ARTÍCULO

Multi-agent Platform for Distributed Soft Computing

Piotr Biegański, Aleksander Byrski, Marek Kisiel-Dorohinicki

Department of Computer Science
AGH University of Science and Technology
Mickiewicz Avn. 30
30-059 Kraków
piotr.bieganski@gmail.com, {olekb,doroh}@agh.edu.pl

Abstract

This paper presents an approach to construct efficient and extensible multi-agent platform for distributed soft computing. Various software engineering techniques are employed to implement reliable and reusable system architecture. Extensible XML based configuration is used to simplify the process of repetitive simulations performed with use of constructed toolkit. The general purpose of the idea is applied to the construction of evolutionary multi-agent computational system.

Keywords: Multi-agent systems, soft-computing, function optimization.

1 Introduction

Soft computing techniques, such as evolutionary algorithms, artificial neural networks, or fuzzy systems, have attracted growing interest during the last decade. A prominent role in the research in the field play hybrid systems—based on combining different ideas and methods—that by the effect of synergy are often said to exhibit some kind of intelligent behaviour [**?**]. This is sometimes called *computational intelligence* (CI) as opposed to rather symbolic artificial intelligence.

Such systems are used today for more and more complex problems requiring processing of huge amounts of data and long computational time. Parallel and distributed implementations seem to be a promising answer to this problem, especially that many techniques (such as evolutionary computation or ant colony optimisation) are highly parallel by nature. What is more, it turns out that their population-structured models are often able to provide even better solutions than comparably sized classical ones—considering not only the quality of obtained solutions and convergence rate, but first of all a *global* convergence reliability. As a typical example of such approach, decomposition (fine- and coarse-grained) models of *parallel evolutionary algorithms* may be recalled here, as they have been successfully used in a number of applications [**?**].

The idea of building such systems can be essentially ordered and enriched using the notion of an intelligent agent—a software entity situated in some environment and autonomously acting on it so as to satisfy its own goals. A multi-agent system (MAS) is designed and implemented as a set of interacting agents and the interactions *cooperation, coordination or negotiation* turn out to be the most characteristic and powerful component of the paradigm [**?**]. Conceptual relation between particular soft computing techniques and agents or their populations forms a base for distinguishing various levels of architectural design. It can be foreseen that the proposed approach can not only give a universal platform for cooperation of

different known methods but also magnify their solving abilities up to gaining *computational intelligence* at a level of agent populations [**?**].

Nevertheless considering the complexity of these systems, a serious need arises to create flexible and extensible, yet well-structured management scheme that will help to create reliable and scalable distributed environments. In the paper a platform that supports a processor-based management scheme is described. The scheme allows for the definiton of workflows in terms of cooperating software components that constitute the computational units (agents) of the system.

The paper is organized as follows. Section 2 gives fundamental assumptions and identifies core funtionalities of the system. In section 3 the proposed architecture is discussed in depth. Selected implementation details conclude the work in section 4.

2 Needs and expectations

The primary goal of the platform design is to relieve the developer from implementing majority of the functionality not related to the problem to be solved. This entails specific expectations that cover three aspects of the system:

Problem issues – problem-dependent requirements that must be fulfilled by the system – e.g. the possibility of introducing ordered graph of connections between computational nodes (virtual computing nodes corresponding to the evolutionary islands from the multiple-deme evolutionary computation model, on every physical computer, one or more computing nodes may be present), or the components supporting specific operations (e.g. evolutionary ones, such as crossover, mutation, migration etc.).

Technological issues – based on specific technology (such as MPI, PVM e.a.) devoted to support communication in a distributed environment among many computational nodes, concurrent computation and reusable component-based structure of the system.

Management issues – including dynamic reconfigurability, on-line monitoring, easy management of the whole system.

Problem presumptions

The constructed distributed environment consists of one or more *locations* representing regions containing local populations of the individuals. Each location usually provides specific management procedure (e.g. evolutionary or ant colony computation) and may be implemented at will. Such a decomposition approach makes possible to perform problem processing in distributed network. Additionally it is allowed to define *topology* of region connections in the form of directed graph, specifying any *neighbourhood relation*.

Considering flexibility of the system, which should perform many different operations on groups of agents (populations) contained in the locations, the system should provide the developer with many different components (processors) that may ease the changes introduced into the populations. Considering evolutionary system, such operators may perform functions of crossover, mutation, preselection e.a. In fact the implementation of these processors may vary depending on the specific application issues (such as genotype representation in evolutionary computation).

Technological presumptions

The designed software platform should allow for *concurrent computation* and *asynchronous communication* among the processes in the system. In order to simplify the implementation process and efficiency of the whole solution, apart from *parallelisation*, *event-driven simulation* is considered. I.e. some parts of the system are realised as parallel tasks (processes or threads), and run concurrently on a multi-processor machine, while other parts of the system may undergone an event-driven simulation: entities are activated one by one, or wait for their activation in a queue.

Usually the parallel processes are used to implement locations [**?**] and *inter-location communication*, while the event-driven simulation used inside every process, in order to support the agents' behavior and *intra-location communication* (application of the whole system to the problem of evolutionary computation leads to considering agents as individuals, locations as demes, and the

whole system as multiple-deme distributed environment).

The system should be constructed in such way that the consecutive steps of creating of the system should to a great extent rely on the *reusable predefined components*. It should allow for quick adaptation of the new programmer to the specific features of the platform, and limit the cost of debugging. The design patterns should become especially useful for implementing such a complex system.

Management presumptions

Considering soft-computing purposes, it is usually needed to perform parametric tuning of the simulation, e.g. looking for the good solution of the given problem, that is strictly dependent on the specific parameters of the simulation. To achieve this, one must perform sequence of similar tests with different sets of parameter values. Note that the *parameter* of algorithm may be even a proper implementation of some subroutine.

Automatic testing all possible variants should become one of the main objectives of the constructed platform. It is required to allow to specify ranges of values or possible implementation variants in the configuration. To perform all tests it is required to automatically iterate through all possible configuration versions during runtime. Another important idea is the way of specifying global *meta-configuration*, describing all possible variants, and mechanism of successive and automatic variant distribution.

The ability of *seamless reconfiguration* of the whole system is crucial for the simulation problems, where the work of the system is dependent on the set of parameters, and optimal parameter configuration leading to obtaining optimal solution needs usually many simulation with different parameters to be run. The configuration of the platform should also be extensible, in order to adapt to the changes of the user's requirements. In fact, user should be able to easily influence parameters of the simulation, as well as the simulation algorithm. Thinking of the technology used as a base for this kind of configuration, the use of XML should be considered, because of the popularity of this standard.

Complex simulation or computing systems may produce lot of more or less important results. Some of the data should be kept for future analysis, but some should be verified as fast as possible. Unfortunately, usual practice of the monitoring consists in simply logging the results into files or databases, so it is impossible to perform automatic interpretation, without the interaction with user. An useful idea should be to make possible *on-line monitoring* of the specific simulation parameters at runtime. Collecting statistics and results should be entrusted to a separated network module, so it does not interfere with the computational procedures.

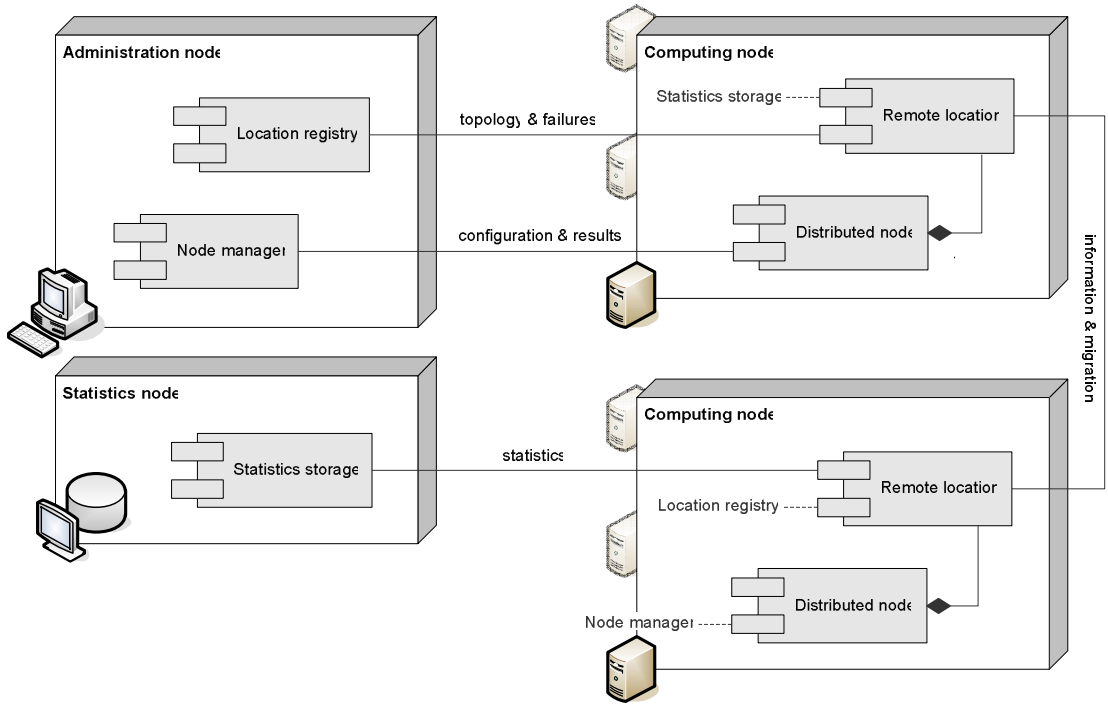

Fig. 1. Distributed structure of the system

3 System architecture

Each role in the platform is represented by a separate module, working at the node of specific type (fig. 1.):

Managing node controls the whole computing environment, contains *locations registry* and *node manager* which are responsible for different aspects of distributed system management:

- *Location registry* determines the environment topology using neighbourhood relation.
- *Node manager* globally controls the stopping condition, delivers local configurations, and collects final results.

Processing nodes perform the computation. Each node may run its own *location*, and may obtain a list of neighbour locations from the location registry to communicate with them.

Observing node is not directly involved in the computation. *Statistics storage* collects lots of data about the computation: the results of processing cycles, states of consecutive populations, and other factors, like effects of mutation, migration, etc.

Node manager is responsible for the management of the distributed system. It is placed in the *administration node* and communicates with the *distributed nodes* which are situated in the computation nodes, being the interface for the distributed computation. One of the most important functions of the *node manager* is the handling of the system's meta-configuration, which may be perceived as a skeleton for the specific configuration of the computation nodes. Using the meta-configuration information, consecutive versions of node configuration are generated, which differ only by certain features (e.g. values of some parameters). *Node manager* performs following services for the *computation node*:

- provides *computation node* with specific configuration,
- controls the stop condition of the computation,
- synchronizes work of the *computation node*.

Node manager stores in every step of the work the best solution of the given problem found in the nodes it manages.

Each computing node is placed on a separate machine, and works as the slave subsystem for the node manager. In this case it accepts consecutive variants of configuration sent by the manager and runs proper computing routines (for example an evolutionary procedure). It is possible to perform computations in synchronized and non-synchronized mode, or even to omit the manager and define the whole procedure in the configuration of computing node.

Remote locations are situated in the computing nodes, they provide specific interface for communication and migration of the agents among the locations. The process of agent-migration in the described system is realized using simple data-migration technique – the agent that should migrate is removed from the one of the computing nodes and is recreated on the other computing node using specific information describing its state. *Remote locations* are managed using the *locations registry*.

Location registry is placed in the *administration node*, along with the *node manager*, and is used for system initialization (remote locations have to localize themselves in the network environment before beginning of the computation) and coordination of the whole system. Location registry performs following functions:

- allows for registering and unregistering of distributed locations,
- provides locations with information about its neighbours (newly appearing and being removed ones),
- notices certain locations about new locations appearing in the system,
- performs logging of the communication error messages between locations,
- removes non-working (lost) locations from the environment.

Information may be exchanged at several stages of the location's algorithm. Especially the migration may be realized by the main procedure of the computing node, or it may be the part of evolutionary procedure. The aspect of agents may provide delegation of the decision authority to the individual, therefore the decision-making procedure will be invoked during the process of evolution, which is an independent enditity – as described above. This requires communication support for the developer, offered in library as the processing module.

The communication between locations is possible since the location programs becomes visible to each other. This is usually realized using some form of net addresses, delivered with broadcast messages or by the managing module. For lot of systems it may be assumed that the communication must not be chaotic. Therefore the second solution gains much more sense after introducing concept of environment topology.

Specifying topology implies definition of some contiguity relation between locations and requires the location registry program to manage the environment, delivering selected net addresses to properly set up locations.

Architecture of the processing node

The management procedure of the computation node depends on the chosen mode of distributed processing. It may be easy changed by replacing configuration file. The procedure obtains configuration variants from the remote node manager and performs computations, cyclically verifying global stop condition, controlled by the manager.

Except for the distributed communication, architecture of processing node strongly depends on the application of introduced system.

The shape of evolution process is provided as a part of configuration of location program. It indicates that contents of the procedure, their number, order and allowable parameters cannot be foreseen. The only thing predictable is that an input and output of the whole evolutionary mechanism is the population. Therefore the simplest soluition allowing to create flexible platform is to define adequate interface of population processor, and to leave implementation to the developer.

While the procedure can be seen as the population processor, its ingredients may also be represented as the chain of such processors, performing successive steps of the algorithm.

For the case of more complex flowchart, where:

- some subroutines have to invoked conditionally,
- population must be split into parts and serviced by different processors,

- different kinds of data must be processed by specialized subroutines,

it is required to specify hierarchical organization of structure and to define specialized interfaces (fig. 2.).

The complex algorithm is divided into consecutive steps, and these steps are implemented inside the specific processor components. Each processor is capable of transforming the whole population, and the sequence of processors is used to implement complex transformations (e.g. in evolutionary systems, processors may be used to implement different phases of evolutionary algorithm – preselection, mutation, recombination e.a.).

The processors may be used to transform the whole population of the individuals (e.g. population provider used to initialize the population), the specific group of the individuals (e.g. recombination processor used to create new individuals using the genotypes of its parents) and single individual (e.g. mutation processor used to perturb the genotype of the newly created individual).

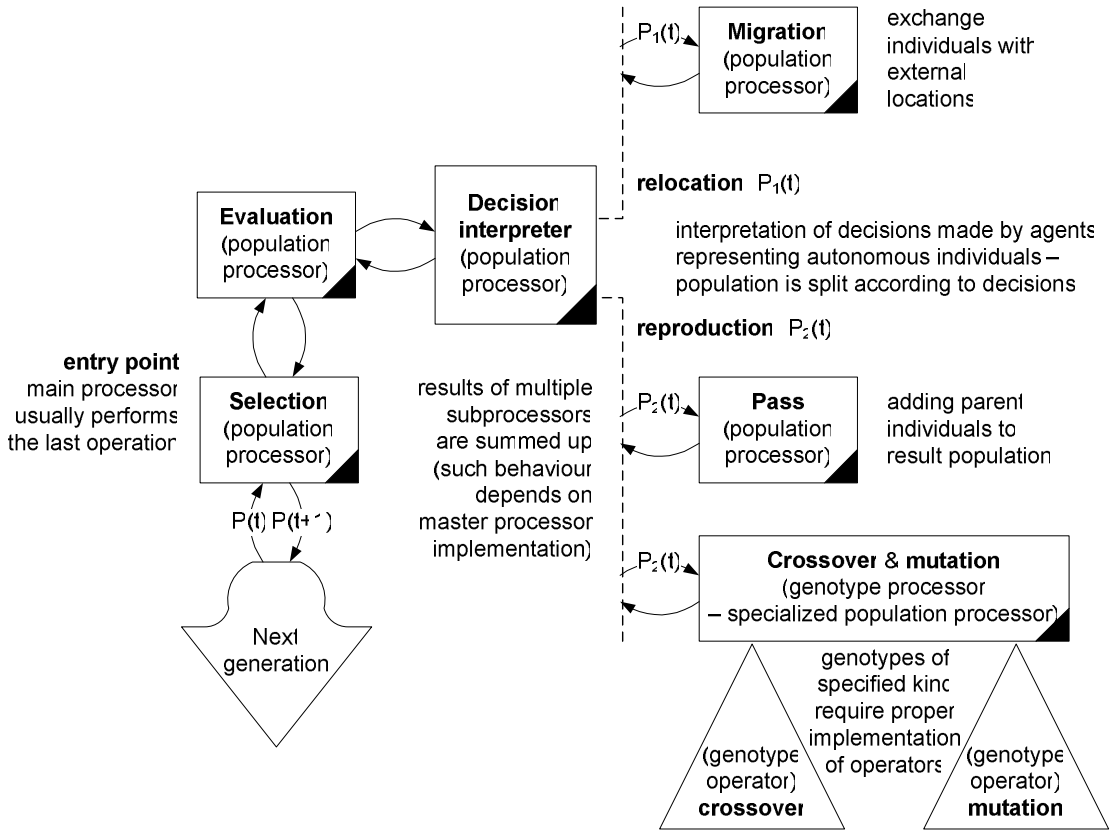

Fig. 2. Evolutionary routine of processing node

Finally the evolutionary procedure is the form of functional language, representing hierarchical structure of processing objects, obtaining some parameters and subsequent parts of the hierarchy. This can be easily described using the XML syntax.

As long as sets of functions are represented by interfaces, it is possible for developer to design his own solutions or to use some offered by the platform library.

4 Selected implementation issues

The platform comes as a library of modules implemented with the use of Java technology, RMI-based remote communication, and a set of external libraries (*log4j* for the purpose of logging, *dom4j* to deal with XML parsing, and *JFreeChart*

for the statistics module). To implement the complete system it is required compose predefined and user-defined components, preparing appropriate configuration files. The implementation methodology may be expressed by the following **construction layers**:

- a set of well-defined interfaces describes assumptions for main program units and internal modules,
- a set of executable units must conform to the requirements of the distributed architecture,
- a library of predefined, universal or specialized implementations, also for specific classes of systems (for example evolutionary systems),
- the configuration defines rules of coupling and setting up all required objects.

These layers form a base for reuse and extensibility of the platform—most of the functionality in the system is represented by (as many as possible independent) interfaces, and the selection of adequate objects and establishment of object connections is left for the configuration level.

To speed up the development of particular applications the implementation is based on well-known **design patterns**. High flexibility was achieved thanks to use of *façades* and *mediators*, with indicated profits of encapsulation provided by interfaces and employment of *decorators*. *Factories* support substitutability of families of interacting objects related to particular solving methods or particular problems. The control flow of the main management routines is defined in terms of *state machines*, while the computing routine utilises a *composite* of processors. Many of them are realisations of *interpreter* or *strategy* patterns. More detailed description of the core structures follows:

Factories. The main routine of the solving (e.g. evolutionary) subsystem is assumed to work aside from the solved problem. Inside the routine there may be lots of objects created: agents, evaluation results, decisions, etc., which are strongly irrespective of the determined control flow. In order to allow creation of objects of unspecified shape, the solving procedure widely uses *Polymorphic factories* and *Abstract factories*.

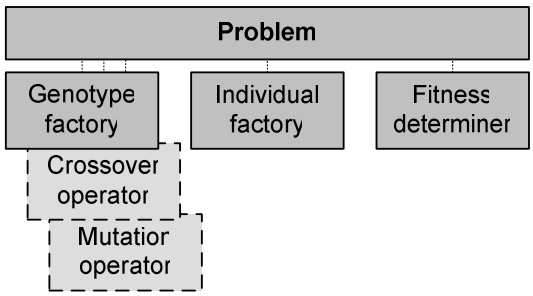

Fig. 3. The problem described with three factories for the evolutionary procedure

Application of factories allows to separate implementation of computational parts from the details of the problem to be solved. All problem-dependent entities are created with specified factory, yet used through interfaces during the processing, like an individual's genotype or an individual itself in evolutionary computation (fig. 3.). By providing factory interfaces the platform transfers some reponsibility to the developer, but such an approach brings lots of profits. Not only existing solutions may be easily adopted by implementing proper interfaces, but also even ready systems still remain flexible and provide extendable and reusable modules.

Composition. For easy setup and modification of the routine of computation the *composite* pattern was used, imposing a common interface of processors. Processors may perform some computation on the acquired data, or control the flow of data, passing it to other processors to perform their jobs. The reconfiguration of the solving procedure is thus possible without recompilation of the code, just by modifying the configuration. Thanks to the common interface, processors may be relocated without watching out if one may be really connected to another.

Thanks to **loose coupling** it is easy to replace almost any part of the system with a more appropriate implementation. This concerns the computing node module, which may be substituted by any agent able to read manager's requests and perform any actions to produce the result to be sent back. Another example is the statistics storage, already implemented in two ways: as a dummy module, and as the persistent storage providing visualisation.

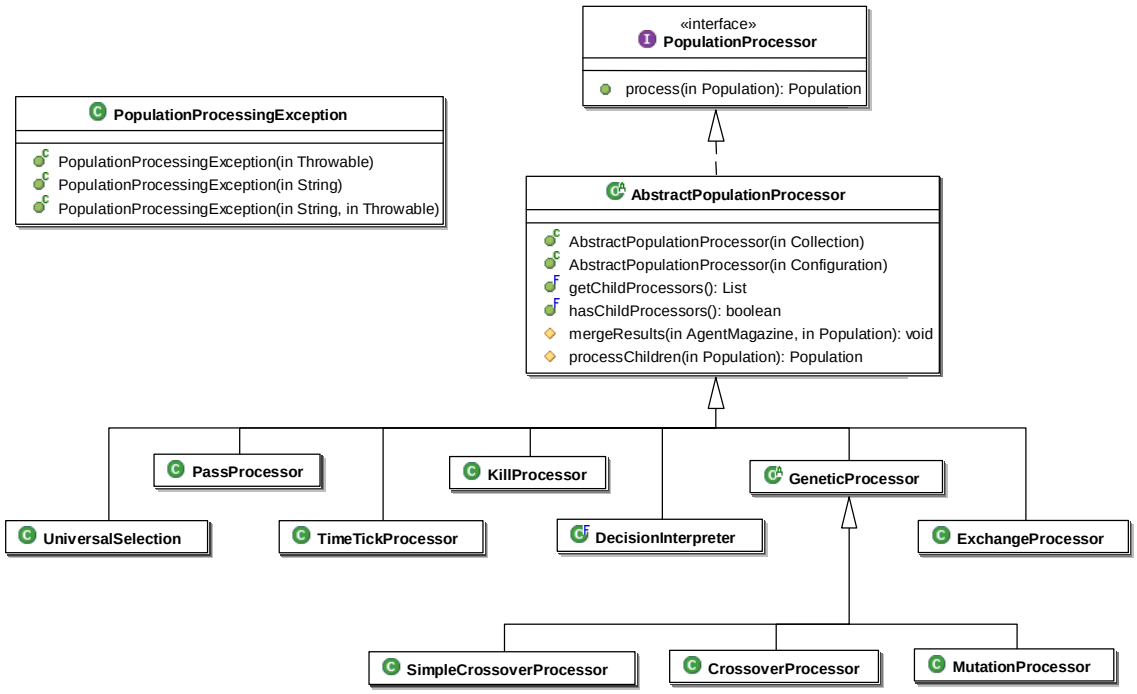

Fig. 4. Numerous population processors based on the same skeleton

The purpose of the **library of ready solutions** is to provide reusable components required to introduce many solving algorithms and several kinds of distributed porcessing schemes, that have been designed as possibly abstract interfaces. Often some skeleton implementations were additionally proposed (see e.g. fig. 4.).

5 Concluding remarks

In the paper, the idea and specific demands of the distributed soft-computing platform were presented. Trying to fullfil design presumptions that were posed in the beginning, a system was implemented using Java technology. The main effort was put on the reusability and reconfigurability of the system, in order to simplify the process of development. Additionally complex user interface including the ability of on-line monitoring of the certain system activity was implemented. The whole system may be succesfully used to construct complex soft-computing solutions, such as systems based on the evolutionary agent computation paradigm.

References

[1] P. Bonissone. Soft computing: the convergence of emerging reasoning technologies. *Soft Computing*, 1(1):6–18, 1997.

[2] A. Byrski, L. Siwik, and M. Kisiel-Dorohinicki. Designing population-structured evolutionary computation systems. In T. Burczyński, W. Cholewa, and W. Moczulski, editors, *AI-METH2003 Methods of Artificial Intelligence*. Dept. for Strength of Materials and Computational Mechanics, Dept. of Fundametals of Machinery Design, Silesian University of Technology, Gliwice, 2002.

[3] E. Cantú-Paz. A summary of research on parallel genetic algorithms. *IlliGAL Report No. 95007. University of Illinois*, 1995.

[4] J. Ferber. *Multi-Agent Systems. An Introduction to Distributed Artificial Intelligence*. Addison-Wesley, 1999.

[5] M. Kisiel-Dorohinicki, G. Dobrowolski, and E. Nawarecki. Agent populations as computational intelligence. In L. Rutkowski and J. Kacprzyk, editors, *Neural Networks and Soft Computing*, Advances in Soft Computing, pages 608–613. Physica-Verlag, 2003.