Intelligence Integration in Distributed Knowledge Management

Dariusz Król
Wrocław University of Technology, Poland

Ngoc Thanh Nguyen
Wrocław University of Technology, Poland
Chapter IX
The Concept of Autonomy in Distributed Computation and Multi-Agent Systems

Mariusz Nowostawski
University of Otago, New Zealand

ABSTRACT

The concept of autonomy is one of the central concepts in distributed computational systems, and in multi-agent systems in particular. With diverse implications in philosophy, social sciences and the theory of computation, autonomy is a rather complicated and somewhat vague notion. Most researchers do not discuss the details of this concept, but rather assume a general, common-sense understanding of autonomy in the context of computational multi-agent systems. In this chapter, we will review the existing definitions and formalisms related to the notion of autonomy. We re-introduce two concepts: relative autonomy and absolute autonomy. We argue that even though the concept of absolute autonomy does not make sense in computational settings, it is useful if treated as an assumed property of computational units. For example, the concept of autonomous agents facilitates more flexible and robust architectures. We adopt and discuss a new formalism based on results from the study of massively parallel multi-agent systems in the context of Evolvable Virtual Machines. We also present the architecture for building such architectures based on our multi-agent system KEA, where we use an extended notion of dynamic and flexibly linking. We augment our work with theoretical results from chemical abstract machine algebra for concurrent and asynchronous information processing systems. We argue that for open distributed systems, entities must be connected by multiple computational dependencies and a system as a whole must be subjected to influence from external sources. However, the exact linkages are not directly known to the computational entities themselves. This provides a useful notion and the necessary means to establish an autonomy in such open distributed systems.


INTRODUCTION

This work concentrates on the general notion of autonomy in multi-agent systems. We will initially define an abstract concept of relative and absolute autonomy in the context of a computational agent. We think that the concept of autonomy must be always linked with the context and with the reference to what a given notion is applied to. Autonomy means different things to various researchers and it seems necessary to provide appropriate context and qualification of the term. Based on the notion of relative autonomy, we review some of the existing multi-agent systems. We will then discuss the objectives of the research community and the motivations regarding the concept of autonomy of a given computational unit (an agent) in the context of open multi-agent systems, adaptability and complexity growth. We argue that, to build an open and adaptable multi-agent system, agents must be subjected to constant external influences. These influences must (possibly indirectly) affect and control a given agent’s behaviour, and therefore negate the generally accepted requirement of agent’s absolute autonomy. Based on our results with experimental Evolvable Virtual Machines (EVM) (Nowostawski, Epiny, & Purvis, 2005a) framework, we draw conclusions that computational agents can never be truly autonomous or else the applicability of multi-agent systems in solving complex problems in an open environment would be constrained or even impossible. That means that restrictions on autonomy imposed by the multi-agent system designer are not only based on the pragmatic needs to limit and manage general complexity of the system. These restrictions come directly from an inherent property of the dynamics of the MAS as a distributed asynchronous computational system.

To demonstrate and discuss the issues related to autonomy, we build an experimental framework called Evolvable Virtual Machines (EVM). This framework has been used for modeling and analysis of metacomputational architectures, metalearning, self-organisation and adaptive computing. We will present results related to a contemporary model of computation for massively parallel open-ended evolutionary computations based on EVM (Nowostawski, Purvis, & Cranefield, 2004; Nowostawski, Epiney, & Purvis, 2005b; Nowostawski et al., 2005a). The model has been used to investigate properties of asynchronously communicating agents in a massively parallel multi-agent system. In this context, we discuss the concept of computational complexity, evolutionary learning and adaptability. We will show that with our computational evolutionary system a constant flux of external information is necessary to provide an open-ended increase of complexity of generated (discovered) computational programs. Our results suggest that any closed (or fully autonomous) collection of computational agents would be limited in their ability to learn and adapt to new circumstances. We claim that the notion of autonomy should be revisited and used in a clearly specified context. Computational agents should be subjected to direct or indirect external influences to allow continuous learning and adaptation by the system as a whole.

AUTONOMY IN MAS

Autonomy, from the Greek Auto-nomos (auto meaning self, and nomos meaning law), refers to an entity that gives oneself its own laws. In other words, autonomy means self-governance, and freedom from external influence or authority.

Generally, in multi-agent systems there are two basic attempts and formalisations of the concept of autonomy: internal and external views. The internal notion applies the above definition of autonomy to the agent itself, and specifies a set of principles or architectural constraints that are claimed for an autonomous operation of a given agent. The external view takes a different approach. It does not prescribe anything about internals of the agent or agent architecture itself.
It is, rather, an assumption that other agents are autonomous in an abstract sense and cannot be controlled/influenced directly. Agent’s behaviour cannot be imposed by any other agent; hence the interactions and agents collaboration must take into account various aspects of the assumed participants’ autonomy. We discuss briefly these two notions below.

The general notion of autonomy is invariant of the usual architectural or behavioural interpretations. In the next subsections, we review proposals for the definition from the internal and the external point of view.

**Internal Autonomy**

From a simple engineering perspective, the concept of autonomy has been used as one of the distinguishing features between traditional object-oriented and agent-driven systems. See, for example, discussion in Franklin and Graesser (1996) or Castelfranchi (1995). It is important to note that the notion of autonomy in MAS is often confused with the notion of automatic or independent operation. We want to stress that autonomy does not collapse to a mere independent operation. In complex software systems, it is a simple truism that many complex inter-dependencies and influences must exist between various computational units. However, there is always an element of choice. Indeterminacy is essential, from the external observer point of view, to be able to talk of autonomous computing. As an example of internal view of autonomy, consider the work of Luck and d’Inverno (1995), who have postulated that agent’s motivation and the ability to create own goals is essential for autonomy. Using the Z specification language, they described a three-tiered hierarchy comprising objects, agents and autonomous agents where agents are viewed as objects with goals, and autonomous agents are agents with motivations. The ability to create goals according to some internal hidden and changeable agenda/motives is, according to their classification, essential for achieving true autonomy.

**External Autonomy**

Compared to internal autonomy, we can turn the roles around. Instead of concentrating on our own agent and its autonomy, we can insist on the assumption that all entities and agents that our software agent interacts with are autonomous in the abstract sense. How this is achieved, or if it is possible at all, is not our primary concern. What is important is the fact that no fixed assumptions can be made regarding the interactions, agents, goals delegation, motives, environment, and so forth. The research community is somewhat divided into two independent groups. One follows a strict internal view of autonomy, and proposes ways to enhance and promote autonomy in various agent architectures. The other group moved away from the initial strict internal requirements on agents’ autonomy, toward more open, distributed systems that are driven by interactions, dialogues, negotiations and collaborations of multiple individual participants, that are assumed autonomous from the external point of view. The role of autonomy for individual agents became an external assumption, rather than architectural requirement. The best discussion on this is presented in the work of Weigand and Dignum (2003). In their work, they have argued that architectural requirements of autonomy on agents are not as important as the expectations of autonomy on behalf of other agents. The agents that a given software agent interacts with must be assumed to be autonomous. Agents must be prepared to deal with other agents’ autonomy, and participate and collaborate with supposedly autonomous participants. This somewhat inverts the original requirements from those that support autonomy directly through elaborated architectures, into those that support features that work with autonomous agents.
Concept of Autonomy in Distributed Computation and Multi-Agent Systems

COMPUTATIONAL AUTONOMY IN MAS

Most researchers base the definition of autonomy on two primitives: self-governance and independence (e.g., Gouaich, 2003; Carabelea, Boissier, & Florea, 2003). Self-governance refers exclusively to the internals of the agent and its architecture. As we pointed out, this is not necessary in general discussion or in practical agent-oriented software engineering directly. Both notions, however, seem relevant when trying to formalise the concept of autonomy. One of the attempts in providing comprehensive definition is provided in Carabelea et al. (2003):

An agent A is autonomous with respect to B for p in the context P, if, in context P, its behaviour regarding p is not imposed by B.

The symbols used are: p represents a property, A and B represent the active entities, agents, and P represents the context. The property p in the above definition relates to the object of autonomy, and emphasis is placed on the relational nature of the concept of autonomy. There are, however, two main problems with the above definition. The first problem lies in the fact that multiple vague concepts are being used: context, property (or autonomy object) p and the notion of imposed. The precise and formal meaning of these terms in the above definition is not clear. Nevertheless, the above definition is useful and conveys the common-sense understanding of the concept of autonomy.

Formal Definition

To make the above definition less ambiguous, we propose to base the definition on a formal notion of computation. Let us assume computation C to mean the universal Turing machine transformation of input data from the input tape into output data on an output tape (we assume here a two-tape setup, with read-only input tape and write-only output tape). For more details and formal introduction to Turing-machine computational models, see for example, Lynch and Tuttle (1989) or Hopcroft and Ullman (1979). We will denote computation from input X into output Y as: X → Y. Let us assume data D to be a particular mapping of symbols into an input tape for the universal Turing machine. Let us assume that a computational agent A has access to a particular collection of data sources Di ∈ E, where E stands for environment, or context. In other words, agent A is capable of performing universal Turing machine computation on a set of data accessible from its environment E. The data can be represented as a sequence of symbols from a particular alphabet, for example, [0,1] as in the original work of Turing (1936, 1937). Without any loss of generality, let us assume the following properties:

- Data decomposition: ∃Di = O, ∀D = Di + Dj;
- Data composition: ∀Di,Dj + Dk = Di : Dj − Dk and Dk − Dk = Dj ; and
- Computational composition, ∀Di,Dj → Di+j and Dj+i → Dj+2i : Di → Di+j.

Data composition and decomposition simply captures the fact that data can be combined or split, without any loss of information. The computational composition ensures that computations do not have any side effects. Note that data can be read from or written to by various agents, and there is no distinction for input or output data. Only during the actual computation can a single data source be used as input or output (exclusive or).

Agent A is not autonomous in respect to agent B in the context Es, and Agent B is said to control agent A, if:

∀Di ∈ Es, ∃Dj → Dk : Dl → Dm. (1)
If no such agent $B$ exists, then we say that Agent $A$ is relatively autonomous in context $E_s$:

$$\exists D_i \in E_s : \forall D_i \rightarrow D_i, \forall D_y \rightarrow D_k, D_k \neq D_i.$$  

(2)

The agent $A$ is absolutely autonomous in the context $E_s$, if:

$$\exists D_i \in E_s : \forall D_i \rightarrow D_i, \forall D_y \rightarrow D_k, D_i = D_k.$$  

(3)

**Discussion of the Definition**

The above equations provide the following intuitive interpretations.

1. If Agent $A$ uses a particular subset of its environment $E_s \subseteq E$ with data sources $D_i \in E_s$ to perform its computation $C_{E_s}$, and there exists an agent $B$ that can output into all of $D_i$ sources, we say that agent $A$ is not autonomous in respect to agent $B$ in the context $E_s$. Agent $B$ is said to control agent $A$.

2. If no such agent $B$ exists, then we say that Agent $A$ is relatively autonomous in context $E_s$.

3. If there is no set of agents that can collectively output to all of the $D_i \in E_s$, then we say that agent $A$ is absolutely autonomous in the context of $E_s$.

Based on the above definition, we propose the following general autonomy classes in MAS. The three general classes below are often informally discussed in MAS literature, and these are now easy to define formally.

- **User autonomy.** Agent $A$ is said to be autonomous in respect to a user, if the user does not provide all the data inputs that control agent $A$. In such a case, users cannot impose agent’s behaviour directly; hence, we talk about agent’s relative autonomy in respect to the user.

- **Interactions autonomy (social autonomy).** Agent $A$ is autonomous socially, if it not only takes its inputs from other agents through interactions, but uses other sources of input at the same time. These sources are not bound to social interactions (e.g., user input). This means that agents cannot simply impose any goals or behaviour directly on other agents, because interactions are not enough to “drive” agent’s computations.

- **Organisational autonomy (norm autonomy).** Organisational and institutional norms modeled as data sources cannot be used to impose behaviour of agents directly. Agents use various data-sources that influence their behaviour and computational choices.

Some authors, in particular Carabelea et al. (2003), postulate also a notion of environmental autonomy. In our definition of computational agents, environment encompasses all the possible input data sources for a given agent: user input, other agents, static data, norms, and any other. Therefore, there is no possibility of an agent performing any other computational mapping than $E_{input} \rightarrow E_{output}$. Agent is, by definition, just a computational function from the input environment, to the output environment. The concept of environmental autonomy, in our setup, does not make sense. To discuss environmental autonomy, we would need to establish a partitioning of $E$ into subenvironments, one exclusively called environment, and other subsets labeled differently. We feel that partitioning of $E$ into such disjoint classes is questionable in a general sense, although it might be useful for certain aspects of MAS, namely user interactions, social interactions and organisational interactions. If we model a closed system, where all data sources are in some way dependent upon agent’s interactions and computations, then each single agent cannot be absolutely autonomous. To
have a meaningful concept of absolute autonomy we have to deal with open systems, where some of the data sources are beyond the scope of the MAS itself. In that case, we talk about a stream of randomness (or indeterminacy) that comes from outside of the system itself.

**INDETERMINACY AS AUTONOMY**

Our definition of autonomy as presented above rests entirely on the formal and intuitive notions of indeterminacy. Let us consider a case of two simple homoeostatic processes: one performed by a thermostat, and one performed by a bacterium. We intuitively feel that there is some difference between these two in respect to how autonomous they are. In the case of the thermostat, even though it operates completely automatically and independently and we do not know or cannot predict exactly when it will switch from state to state, the degrees of freedom are quite limited. A thermostat is usually embedded in a well-insulated environment, where the temperature reacts almost exclusively to the heater/cooler system controlled by the thermostat itself. In the case of bacterium, even though the performed functions are sometimes as simple as those of the thermostat, the actual degrees of freedom seem to be larger. This is mostly due to the fact that in the case of a thermostat, the environment is almost exclusively controlled by the thermostat itself (the thermostat can make the ambient temperature to go up or down). The environment, to a certain extent, is simple and reactive. In the case of bacterium, there is no direct control over the environment as such. The interactions with the environment are of different types. Bacterium must deal continuously in a highly unpredictable environment. We will not argue if there is any categorical distinction between these two autonomy classes. We just want to point out, that the main distinguishing feature from autonomous and nonautonomous process lies in the indeterminacy and predictability of the environment. If there is a process, that is entirely deterministic and predictable from a given observer’s point of view, then we say that there is no autonomy within that process. The process is simply determined as a function of its environment. If the process is not entirely predictable, then we talk about autonomy, and about a choice (the process can choose one or the other trajectory for its evolution).

Let us consider a multi-agent system within a formalism of Chemical Abstract Machine (cham) (Berry & Boudol, 1989). Cham has been successfully used as a modeling formalism for other process calculi and process algebras, most notably for Milner’s CCS (Milner, 1989), and Nicola’s and Hennessy’s TCCS (Nicola & Hennessy, 1987). It is possible to model any asynchronous computational system within cham formalism, and therefore some observations within cham can be extended to any other process calculi. In cham the states of a machine are modeled as solutions consisting of data-structures floating and interacting in an abstract space. These data-structures can be of any type: primitive, such as numbers and strings, complex objects, or agents. Assuming that the data-structures are individual agents, and the interactions are equivalent to reactions in cham, we can talk about two aspects in respect to autonomy (and indeterminacy):

- Interactions are random. They are not preordered or prespecified by the system design, and
- Reaction rules may or may not be followed by the individual agents. Note that in the original cham formalism all the reaction rules must be strictly followed by the system.

Now, let us consider a system (example inspired from Banâtre et al., 1988), consisting of $n$ agents named $2 \ldots n+1$ and a reaction rule (interaction) between agents, such as if $A_i, A_j$, where $i, j \in [2, n + 1]$, then $A_i \text{annihilates itself}$. That means that...
if two agents meet, and one of the agents is a multiple of another, the multiple will annihilate itself. From the initial solution of all $n$ agents, after some time, there will be only agents named with prime numbers left. This is assuming both, autonomy in the interaction choices and autonomy in the adoption of the general annihilation rule.

The above example demonstrates that in some circumstances, global coherent behaviour can be obtained in systems where autonomy is present on some of the underlying levels of abstraction. However, this is not always the case with all the systems. In some systems, autonomy must be restricted for the system to achieve a desirable stable point. For example, in the case of cham it is not easy to advise autonomous rules that would lead the system to calculate factorial. We will discuss this in more detail in the context of our EVM model. In the next section, we will briefly introduce the notion of autonomy in our multi-agent system KEA (Nowostawski, Purvis, & Cranefield, 2001).

**MULTI-AGENT SYSTEM KEA**

The aim of the KEA project (Nowostawski et al., 2001) is to provide a modular agent platform with an enterprise-level backend. The architecture supports the use of agent-oriented ideas at multiple levels of abstraction. At the lowest level are micro-agents, which are robust and efficient implementations of agents that can be used for many conventional programming tasks. Agents with more sophisticated functionality can be constructed by combining these micro-agents into more complicated agents. Consequently, the system supports the consistent use of agent-based ideas throughout the software engineering process, because higher level agents may be hierarchically refined into more detailed agent implementations. This enables scalability, flexibility and robustness of the platform, providing at the same time uniform modeling and programming paradigm.

The main distinguishing feature of KEA architecture as compared with traditional software engineering techniques is the autonomous dynamic linking facility. In traditional statically linked code, the function call is statically linked with appropriate library during compilation time. In dynamic linking, the function call is not linked with an appropriate implementation until the runtime. Then, the code is dynamically linked. The dynamic linkage with the library is unconditional (the library cannot refuse the linkage). Once the linkage has been made, it (usually) lasts until the end of the execution of the runtime system.

In KEA, the concept of dynamic linking has been extended further. The association between agents (or function calls if using the traditional programming nomenclature) is postponed until the very time when it is needed. At that time, the linkage is initiated, and may or may not be established. The participating party may refuse participation, in which case the caller will have to deal with this situation by trying alternatives. In case of successful association (when the linkage has been established), it will only last until the end of the current task (or function). After that, a new dynamic linkage must be initiated and established again.

Such a model promotes high-levels of autonomy because no fixed assumption can be made upon available participants. Agents must be prepared to deal with situations where given functionality may not be immediately available, and alternative means of achieving one’s goals must be undertaken. More details about the KEA platform can be found in Nowostawski et al. (2001).

**EVALVABLE VIRTUAL MACHINES (EVM)**

**Overview**

There has been research conducted regarding autonomous asynchronously-interacting com-
computations pursued in diverse areas of theoretical computer science. Certain properties investigated in those settings have been found to be invariant and shared between different complex systems. Our original desire was to integrate the recent advances from various fields onto a single coherent theoretical model, together with an experimental computational framework which could be used for practical investigations on massively parallel computational framework. Originally designed as an artificial evolution modeling tool (Nowostawski et al., 2005b), the EVM architecture is a model for autonomously interacting, evolving, complex and hierarchically organised software systems. The EVM architecture stems from recent advances in evolutionary biology and utilises notions such as specialisation, symbiogenesis (Margulis, 1981), and exaptation (Gould & Vrba, 1982). From the computational perspective, it is a massively distributed asynchronous collection of interactive agents that utilises computational reflection. The EVM framework has been used for multitask learning and metalearning. Hence, computational reflection and reification, on one hand, provide a compact and expressive way to deal with complex computations, and on the other hand, provide ways of expanding a computation on a given level via the metalevels and metacomputations.

Symbiogenesis researchers argue that symbiosis and cooperation are primary sources of biological variation, and that acquisition and accumulation of random mutations alone is not sufficient to develop high levels of complexity (Margulis, 1970, 1981). Other opponents of the traditional biological gradualism suggest that evolutionary change may happen in different ways, most notably through exaptation (Gould & Vrba, 1982), that is, a process whereby a structure evolved for one purpose that has come to be used for another, unrelated purpose (or function).

The EVM architecture follows the biological models of: symbiogenesis, exaptation and specialisation. EVM allows independent computing elements to engage in symbiotic relationships, same as in cham, where independent agents are engaged in relationships through reaction rules and the concept of a membrane, that limits interactions only to local data within a membrane. In the case of EVM the interactions are not only 2-way, but they may involve an arbitrary number of participants. EVM allows a given agent to specialise in specific tasks, or to evolve toward new, more complex, tasks, similarly to the specialisation principle from biology. EVM also allows agents to be used in different contexts than originally designed for, similar to the exaptation principle.

The EVM architecture can be also seen as a computational model that combines the features of a trial-and-error machine (Bringsjord & Zenzen (2003) and the multi asynchronously-interacting machines paradigm. The trial-and-error behaviour is achieved through continuous looping of different hypotheses and their re-evaluation until the desired precision of the hypothesis is achieved.

The EVM model is similar to the one of cham. There are, however, some main differences. In cham, reaction rules are (typically) written between two agents in the solution. In EVM the interactions can happen between more than a pair of agents. Also, in cham, the reaction rules are written beforehand, and not changed during the abstract machine execution. This is not the case for EVM. In EVM, the initial machines executed can modify the rules. It is beyond the scope of this article to analyse the exact formal equivalence and relationship between these two models, and we leave it for future work.

In the following subsection, we will present the details of the EVM implementation, and discuss the experimental.

Implementation

Our current implementation of the EVM architecture is based on a stack-machine. With small differences, the EVM implementation is comparable to an integer-based subset of the Java Virtual Machine (JVM). There are two independent but
compliant implementations: one is written entirely in Java and the second one in C. Developers and researchers can obtain the sources from CVS http://www.sf.net/projects/cirrus. The basic data unit for processing in our current implementation is a 64-bit signed integer. This somewhat arbitrary constraint is dictated by practical and efficient implementation on contemporary computing devices. The basic input/output and argument-passing capabilities are provided by the operand stack, called the data stack, which is a normal integer stack. At the moment, only integer-based computations are supported. All the operands for all the instructions are passed via the stack. The only exception is the instruction push, which takes its operand from the program list itself. Unlike other virtual machines (such as the JVM), our virtual machine does not provide any operations for creating and manipulating arrays. Instead, the architecture facilitates operations on lists. There is a special stack, called the list stack, for storing integer-based lists.

Execution frames are managed in a similar way to the JVM, via a special execution frames stack. There is a lower-level machine handle attached to each of the execution frames. Machine is a list

Figure 1. Schematic view of a single EVM processor: Program and Program Counter, BaseMachine, data (operand) stack and list stack. Below, a concrete instantiation during the execution of swap instruction
of lists, where each individual list represents an implementation of a single instruction for the given machine. In other words, the machine is a list of lists of instructions, each of which implements a given machine instruction. If a given instruction is not one of the primitive Base Machine units, that is, primitive instructions for that machine, then the instruction sequence must be executed on another, lower-level machine. The Base Machine implements all the primitive instructions that are not reified further into more primitive units. To distinguish those primitive instructions that are executed on the Base Machine, we refer to them as operations.

Potentially, EVM programs can run indefinitely, and therefore, for practical reasons, each thread of execution has a special limit to constrain the number of instructions each program can execute. This is especially crucial in a multi-EVM environment. Once the limit is reached, a given program will unconditionally halt.

The EVM offers rich reflection and reification mechanisms. The computing model is relatively fixed at the lowest-level, but it does provide the machines with multiple computing architectures to choose from. The model allows the programs to reify the virtual machine on the lowest level. For example, programs are free to modify, add, and remove instructions from or to the lowest level virtual machine, as well as any other level. Also, programs can construct higher-level machines and execute themselves on these newly created levels. In addition, a running program can switch the context of the machine, to execute some commands on the lower-level, or on the higher-level machine. Altogether, the EVM provides limitless flexibility and capabilities for reifying individual EVM executions. Due to this high level of flexibility, there have been no attempts to formalise the full EVM model in any of the existing process calculi or other computational algebras. We have only attempted partial formalisations of the model.

Each individual EVM is a computational agent that can reference any other machine in the multi-EVM environment (the EVM Universe). This is achieved by using the first 32 bytes of the instruction to address any computer in the Internet, and the second 32 bytes for the index of the instruction on that machine. That mean that the theoretical limits of the EVM universe are bound by \(2^{32}\) number of hosts with \(2^{32}\) instructions on each of the hosts.

One possible way of instantiating part of the computational environment for the architectural framework is by adapting bias-optimal search primitives (Levin, 1973), or the incremental search methods (Schmidhuber, 2004). To narrow the search, one can combine several methods together. For example, it is possible to construct a generator of problem solver generators, and employ multiple metalearning strategies for a given computational task at hand. A more detailed description of the abstract EVM architecture is given in Nowostawski et al. (2004). The experimental results are described in detail in Nowostawski et al. (2005a).

**EVM Search Process**

The EVM Universe is composed of a spatially distributed grid of EVM agents (cells) each of which is trying to solve one (or many) tasks provided to the system through the special Task Manager that is part of the environment. Each individual cell works independently of other cells, and it can perform a finite number of operations. We have implemented and deployed several different search algorithms that the EVM can use: random search, stochastic search, genetic algorithms, exhaustive bias-optimal search. The actual search performed by the individual cell is decided by the cell itself. In this research, we concentrated our attention on the interactions and dependencies between the cells. A single EVM agent can use other agents. This is typically done in such a way that a solution for a complex problem is decomposed into subtask,
each of which is delegated to other agents. It is common for any of the search methods to quickly solve subtasks by delegating the processing to the other agents. Each cell can specialise in a given subtask by reusing another cell for this subtask. Such a cell will be called a first-level parasite. If a parasite uses another parasite, we call it higher-order parasite. Parasites appear often within certain classes of problems, as it becomes exponentially easier for a cell to parasite an existing solution than to come up with a solution on its own. This is due to the interactions between the spatial aspects of the asynchronous interactions between agents and internal processing capabilities of the computational agents themselves.

The dependencies and bonding between cells has been achieved through random search process (trial-and-error search). For certain classes of arithmetic problems with which this model has been tested, random search was sufficient to find solutions to tasks consisting of up to 3 subtasks (the neighbourhood of a single cell was restricted to only four cells). For problems with higher order of combinatorial search space the random search was insufficient, and must be augmented with other heuristics to establish proper bonding between computational units.

**Simple Grid Experiments**

In one of our early experiments (Nowostawski et al., 2005a) we have used regular toroidal grid, with only one program per cell. We used four neighbours models, and locality played an important role in the dynamical evolution of the cells’ interactions. Our multi-agent system exhibits properties found in other artificial life systems such as Tierra (Ray, 1991) (for instance, knowledge diffusion, parasitism, self-assembly). The experiments were conducted on a single level of EVM, with the aim of discovering a good compact machine suitable for a given sequence of tasks. We have used two probability-based learning methods: one based on individual probabilities of instructions, and the second based on conditional probabilities of sequences of instructions. We have introduced, investigated and compared five different specialisation mechanisms:

1. Random search and exhaustive search;
2. Classic genetic algorithm;
3. Ad-hoe stochastic search for a fixed-size program;
4. Improved tree search based on successful patterns (building-blocks); and
5. Universal reinforcement acceleration mechanism.

All these search mechanisms operate at the low, cellular level, and dictate, in interaction between agents within the multi-agent system and the macroscopic behaviour of the system. A typical run, exhibiting many features of our self-organising, self-adaptable cellular system is presented in Figure 2.

In order to be efficient in a multitask context, we have assumed that the model must fulfill these requirements:

1. All tasks must be eventually solved.
2. Solving difficult tasks should lead to greater rewards than solving easy ones.
3. Computational resources should focus on unsolved tasks.
4. Solutions must not be forgotten, as long as they are useful.
5. Knowledge diffusion should be facilitated (previous solutions must be accessible).
6. Dynamic environments should be supported: tasks can be added or removed at any time, dynamically.

For arithmetical problems, cell’s programs are short (typically four instructions). The reason is that we want to focus on the way these programs will collaborate and reuse other existing subexpressions from neighbours. Some of the tasks tackled are enumerated in Table 1. Note that, for
Figure 2. Typical evolution of the cellular system. Note that the topology is toroidal. For example, right neighbours of the cells on the right border of the grid are the cells along the left border of the grid.
instance, the solution to $2x + 3y$ can be only five instruction long (leftNeighbourProgram swap rightNeighbourProgram add halt), and thus much more likely to be found, if a cell has a (left) neighbour that solves $2x$ and another (right) that solves $3x$. That's the whole point of our system, that of reusing knowledge.

In the table below, we present simple tasks that a given agent must discover using random search. The EVM assembly language is similar to the JVM opcodes. For example, const_2 is equivalent to a constant 2, add, mul are equivalent to addition and multiplication operations, inc increments the top element on the stack, swap changes the order of the first two elements on the stack, and so forth.

### AUTONOMY AND EVM INTERACTIONS

When the search process starts, initially all the agents are free to try to solve the problem themselves. At the same time, agents can try to re-use other agents’ capabilities. When being used by others, agents can dynamically change their behaviour and therefore break the bonding between themselves and other agents. With such a MAS setup, we can say that it comprises a high-level of relative autonomy, because agents are free in their choices to conduct or not any interactions with other agents.

The agents are given a list of tasks, some of which are simple, some of which are hard and comprise of subtasks. The harder tasks require a single agent to search a large program search space, to solve all the required subtasks first. Due to the complexity of the search process, it is easier for the agents to cooperate, and reuse solutions to subtasks already solved by other agents. This is easier than to build a whole solution on their own. When the external flow of problems is fixed, agents self-organise into fixed clusters and rarely break the bonding, even though they could. This is due to the fact that agents have no incentive of breaking the bonding, as it would be less efficient to break the bonding and try to find a new stable state, than it would be to remain in the current bonding permanently. In dynamic environments, however, agents are frequently forced to break the bonding due to changing requirements. Agents frequently form new quasi-stable bonding clusters. This behaviour is intuitively simple, and boils down to agents’ benefit calculations: if it is more efficient to remain in the current state, than to change it, the agent will continue with current bonding arrangements. If it is faster to obtain benefits by cooperation and bonding, then this

**Table 1. Some examples of the arithmetical tasks tackled**

<table>
<thead>
<tr>
<th>Task</th>
<th>Solution</th>
</tr>
</thead>
<tbody>
<tr>
<td>$x+y$</td>
<td>add halt</td>
</tr>
<tr>
<td>$xy$</td>
<td>mul halt</td>
</tr>
<tr>
<td>$2x$</td>
<td>const_2 mul halt</td>
</tr>
<tr>
<td>$3x$</td>
<td>const_2 inc mul halt</td>
</tr>
<tr>
<td>$X$</td>
<td>ifge nop neg halt</td>
</tr>
<tr>
<td>$2x+y$</td>
<td>const_2 mul add halt</td>
</tr>
<tr>
<td>$2x-y$</td>
<td>const_2 mul sub halt</td>
</tr>
<tr>
<td>$2x+3y$</td>
<td>const_2 mul swap const_2 inc mul add halt</td>
</tr>
<tr>
<td>$7$</td>
<td>const_2 const_2 inc add halt</td>
</tr>
</tbody>
</table>
Concept of Autonomy in Distributed Computation and Multi-Agent Systems

Figure 3. Self-assembly. Bottom. Left: first, some cells discover the solutions to the easy tasks (2x and 3x). These solutions can be reused by their neighbour to compute a more difficult, but related task (3x + 2y). These three cells live in symbiosis together (middle). At the same time, two other cells (green and grey) manage to discover together a solution to (49 − x). The green cell computes that solution with the help of the grey cell. The grey cell solves no task alone, but still gets rewards and survives because it contributes to the computation of the green cell. Right: Eventually, a cell between these two blocks of cells connects them to solve a more complex task (49−(3x+2y)). Top: Details of the cells’ programs.

is observed. If simple tasks are more efficiently solved by individual agents alone, then this is a predominant behaviour. This is very similar to other phenomena in complex asynchronous systems in physics, chemistry and biology. This phenomenon of restricting degrees of freedom of a given entity is called enslavement (Haken, 1983). This means, that a certain stable energy level has been obtained. The amount of participation in a particular arrangement beyond a certain threshold locks more and more participation. A process of enslavement is being observed.

Some researchers designing and developing MAS have also noticed that the autonomy and independence are not always the most desirable features. Some constraints are often necessary for the overall goals of the MAS system to exceed capabilities of individual agents. Based on our experiments, we can conclude that full autonomy and complete freedom is not always desirable. A form of enslavement must be present when designing and building MAS. Enslavement helps the system to settle in certain configurations or
dynamical patterns, which would be not attainable otherwise.

Another related question is how to make MAS (absolutely) independent from the actual programmer. This requires that agents have the abilities to learn new activities, adopt new goals and devise new learning strategies themselves (see also Witkowski & Stathis, 2003). The aim in open MAS is not (or not primarily) to create certain outcomes by fixed programs that agents execute. Rather, it is to support and embody a transformation whereby agents subscribe to an open-ended mutual learning process, as in the programming process of purely computational agents within EVM framework. In such computational ecosystems the practices are considered open-ended, that is, no preconceived result is intended for individual agents. Instead, only the initial boundaries or rules of some process are defined and the actual development is left to the interaction of participating agents.

CONCLUSION

In this chapter, we have discussed the notion of autonomy in multi-agent systems. We have reviewed the existing definitions and formalisation attempts. We have proposed our own formalisation based on the notion of universal Turing machines computational agents, with the abstract notion of data sources and data transformations. Based on the assumed notions of computation, the concept of relative and absolute autonomy for a given computational agents have been presented. We compared our definition to existing intuitive definitions in multi-agent literature. We have provided also a comparison of general autonomy classes in MAS, with intuitive and formal notions of autonomy.

In the context of autonomy in MAS, we have presented details of two multi-agent systems: KEA and EVM. These frameworks tackle the challenges of autonomous computing in various ways. In both the emphasis is placed on the central notion of unsecured and unreliable interprocess (or interagent) communication. The KEA framework is using the notion of autonomous dynamic linking between agents. The EVM system is using the notion of unstructured self-assembly and dynamic aggregation of computational components.

Based on the literature review and our own observations, we have concluded that the autonomy is directly linked with the concept of indeterminacy in a sense of Turing-computability. In that context, it is easier to understand why autonomy is a subject of continuous restrictions from various angles within MAS community. From one hand, unlimited autonomy makes it extremely hard to design, program and analyse MAS systems. Therefore, restricting autonomy is one way of dealing with the complexities of MAS design. On the other hand, restricting autonomy is an inherently needed property to achieve global coherent behaviour, which may otherwise be unattainable.

We have discussed EVM-based experiments which show that only through limiting individual agent autonomy and restricting the freedoms of choice can a more complex computational structures be achieved. This seems to be an inherent property of any complex systems composed of a large number of autonomously interacting entities. This phenomenon is called enslavement in synergetics (Haken, 1983).

REFERENCES


Gouaich, A. (2003). Requirements for achieving software agents autonomy and defining their responsibility. In Nickles et al. (Eds.), (pp. 128-139).


Weigand, H., & Dignum, V. (2003). I am autonomous, you are autonomous. In Nickles et al. (Eds.), (pp. 227-236).