Functional Integrity of Multi-agent Computational System Supported by Component-Based Implementation

Kamil Piętak, Adam Woś, Aleksander Byrski, and Marek Kisiel-Dorohinicki

AGH University of Science and Technology, Kraków, Poland
{kpietak,awos,olekb,doroh}@agh.edu.pl

Abstract. In the paper a formalism is proposed to describe the hierarchy of multi-agent systems, particularly suitable for the design of a certain class of distributed computational intelligence systems. The notions of algorithms and dependencies among them are introduced, which allow for the formulation of functional integrity conditions for the whole system. General considerations are illustrated by modeling a specific case of an evolutionary multi-agent system. Component techniques introduced in *AgE* computing environment facilitate the implementation of the system in such a way that algorithm dependencies are represented as contracts, which support checking of the system's functional integrity.

Keywords: functional integrity, components, mutli-agent systems.

1 Introduction

The notion of *functional integrity* may be understood as the ability to fulfill functional requirements in a complex system. In agent-based environments, it may be really difficult to check whether the set of cooperating but autonomous agents is able to achieve the global goal of the system (solve the problem) [1]. Due to their dynamic nature, it may require to forecast and manage different critical situations, such as sudden breakdown of hardware [2]. But first of all agents must be able to cooperate with one another, which requires adequate infrastructure addressing interoperability issues [3].

The paper focuses on a specific class of agent systems, which use computational intelligence paradigms – particularly hybrid techniques based on the concept of decentralized evolutionary computation [4]. These systems consist of a hierarchy of agents and nested multi-agent subsystems, which should use compatible structures and mechanisms to be able to work together. What is more, some agents perform similar tasks, but work with different structures and mechanisms. Thus from the software engineering perspective it may be said that the system is decomposed into particular agents, but a single agent implementation is too complex to serve as an assembly unit. In fact agents implementations may be further decomposed into functional parts (components), which are replaceable, as long as they are compatible to one another, even when used by different agents (this ensures agents interoperability at implementation level).

As it was discussed in [3], both agents and components can be considered in software development as assembly units, which implement in various ways the concept of responsibility delegation. However, agents-based technology focuses more on executing complex tasks in a community to achieve defined goals and, on the other side, component-based technology is rather aimed at reusability and integrity aspects of software development [5]. Indeed, it seems that component-oriented approach can be also successfully exploited in agent-based systems for assembling parts of agents implementation and checking the integrity of the system.

To facilitate the design of the systems under consideration a dedicated formalism was proposed in [6]. The goal of this paper is to show how it may be extended to formulate the *functional integrity* conditions of the system, and how these conditions may be supported by component technology. In the course of paper the notion of algorithms (used by agents for performing actions) is introduced, as well as additional relation (dependency) imposing the existence of algorithms required by actions, as well as other algorithms. Agents and algorithms are implemented and provided to *AgE* computing environment[1] as components. Relations between them are implemented as component contracts handled by the configuration engine of *AgE* system, which is also responsible for the verification of the system functional integrity.

The paper begins with the presentation of a formalism, which allows for the formulation of functional integrity conditions of the system. The case of an evolutionary multi-agent system is discussed as an illustration in the next section. Finally the realization of these concepts using component techniques in *AgE* computing environment is shown and some conclusions are drawn.

2 Functional Integrity of a Computing MAS

The model proposed in [6] defines an agent as a tuple:

$$AG \ni ag = \langle id, tp, dat_1, \ldots, dat_n \rangle \tag{1}$$

where $id \in ID$ is a unique identifier of an agent[2], $tp \in TP$ denotes the type of an agent (depending on its type, an agent is equipped with specific data and may perform specific actions), and $dat_i \in DAT_i, i = 1, \ldots, n$ represents problem-dependent data (knowledge) gathered by an agent.

According to [6] a multi-agent system includes agents, actions to be executed by the agents, and the environment represented by some common data, which may acquired by the agents. This definition must be extended here with a set available algorithms:

$$AS \ni as = \langle Ag, Act, Alg, qr_1, \ldots, qr_m \rangle \tag{2}$$

[1] http://age.iisg.agh.edu.pl/

[2] For each element of the model its domain, which is a finite set of possible values, is denoted by the same symbolic name in upper case, e.g. *ID* is the set of all possible agent identifiers.

where $Ag \subset AG$ is the set of agents of as, $Act \subset ACT$ describes actions that may be performed by the agents of as, $Alg \subset ALG$ is the set of algorithms available in as, and $qr_i \in QR_i, i = 1, \ldots, m$ denote queries providing data (knowledge) available for all agents in as.

An agent may provide an environment for a group of other agents, which by themselves constitute a multi-agent system, which is essentially different then the one of the "parent" agent. These nested (multi-agent) subsystems introduce a tree-like structure, which will be further referred as *physical hierarchy* of agents. The relation $\gamma : AS \to AG \cup \{\varnothing\}$ identifies an agent that provides the environment for a particular agent system. For details see [6].

In the space of types, a subsumption relation "$\preceq$" $\subset TP \times TP$ is defined, introducing a partial order in TP. In terms of this relation, $A \preceq B : A, B \in TP$ means that A is a subtype of B.

Agents may perform actions in order to change the state of the system. An action is defined as the following tuple (Hoare's triple equivalent [7]):

$$ACT \ni act = \langle tp, pre, post \rangle \tag{3}$$

where $tp \in TP$ denotes the type of agents allowed to execute the action (only agents of the type tp and descendant types – according to the "$\preceq$" relation – may perform the action); $pre \in X$ is the state of the system which allows for performing action act; $post \in X \times X$ is the relation between the state of the system before and after performing action act.

Actions may depend on algorithms, i.e. in order to perform an action, one or more algorithms may be needed. This dependency is described by the following relation:

$$\text{"}\rightsquigarrow\text{"} \subset ACT \times ALG \tag{4}$$

For algorithms there is also a subsumption relation "$\preceq$" defined, which states whether one algorithm is a specialization of another:

$$\text{"}\preceq\text{"} \subset ALG \times ALG \tag{5}$$

Relation "$\preceq$" introduces a partial order in ALG (it is reflexive, transitive and antisymmetric), i.e. if $A1 \preceq A$ then $A1$ can be used in place of A when needed.

When an action is about to be performed, a subset of algorithms is selected from Alg according to the $\preceq$ relation and any further restrictions described below. These algorithms are said to be "available" in the environment for the execution of a particular action[3]. For example, if action act depends on algorithm A, and in a particular system $\exists!\, subA \in Alg : subA \preceq A$, then when the action is executed, it uses algorithm $subA$.

Further dependencies between algorithms used in the system may be described using the following relation:

$$\text{"}\rightsquigarrow\text{"} \subset ALG \times ALG \tag{6}$$

[3] A discussion on how this selection is realized in AgE is provided in section 4.

If algorithm A depends on algorithm B ($A \rightsquigarrow B$) it means that B or its subtype is needed for A to function properly, and must be available in the system. Relation "$\rightsquigarrow$" allows for defining families of algorithms that are designed to be used together (i.e. if one of them is selected by the environment for the execution of a particular action, then other algorithms from the same family are also selected).

For example, let us assume that the set of all known algorithms is $ALG = \{A, A1, B, B1, B2, C\}$, where $A1 \preceq A$, $B1 \preceq B$ and $B2 \preceq B$, the set of all known actions is $ACT = \{act\}$ and that $A1 \rightsquigarrow B1$ and $A1 \rightsquigarrow C$. Moreover, let us consider an AS with the set of available algorithms $Alg = \{A1, B1, B2, C\}$. When action act dependent on both A and B ($act \rightsquigarrow A$, $act \rightsquigarrow B$) is to be performed, the set of algorithms that must be available for its execution is determined. For this set, $A1$ is selected as the only algorithm subsuming A, and $B1$ is selected because $A1$ depends on it, even though $B2$ could be selected as well if dependencies between algorithms were not considered. Moreover, C is selected because $A1$ depends on it, even though act does not depend on C explicitly.

The proposed formalism allows to formulate the conditions of *functional integrity* of the whole system. The system is functionally integral when the following coherency conditions are true for all agent subsystems $AS \ni as = \langle Ag, Act, Alg, qr_1, \ldots, qr_m \rangle$:

$$\forall\, act \in Act\, [\, (\exists\, alg \in ALG : act \rightsquigarrow alg) \Rightarrow (\exists\, alg_c \in Alg : alg_c \preceq alg)\,] \tag{7}$$

$$\forall\, alg_1 \in Alg\, [\, (\exists\, alg_2 \in ALG : alg_1 \rightsquigarrow alg_2) \Rightarrow (\exists\, alg_g \in Alg : alg_g \preceq alg_2)\,] \tag{8}$$

i.e. for each action that may be performed in the agent system ($act \in Act$), if this action depends on algorithm alg, then an algorithm alg_c subsuming alg must be available in the system (i.e. must be present in the Alg set of the system). As was mentioned before, the subsumption relation "$\preceq$" is a partial order, and therefore alg_c can equal alg because $alg \preceq alg$. Similarly, for each algorithm alg_1 that is available in the AS, if alg_1 depends on alg_2, then an algorithm subsuming alg_2 (in particular, alg_2 itself) must be also available in the agent system ($\exists\, alg_g \in Alg$).

3 Functional Integrity of an Evolutionary Multi-agent System

The idea of an evolutionary multi-agent system (EMAS) was proposed as a particular technique of decentralized evolutionary computation [8,4]. The system consists of individual agents decomposed into several subpopulations (demes). Agents possess (possibly partial) solutions of the given optimization problem. They also possess a non-renewable resource called *life energy*, which is the base of a distributed selection process. Agents exchange their energy based on the quality of their solutions (fitness). Those which gather more energy have greater chances of reproducing, and those with low energy have greater chances of dying. This energy-based selection is used instead of classical global selection mechanisms, because of the assumed autonomy of agents. Agents may also migrate to another subpopulation if they have enough energy.

The model of EMAS which follows the concepts of [6] defines two types of agents:

$$TP = \{ind, isl\} \tag{9}$$

where *ind* denotes the type of an individual agent (as described above), and *isl* — of an aggregate agent, which is introduced to manage subpopulations of individual agents (an evolutionary island).

Consequently, at the top of the physical structure of EMAS there is a system of evolutionary islands:

$$as = \langle Ag, \varnothing \rangle \quad \gamma(as) = \varnothing \tag{10}$$

where:

$$Ag \ni ag = \langle id, isl, Nb \rangle \tag{11}$$

and $Nb \subset Ag$ is the set of evolutionary islands, which is used to define the topology of migration.

Every evolutionary island ag provides an environment for the population of individual agents:

$$\forall\, ag \in Ag \;\; \exists\, as^* = \langle Ag^*, Act^*, Alg^*, findAg, findLoc \rangle : ag = \gamma(as^*) \tag{12}$$

and an individual agent is defined as:

$$Ag^* \ni ag^* = \langle id, ind, sol, en \rangle \tag{13}$$

where:

$sol \in SOL$ is the solution of the problem (usually for optimization problems $SOL \subset \mathbb{R}^n, n \in \mathbb{N}$),
$en \in \mathbb{R}^+$ is the amount of energy gathered by the individual agent,
$ACT \supseteq Act^* = \{init, migr, get, repr, die\}$ is the set of actions available in as^*:
- $init$ – initialization of agent's solution,
- $migr$ – migration of an agent from one to another subpopulation,
- get – transfer of a portion of energy from one to another agent,
- $repr$ – creation of a new agent by two parents,
- die – removing of an agent from the system,

$ALG \supseteq Alg^* = \{rand, prep, eval, recomb, mut\}$ is the set of algorithms available in as^*,
$findAg : 2^{AG} \to \mathcal{M}(AG)$ is the query which allows to choose the neighboring individual agent (another agent present in the same system),
$findLoc : 2^{AG} \to \mathcal{M}(AG)$ is the query which allows to choose the neighboring island (using $Nb \subset Ag$).

An action of solution initialization *init* performed by $ag^* = \langle id, ind, sol, en \rangle \in Ag^*$ is defined in the following way:

$$Act^* \ni init = \langle ind, [\tau(sol) = 0], [\tau(sol) = 1] \rangle \tag{14}$$

where $\tau : SOL \rightarrow \{0, 1\}$ is a problem-dependent function, which indicates if a solution is initialized (by returning 1) or not (by returning 0). During this process a problem-dependent algorithm $prep \in Alg^*$ is used, and therefore: $init \rightsquigarrow prep$. Also, algorithm *prep* depends on another algorithm *rand*, necessary for generating random solutions. This is denoted as: $prep \rightsquigarrow rand$.

Actions *migr*, *get*, *repr* and *die* were defined in [6] and will not be discussed in detail here. However, several interesting observations follow. The action of energy transfer *get* is performed by an individual agent depending on the quality of its solution, which is given by problem-dependent algorithm *eval*. This relation may be defined as: $get \rightsquigarrow eval$. Similarly, the action of reproduction *repr* depends on the algorithms realizing recombination and mutation operators — *recomb* and *mut* respectively. This is denoted as: $repr \rightsquigarrow recomb$, $repr \rightsquigarrow mut$.

Application of EMAS to solving concrete optimization problems requires undertaking several decisions typical for the evolutionary approach, e.g. the choice of genotype representation and adequate variation operators. As an illustration one may consider EMAS with binary representation — it requires specializations (in terms of the subsumption relation) of algorithms *prep*, *eval*, *recomb*, *mut*:

$binPrep \preceq prep$ denotes an algorithm necessary to generate binary solutions,
$binEval \preceq eval$ denotes an algorithm which evaluates binary solutions,
$binRecomb \preceq recomb$ denotes an algorithm responsible for recombination of two binary solutions,
$binMut \preceq mut$ denotes an algorithm responsible for mutation of a binary solution.

All these elements must be introduced into *ALG* and Alg^*, and the latter will be defined as:

$$Alg^* = \{rand, binPrep, binEval, binRecomb, binMut\} \tag{15}$$

Such system definition satisfies equations (7) and (8) so it is functionally integral. According to the mechanism presented in the previous section, all algorithms are applicable without changing existing actions' definitions.

4 Component Techniques for AgE Environment

The model presented is the base for the design of the core of the computing environment *AgE*[4], which is developed as an open-source project at the Intelligent Information Systems Group of AGH-UST. A system implemented on *AgE* platform is composed of agents – the main functional entities, which represent the core logic of the computation [9]. Agents are further decomposed into functional units according to *Strategy* design pattern [10]. Algorithms, as presented in the model, are implemented in the form of strategies in *AgE*. Both agents and strategies can have properties (described in more detail in [6]), which can be either simple values or references to other entities in the system.

[4] http://age.iisg.agh.edu.pl/

Actions are implemented and executed as methods of external strategies classes or the parent agent class, which represents the agent's environment. During the execution of these methods other strategies can be used to perform different activities within the action. Therefore, the dependency (described by the relation "$\rightsquigarrow$") between actions and algorithms is represented in *AgE* as a reference property to the required algorithm, preceded by @Inject annotation.

Dependency between algorithms introduced in the model maps to dependency between strategies. In different cases this relation can be represented in two ways in *AgE*.

Let us consider dependent algorithms A and B ($A \rightsquigarrow B$) represented in AgE by Java classes `A` and `B`. In the first case strategy `A` directly uses strategy `B` and the dependency relation is realized in the same way that dependency between actions and algorithms, i.e. by using a reference property marked with @Inject annotation. In the next case, strategy `A` does not directly use `B`, but `B` is required for proper processing of `A`. For example, a binary mutation strategy needs a binary initialization strategy which generates the proper type of solutions, although the initialization is not directly used by mutation. To define this kind of dependency, class `A` must be preceded by annotation @Require(B.**class**).

An example code of a strategy with dependencies is shown below:

```
@Require(C.class) public class A {
   @Inject @PropertyField(name="b")
   private B b;
}
```

Strategy `A` directly uses strategy `B` and requires strategy `C` for proper processing. Therefore it defines two dependencies, the first by an annotated reference property (for strategy `B`) and the second by adding @Require annotation to class definition (for strategy `C`).

AgE was designed with the emphasis on achieving the main advantages of component-oriented techniques, i.e. independent development, reusability and elementary contracts [7]. Its realization is vastly supported by dependency injection pattern, an implementation of the inversion of control paradigm proposed by Robert Martin [11] and later popularized by Martin Fowler[5], and by utilizing the freely available PicoContainer framework[6].

Component-oriented techniques are exploited to implement the process of automatic assembly of different agents structures with dependant strategies. The input configuration (in XML file format) together with agents and strategies classes with described annotations are used to initialize computing environment and provide appropriate instances in runtime.

The approach allows for creation of fully-initialized components, with all dependent components injected. Moreover, late binding by a container allows for runtime injection which facilitates third-party development of the components. The annotations describing component dependencies, as presented above,

[5] http://martinfowler.com/articles/injection.html
[6] http://www.picocontainer.org

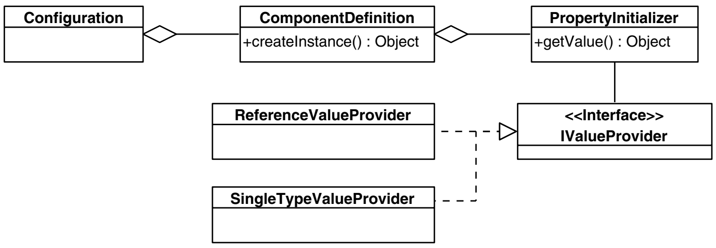

Fig. 1. Configuration model

together with class's public methods treated as component's operations, may be perceived as a requirement closely related to component contracts as proposed by Szyperski [7].

The decision which component should be provided to the other one is made during instantiating particular components based on the configuration model shown in Fig. 1. The functional integrity of the system as defined by (7) and (8) is ensured by verifying the configuration and components requirements – dependencies are processed in order to check if all required components are available in the classpath, no conflicts occur between components, and all requirements are fulfilled. Verification is also performed during each request for instantiating a component, which ensures that no inconsistent unit will be created.

The configuration model describes agents and strategies by defining a set of `ComponentDefinition` objects, aggregated in one `Configuration`. Each definition can have a list of `PropertyInitializer`s which – each assigned to an agent's or strategy's single property – are responsible for their initialization. To facilitate this initialization AgE uses `IValueProvider` objects, which are responsible for providing concrete values to properties. These values can be references (represented by `ReferenceValueProvider`s) or simple type values (represented by `SingleTypeValueProvider`s).

The process of system initialization is divided into two steps. In the first step, the configuration model is created from an XML file with a well-defined structure[7] and a process of verification is executed.

In the next step, a fully initialized system is created, containing agents with already associated strategies. The advantage of this solution is that one implementation of an agent can perform actions which use different realizations of strategies. Therefore a wide variety of possible computations can be performed without changing an agent's class, i.e. without recompiling the system. One thing remains to be done, i.e. to change the XML configuration file.

After being built from an XML configuration file, component definitions are registered in IoC containers (class `PicoContainer`) using special adapters of class `CoreComponentAdapter`, both shown in Fig. 2. A request to create a hierarchy of components (agents and strategies) is directed to an IoC container, which in turn delegates the creation of specific components to adapters assigned to component

[7] http://age.iisg.agh.edu.pl/xsd/age-2.3.xsd

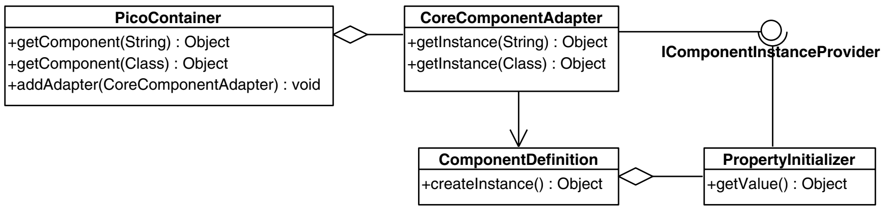
PicoContainer
+getComponent(String) : Object
+getComponent(Class) : Object
+addAdapter(CoreComponentAdapter) : void
CoreComponentAdapter
+getInstance(String) : Object
+getInstance(Class) : Object
IComponentInstanceProvider
ComponentDefinition
+createInstance() : Object
PropertyInitializer
+getValue() : Object

Fig. 2. Dependency Injection pattern in AgE

definitions. In the method `ComponentDefinition.createInstance`, the definition creates an instance of a component it describes. This method is responsible for properly initializing all the component's properties, both simple values and its dependencies on other components. In order to initialize the dependencies correctly, the definition can retrieve (via the `IComponentInstanceProvider` interface) an instance of a required component (either by name or by type) from the IoC container associated to it. The retrieval of required values and assignment to component properties is done by specialized `PropertyInitializer` objects, one for simple type values and one for references to other components.

5 Conclusions

The formalism proposed in the paper is solely used for design, and that is why such details as the precise definition of the system state space or state transition functions were not taken into consideration. The notion of algorithms and dependency relations (imposing the existence of algorithms required by actions and other algorithms) were introduced to define functional integrity conditions for the system. So far the model can be fully mapped to the concepts present in *AgE* computing environment. Also component-based approach proved to be a convenient and flexible technique supporting the assembly of a particular system according to the provided configuration, and its validation with respect to the proposed functional integrity rules. Moreover, a prototype graphical configuration editor was created based on the presented solution. It uses the proposed techniques for suggesting available assembly options according to components contracts.

Further research should allow to extend the model to cover other computation intelligence techniques based on agent paradigm, such as iEMAS (immunological evolutionary multi-agent system) [12] or HGS (hierarchical genetic search) [13]. The mapping between the proposed formalism and existing models describing these techniques will be provided. The current focus of *AgE* development is to fully implement the verification logic allowing to check the integrity rules for delivered components. Also, because the implementation of AgE framework continues, the formalism will surely be updated in the near future.

Acknowledgments. The authors would like to express their gratitude to Mr. Tomasz Kmiecik, a former student of AGH-UST, for his invaluable contribution to the design and implementation of dependency injection mechanisms in AgE.

References

1. Cetnarowicz, K., Dobrowolski, G., Kisiel-Dorohinicki, M., Nawarecki, E.: Functional integrity of mas through the dynamics of the agents' population. In: Proc. of 3nd Int. Conf. on Multi-Agent Systems (ICMAS 1998). IEEE Computer Society Press, Los Alamitos (1998)
2. Jamont, J.P., Ocello, M.: Using self-organization for functional integrity maintenance of wireless sensor networks. In: Proc. of the International Conference on Intelligent Agent Technology IAT 2003, Washington, DC, USA. IEEE, Los Alamitos (2003)
3. Bergenti, F., Gleizes, M.P., Zambonelli, F.: Methodologies and Software Engineering for Agent Systems. Kluwer Academic Publishers, Dordrecht (2004)
4. Kisiel-Dorohinicki, M.: Agent-oriented model of simulated evolution. In: Grosky, W.I., Plášil, F. (eds.) SOFSEM 2002. LNCS, vol. 2540, pp. 253–261. Springer, Heidelberg (2002)
5. Krutisch, R., Meier, P., Wirsing, M.: The agentComponent approach, combining agents, and components. In: Schillo, M., Klusch, M., Müller, J., Tianfield, H. (eds.) MATES 2003. LNCS (LNAI), vol. 2831, pp. 1–12. Springer, Heidelberg (2003)
6. Byrski, A., Kisiel-Dorohinicki, M.: Agent-based model and computing environment facilitating the development of distributed computational intelligence systems. In: Proc. of the Computational Science - ICCS 2009, 9th International Conference, Baton Rouge, U.S.A., May 25 - 27, 2009. Springer, Heidelberg (2009)
7. Szyperski, C.: Component Software: Beyond Object-Oriented Programming. Addison-Wesley Longman Publishing Co., Inc., Boston (2002)
8. Cetnarowicz, K., Kisiel-Dorohinicki, M., Nawarecki, E.: The application of evolution process in multi-agent world (MAW) to the prediction system. In: Proc. of 2nd Int. Conf. on Multi-Agent Systems (ICMAS 1996). AAAI Press, Menlo Park (1996)
9. Kisiel-Dorohinicki, M.: Agent-based models and platforms for parallel evolutionary algorithms. In: Bubak, M., van Albada, G.D., Sloot, P.M.A., Dongarra, J. (eds.) ICCS 2004. LNCS, vol. 3038, pp. 646–653. Springer, Heidelberg (2004)
10. Gamma, E., Helm, R., Johnson, R., Vlissides, J.: Design patterns: elements of reusable object-oriented software. Addison-Wesley Professional, Reading (1995)
11. Martin, R.C.: The dependency inversion principle. C++ Report 8(6), 61–66 (1996)
12. Byrski, A., Kisiel-Dorohinicki, M.: Immunological selection mechanism in agent-based evolutionary computation. In: Klopotek, M., Wierzchon, S., Trojanowski, K. (eds.) Proc. of the Intelligent Information Processing and Web Mining IIS IIPWM 2005, Gdansk, Poland. Advances in Soft Computing. Springer, Heidelberg (2005)
13. Schaefer, R., Kołodziej, J.: Genetic search reinforced by the population hierarchy. Foundations of Genetic Algorithms 7 (2003)