.

  – At this point the robot normally moves straightforward and when it encounters a wall it moves back to be far enough. Thus the test 1A is passed.

- *Feelforce* evaluates where the most urgent danger of collision is. It writes its direction to **force**.
- *Runaway* proposes an opposite direction and writes it into **heading**.
- *Turn* controls rotation of wheels so that they followed the proposed heading. Sometimes it needs also backward motion, therefore it writes also to **forward**. There is a feedback between rotation caused by *turn* and content of **map** and consecutively it has an influence on **heading**. Therefore more deviation from a potential collision causes less requirements to rotate.

  – At this point the test 1B is passed. The AVOID activity is implemented. The robot moves straightforward until the danger of collision occurs. Then it

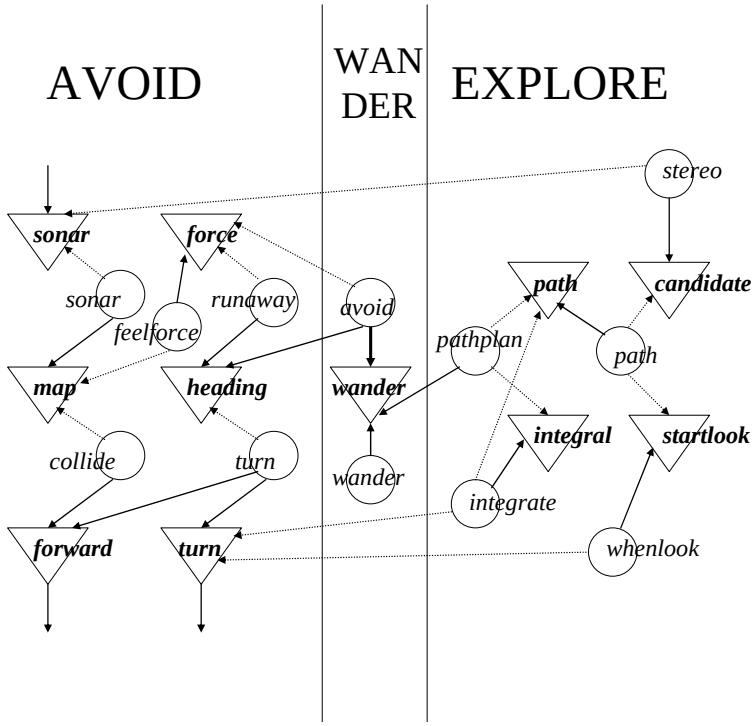

Fig. 16. The structure of ALLEN expressed under the agent-space architecture

avoids the obstacle or the wall and follows straightforward movement again. As a side effect, it is able to avoid also a moving object. Under certain circumstances, it can happen that the robot gets to periodic path, but this situation is acceptable at this stage of development.

– Now we would like to provide that motion is occasionally changed in spite of lack of potential collisions.

- *Wander* is an agent with a larger sleeping period which makes a proposal (stored in **wander**) containing another heading of the robot as the current one.
- *Avoid* takes this proposal, checks in **force** for potential collisions, and in case of no danger, it overwrites **heading**. Thus, *turn* acts as if the direction of wandering was a safety direction for avoiding a potential collision. The reading from *force* is a case of wiretapping, and the writing into **heading** is a case of suppression. So the implementation of the WANDER activity employs implementation of the AVOID activity.

  - Now we have passed the test 2. Since *avoid* checks **force** it is not even inevitable to repeat the test 1. The WANDER activity is implemented.
  - We are not able to predict the robot movement now, but still it can jumble in a small region. Now we will implement the next stage to provide movement among such regions.

- *Whenlook* detects from robot movement whether it jumbles (it wiretapes rotation commands). When jumbling is detected, *whenlook* issues the **startlook** command to find a suitable direction for transfer to another region.
- *Stereo* looks (permanently) for a suitable direction of relocation. When such a direction is detected, it is written to **candidate**.
- *Path* decides to move to another region following direction **path**. **Path** is written and then frozen for a longer period, so further re-decisions, which appear during this period, have no effect. However, **path** is not an absolute direction, therefore for using it we need to know the deviation of the current robot heading from its heading when the decision was made.
- *Integrate* calculates this deviation into **integral**.
- *Pathplan* calculates from the decision and the deviation such a rotation that keeps the robot on the chosen track. Under the subsumption principle, this rotation is understood by lower stages as if it were generated by *wander*. Therefore the *pathplan* writes the rotation into **wander** at much higher frequency than *wander*, in the way that it overwhelms its impact on the robot motion. (There is a little difference from the original solution, where the value written from the EXPLORE stage had simply higher priority and the robot was not consequently able to wander at all during its transfer to another region. In our architecture the suppression has never been so perfect. On the other hand, because of this difference, our solution resembles more living systems than the original one.)

  - All the tests are passed at this point. The development is finished.

In this example we have omitted some details like sleeping periods, validity of blocks, duration of freezing operations, etc. In our opinion this level of description is adequate.

Though here we only express an example from [4] in a different way, there is one remarkable detail which illustrates in what way our architecture is suitable for the description of such systems. Regarding the original paper, there is one module called *Status* which belongs to the lowest stage AVOID, but its only purpose is to provide information to higher levels (namely EXPLORE). Strictly considering, it is a violation of incremental development, since there is no reason to implement it during building the lowest stage. However, within our architecture there is no necessity to have such a module. Thus, the system description is a little bit smarter.

Finally, unlike subsumption architecture, our architecture is applicable not only for the domain of mobile robotics. It is more universal. The original example is

considered to be confined to the domain of mobile robotics only. However this transformed version can be directly used as an example for building any real-time system in general.

5.3 Command Fusion

Up to this moment we have shown that our architecture provides at least the same opportunities as subsumption architecture. However, can it offer any advantages? Firstly, we will compare our approach to [20] which is also inspired by subsumption architecture. Secondly, we will argue for our architecture to have more general concept which is even suitable for such modeling as described in [18].

Rosenblatt and Payton criticized subsumption architecture and proposed their own derivate, which used system description based on neural networks (from this point of view, our approach is parallel: we also use a different description, but based on multi-agent systems). There was the so-called command fusion problem in the center of their criticism. When compared to our approach, we can say that they preferred the joining of two data written by two agents into the same block, instead of rewriting. On the other hand, Brooks preferred rewriting but according to strictly defined priority of agents. Our approach is closer to Brooks than to Rosenblatt and Payton. However, sometimes we would also like to allow a lower-level agent to rewrite what a higher-level agent has written. We would like to let them compete, each one with a different frequency of its *write* operation. Of course, we are also able to set up priorities if we use the *freeze* operation. However – unlike subsumption architecture – priorities are taken as something exceptional in our approach. In the approach of Rosenblatt and Payton, two commands can be joined together, but the mechanism of the joining (e.g. addition) cannot differ within the system. This is a very strong limitation since we often need to use different strategies of command fusion at different places. Moreover, a universal mechanism of command fusion forces us to use just one kind of communicated data (like real numbers form -1 to 1). However, within a real application, we sometimes prefer typed data, other times data records, marshaled objects, XML documents or other kind of data. Therefore, when a necessity of a command fusion appears, we prefer the strategy analogical to the subsumption architecture: we add a new agent which is implementing a particular strategy of the command fusion (Figure 17). It seems to us that regarding of the command fusion, the original Brooks's proposal suits for practical engineering more than some later derivates like [20].

5.4 Disruption of Levels of Competence

Further problem, which is usually discussed in connection with derivates of subsumption architecture, is the so-called disruption of levels of competence. Within our architecture the problem resides in the possibility that a higher-stage agent is able to overwrite an important value in a block which belongs to a lower stage. Of course, this problem is not very acute, since the higher stage should be aware of what

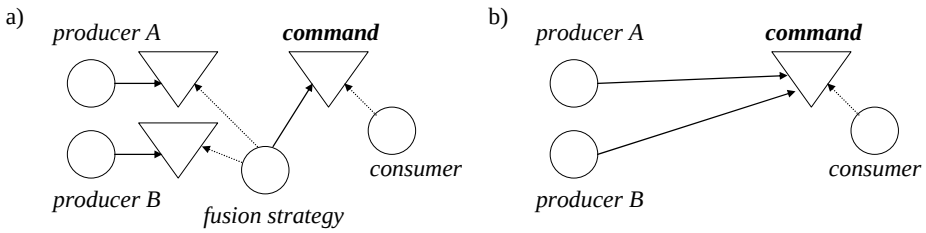

Fig. 17. Command fusion: a) the universal solution, b) the simplest but potentially defective solution which seems to occur in nature

it is overwriting. However, sometimes it can be more comfortable to let the lower stage be active regardless the higher stage tries to suppress it. This mechanism is not present within subsumption architecture, but it is present in some later derivates and it is also possible within our architecture. It can be implemented avoiding the *freeze* operation and applying variable frequency of *write* operations performed by an agent at the lower stage. Usually the frequency is low, and when the higher stage performs a suppression, the impact of the agent is minimal. However, under certain conditions, the frequency can be significantly higher and the higher level is not able to suppress it.

5.5 Advocating Stateless Agents

Using subsumption for system design in practice, it is usually possible to apply its principles easily; however, sometimes it can be quite difficult. There is a significant problem called inaccessibility of internal state, which was superbly recognized by [20]. It resides in the internal state of agents, which we are neither able to monitor nor to change. When the data from a lower stage, which need to be manipulated, are located in space, we can carry out the manipulation without any problem. Otherwise, we would have to modify agents at the lower stage to reveal the data. However, we prefer not to perform this kind of modification.

We can treat this problem perfectly by application of a very simple rule: **all agents must be stateless, i.e. purely reactive**. Then all the data, which agents remember from one course of their loop to the next one, must be stored in space. It will be more difficult to implement agents under this restriction, but later it will be much easier to apply the subsumption principle. As for purely reactive agents, nothing important can be missing in space, though during implementation of the original stage, some data can be put to space only to be remembered through time rather than being communicated among different agents.

At the first glance, respecting the rule, a developer loses certain comfort so that he could obtain a strange kind of development. But, in fact, s/he gets not only an ability to realize subsumption but also very significant profit in general. The additional profit can be demonstrated on the following cases.

When we design a block, which contains a configuration parameter, and later we find out that it should vary during operation, it must be turned to a variable. Usually we implement a parameter in the way that it is read at the beginning and then the system does not take care of its changes. Therefore we have to restart the whole system or its significant part to force the system to reflect that the parameter is changed. If we want to avoid such a restart, we have to modify all agents, which have remembered the parameter in their internal state: we have to implement reload of the changed parameter when a notification about its change is received. However, with the purely reactive agents, absolutely nothing has to be modified in this case. With this kind of agents, it is not simply possible to use a parameter without its regular reload during each agent loop. Moreover, because of the form of the purely reactive agents, this regular reload needs no special code and thus it does not cause additional work for a developer. Of course, while the parameter is stable, its reloading is redundant. However, when we need to turn the parameter to a variable and to change it, it is enough just to rewrite its value in space. Of course, there is still huge redundancy, since we reload the parameter regularly for a lot of times so that we handle just several real changes. However, in general, we prefer to consume power of machine to save developer's work. In our opinion, this general rule is crucial for building complex systems.

However, the ability to configure is not the only attribute, whose quality increases when purely reactive agents are employed. Recovery from errors, which is based on restarts, gets higher quality as well. We have already mentioned that a defective agent can be restarted without any influence on other agents' abilities to communicate with it. However, it does not mean that the other agents do not need to reestablish cooperation. Now, having purely reactive agents, even the cooperation continues (when impact of defection is over) from the same point where it was interrupted. The system becomes more robust. Analogically, the same quality change can be applied on the system initialization. With purely reactive agents the system can be started disregarding the order of agents and it has no impact on both their communication and cooperation.

However, in spite of possible restarts and no expenses for restoring cooperation, it can still happen that wrong content appears in space, in consequence of which the system tends to behave incorrectly forever. In this case, developers usually search for a failure cause and try to modify agents to avoid this state. The subsumption principle can offer another procedure to handle this problem. We can add a new agent which is able to detect such defective content in space and transform the content to normal. Since the added agent acts just when the defective state occurs, we have no need to repeat all the tests. Thus we can prefer to add a new agent instead of changing the already developed ones. This possibility is very convenient, since the ability to modify in general resides in the capacity of an architecture to change functionality of the system without re-implementation of the already implemented modules. Within our architecture we are able to achieve this objective by subsumption. Of course, employment of subsumption is possible only if we do not insist on full understanding of how the resulting behavior of the system is generated from its

structure. A system developed in this way seems like a dress with many patches, but – and this is important – without any hole.

6 COMPARISON WITH ORIGINS

Design of the agent-space architecture is inspired by several different areas:

Real-time systems and a pyramidal Client-Server architecture. Originally we designed our architecture to overcome problems with mutual data exchange between two servers within a pyramidal client-server architecture. (This architecture is usually used for building real-time applications. It is even recommended by the producer of QNX4, the leading real-time OS.) In fact, the space is a special kind of server and agents are its clients. Our architecture is very beneficial here: its solution of deadlock leads to much simpler application code and the ability to configure and to recover from errors is much better. The only disadvantage of our approach is the lack of a reliable synchronization mechanism which leads in practice to utilization of triggers. In comparison with alternatives of the pyramidal client-server architecture like a message queue, we provide better structure of both the communicated data and the application code and much smarter information sharing by separate modules. The communication via channels of a message queue is smart when the communication framework contains one producer and one consumer, but otherwise it makes troubles. In our architecture, this disadvantage has been overcome.

Coordination languages. The way of indirect communication through the space is similar to language LINDA [7] used for parallel programming (with respect to its modern version called Java Space from the Sun Java Jini package [22]). From here, we have adopted the name *space*. However, in our approach:

- It is not possible to match values of the data stored in the space, just their names (the matching of values is not necessary). Data can be represented in various ways, therefore we call them blocks, not tuples. The variability of representation (data buffers, typed data, marshaled objects, XML documents, ...) is very important for engineering of real applications.
- It is possible to read a block before it is written, i.e. no instruction for block creation is required – this leads to much simpler codes of agents. This feature is crucial when there are more producers and consumers sharing a block. As a result, it is crucial also for incremental development.
- All communication is non-blocking, synchronization is provided by triggers only or it is replaced completely by regular refresh. Thus the synchronization is not perfect but a better operation in real time is enabled – due to implicit sampling.

Multi-agent systems. This research domain provides a type of modularity which enables us to structure codes into entities which are always active and thus suitable for creating decentralized systems. (We call them reactive agents – because

of the way how actions are chosen. Though, from the point of their activity they are not reactive but proactive.) The mainstream of this area is represented by so-called collaborative (deliberative) agents [9]. Though some ideas, mainly those related to the kind of modularity, are valid also for our approach [23], we are much closer to a special branch of MAS which deals with reactive agents [10]. The code of such agents is quite simple and not related to traditional artificial intelligence. This approach is more useful for industrial applications, since it directly extends the usual methods from this area: it establishes a new organization of application codes, but their nature remains unchanged. Here, the employed agents have no special components inspired by the so called good old-fashioned artificial intelligence, they are implemented by ordinary means. Considering MAS terminology, we can say that our agents are rather "software" than "intelligent" agents. Also significance of the bottom up development for software agents was recognized in particular application domains like active user interfaces [11]. Further specialty of our approach resides in the emphasis on indirect communication among agents: we do not permit direct communication at all. This strategy is exceptional in MAS but sometimes appears [21]. We consider it very important when we develop a complex system.

Subsumption Architecture and its later derivates [4]. This area provides us with a suitable type of development. However, under the original subsumption architecture higher levels were always prior to lower ones and their influences were strictly ordered according to the position of suppressors and inhibitors. This feature was also inherited by later derivates of subsumption architecture known as behavior-based systems [1]. On the contrary, our approach allows disruption of levels of competence, even it allows to let them compete. This feature can provide a simpler structure of the resulting solution and it is useful for modeling of living systems. For example, such global behavior as modeled by our architecture in Figure 12 or Figure 11 cannot be modeled by any architecture with a strict priority of atomic behaviors. Unlike subsumption architecture and all its known derivates, our architecture can provide a reference by value. We can realize it by a reference block containing a name of another block which should be manipulated. Then the agent which has to provide the manipulation acts according to the content of the reference block. Thus, our agent can vary the block which is manipulated, while nothing similar is possible within the subsumption architecture with wires. This reference is useful for some applications. Namely, it is inevitable for modeling of mind structure called pronomes [18]. Brook's or Payton's wires are proper to model "move apple to table", but – unlike our blocks – make troubles when we model "move X to Y".

Concerning the above-mentioned domains we can say about our architecture that it is a sumbsumption architecture applied for software development and expressed in term of multi-agent system, or we can say that it is a multi-agent system which uses bottom-up incremental development, or it is a new kind of LINDA-like space, etc. However, regardless many sources of inspiration we insist that we have not

only collected ideas from various sources but we created their reasonable fusion and powerful extension. Therefore we decided to give it a specific and shorter name: **the agent-space architecture**.

7 CONCLUSION

In this article we have presented an architecture based on a multi-agent system built from (purely) reactive agents which communicate indirectly through space using its services as read a message from a block, write a message to a block (with given validity), erase a message from a block, freeze content of a block (for a given period) and potentially further additional operations. As for the preferred type of development we recommended the bottom-up incremental development based on the decomposition by activity, where the coordination among activities was provided by the subsumption principle. We advocated for purely reactive agents as the most suitable entities for our approach.

We were trying to introduce into building complex systems several ideas carrying various kinds of biological motivation. The communication used among agents resembles exchange of chemicals, agents can be likened to cells, subsumption to the so-called evolution tinkering (bricolage).

We tried to join and extend these ideas and we applied them for the development of both industrial applications and models of living systems. It is interesting that our approach does not differ from usual software systems too much. From the technical point of view, just some restrictions, specializations and extensions are used. We were not able to eliminate developers from the development process, but we wish to change their thinking. From the philosophical point of view, our approach participates on the revolution started by R. Brooks fifteen years ago, which has not yet finished – in our opinion.

Acknowledgements

I would like to express my gratitude to Jozef Kelemen who introduced this research area to me, gave me very strong philosophical foundations and encouraged me to apply them both in theory and practice.

REFERENCES

[1] Arkin, R.—Balch, T.: Cooperative multiagent robotic systems. In: AI-based Mobile Robotics: Case Studies of Successful Robot Systems (Kortenkamp, D., Bonasso, R., Murphy, R. eds.), MIT Press, Cambridge, Mass., 1998.

[2] Brooks, R. A.: A Robust Layered Control System for a Mobile Robot. IEEE Journal of Robotics and Automation, RA-2, 1986, pp. 14–23.

[3] Brooks, R. A.: A Robots that Walks: Emergent Behaviors from a Carefully Evolved Network. Neural Computation 1:2, Summer, 1989.

[4] Brooks, R. A.: Intelligence without representation. Artificial Intelligence 47, 1991, pp. 139–159.

[5] Brooks, R. A.: Cambrian Intelligence. The MIT Press, Cambridge, Mass., 1999.

[6] Bryson, J.: Intelligence by Design: Principles of Modularity and Coordination for Engineering Complex Adaptive Agents. AITR 2002–003. AI Lab, MIT, Cambridge, Mass., 2002.

[7] Carriero, N.—Gelernter, D.: Linda in Context. Communications of the ACM, Vol. 32, 1989, No. 4, pp. 444–458.

[8] Doran, J.: Distributed AI and its Applications. In: Advanced Topics in Artificial Intelligence (Mařík V., Štepánková O., Trappl R., eds.) Springer-Verlag, Berlin, 1992.

[9] Jennings, N.: On agent-based software engineering. Artificial Intelligence 117, 2000, pp. 277–296.

[10] Ferber, J.: Multi-Agent Systems – An Introduction to Distributed Artificial Intelligence. Addison-Wesley, Harlow, 1999.

[11] Kautz, H.—Selman, B.—Coen, M.: Bottom-up Design of Software Agents. Communications of the ACM, Vol. 37, 1994, No. 7, pp. 143–147.

[12] Kelemen, J.: Multiagent Symbol Systems and Behavior-Based Robots. Applied Artificial Intelligence, Vol. 7, 1993, pp. 419–432.

[13] Kelemen, J.: From Statistics to Emergence – Exercises in Systems Modularity. In: Multi-Agent Systems and Applications, Luck, M., Mařík, V., Štepánková, O., Trappl, R., (eds.), pp. 281–300, Springer, Berlin, 2001.

[14] Knight, K.: Are Many Reactive Agents Better Than a Few Deliberative Ones? In: Proc. IJCAI '93, Chambery, 1993, pp. 132–137.

[15] Lúčny, A.: Searching for a Qualitative Difference. In: Cognition and Artificial Life (Kelemen J., Kvasnička V., Pospíchal J., eds.), Silesian University, Opava, 2002, pp. 167–176.

[16] Lúčny, A.: Modeling of Cognition Failure by Multi-agent System using Stigmergic Communication. In: Cognition and Artificial Life II. (Kelemen, J., Kvasnička, V., eds.), Silesian University, Opava, 2002, pp. 119–132.

[17] Nwana, H.: Software agents: An overview. Knowledge Engineering Review, Vol. 11, 1996, pp. 1–40.

[18] Minsky, M.: The Society of Mind. Simon&Schuster, New York, 1986.

[19] Minsky, M.: Emotion machine. In Proc. EMCSR'2000 (Trappl R., ed.), Vienna, 2000.

[20] Rosenblatt, J. K.—Payton, D. W.: A Fine-Grained Alternative to the Subsumption Architecture for Mobile Robot Control. In: Proceedings of the IEEE/INNS International Joint Conference on Neural Networks, Washington DC, Vol. 2, 1989, pp. 317–324.

[21] Valckenaers, P.—Van Brussel, H.—Kollingbaum, M.—Bochmann, O.: Multi-Agent Coordination and Control Using Stigmery Applied. In: Multi-Agent Systems and Applications (Michael Luck, Vladimir Mařík, Olga Štepánková, Robert Trappl, eds.), EASSS, Prague, 2001, pp. 317–334.

[22] Waldo, J: Mobile Code, Coordination and Changing Networks. CONCOORD, Lipari, 2001.

[23] Wooldridge, M.—Jennings, N.: Pitfalls of Agent-Oriented Development. In Proceedings of the 2nd International Conference on Autonomous Agents (Agents'98), Minneapolis/St. Paul, 1998.

Andrej Lúčny received his M. Sc. degree in artificial intelligence from Faculty of Mathematics, Physics and Informatics of Comenius University in Bratislava in 1994. He is a system development manager in MicroStep-MIS (in Bratislava), where he deals with development of monitoring systems. His research area is the so-called reactive branch of multi-agent systems, which he also teaches at Comenius University.