Simulation of Artificially Generated Intelligence from an Object Oriented Perspective*

Bálint Fazekas and dr. Attila Kiss**

Department of Information Systems, Faculty of Informatics, Eötvös Loránd University, Budapest, Hungary
bfazekas@inf.elte.hu, kiss@inf.elte.hu

Keywords: computational intelligence, artificially generated intelligence, evolution, simulation, agents, object oriented programming

Abstract. The aim of this paper is to outline several different approaches of how the evolution of intelligence can be observed in agents, which are placed inside a given environment. We fully explain a method that relies strongly on current object oriented programming paradigms and technologies, and show a complete explanation of our implementation of the chosen model. In the first parts of the paper, we point out the important characteristics of intelligence and what is to be measured. Then, we introduce ideas that can be used to model the simulation of artificial intelligence evolution. During the explanation of the models, we point out both the strength and the weaknesses of the models. We then move on to a guide of our implementation, then show and explain the results of the simulation. Lastly, we suggest further possible improvements for our approach.

1 Introduction and Ideas

The aim of this project is to observe an entity – or a given set of entities – placed within a defined environment, while the entity observes its surroundings, tries to evolve, and deduce certain rules of the objects in its environment.

1.1 Refined description

First, it is important to clarify that the scope of this paper *does not* include or consider the physical development of the entities that observe their environment – in other words, undertake any sort of physical evolution. The primary aim of this paper is strictly to concentrate on analyzing the intelligence, logical deductive capabilities, and environment observation of the given entities – from here on, which we will refer to as ***agents***.

*This project was supported by the European Union, co-financed by the European Social Fund (EFOP-3.6.3-VEKOP-16-2017-00002)

**A. Kiss was also with J. Selye University, Komarno, Slovakia

Despite the fact, that researches, theories, and algorithmic methods – that had well–tried in the open field – related to artificial intelligence had been around for decades, the topic of this paper is on a higher level of abstraction of theoretical research, than that of artificial intelligence. In contrast to the fields of artificial intelligence, the aim of this paper is to observe whether agents are capable of deduce, logically separate (or classify), and use the objects in its environment for its own purposes. We assume that the agents are given a set of observatory senses (such as light sensors), and that the environment is a logically built, non–physically represented environment.

Definitions This section contains definitions which "cannot very well be defined" scientifically. This problem is present due to the fact that the current state of definitions related to psychological terms and behavioral observations are quite subjective in their nature, and therefore hard to define by default. The difficulty of constructing these definitions is also caused by our extremely limited capabilities to get a full map and process–diagram of a person's thought–process. In the real world, the human brain works slightly different for each individual, and therefore develop differently. Hence, in the case of every individual, we would deal with a non–deterministic, and structurally deviant system. Obviously, the complexity of building such system that is capable of handling many different sub–systems is too large for the purposes of this paper.

The following definitions were inspired by a one–hour long interview with Christof Koch, who is a famous and notorious German–American neuroscientist. Christof Koch has created countless models and publications regarding the *relationship between human cognition and neuron networks*. The interview is available for free to watch on one of MIT's great professors, Lex Fridman's YouTube channel on the following link: https://www.youtube.com/watch?v=piHkfmeU7Wo.

Consciousness: In this paper, we define consciousness as one's ability of self–reflection. By self–reflection, we mean the expression of how well a being can separate, or distance itself from its environment and from other being in its environment, and most importantly *is capable of developing these skills by itself.*

Intelligence: We define intelligence as the ability to create relationships between the "known information" of an individual.

Experience: In this context, experience is defined as an agent's ability of how well, and what can it sense within its given environment.

Object: Object is defined as a set of attributes and functions that can be executed on that object. The parameters of the functions of an object can be of arbitrary type and amount, as an agent has no knowledge of these function in the beginning.

Object–space: Object–space is a physical space within the memory or disk of a computer, that stores all the objects that we intend to use in the environment. Agents that observe the environment are not part of the object–space.

Environment: The environment is an interpretation (or representation) of the object–space that an agent is able to communicate with.

Agent: An agent is an actor that can observe, sense, communicate, and have an impact on its environment.

Our aims with this project can be summarized by observing the *cognitive emergence* of intelligent agents. [1] [5]

2 Model creation

Based on the definitions above, it is not a straightforward task to define a model which is adequate for simulating intelligence–evolution. For this, a necessary amount of creativity and heuristic approach is needed, which can precisely fit for our purposes, but at the same time, neither restrict the problem at hand, and our ability to implement it.

During the planning of a model, we tried to come up with several models that fit in the scope of our definitions. These models were developed considering different mathematical approaches and several services provided by certain programming languages.

First, let us take a look at an approach that would be good for running a simulation, but otherwise very bad when it comes to implementation and analysis.

2.1 Analytic and functional approach based on mathematical models

For this approach, we tried to build a system from a purely mathematical, physical, and analytic point of view. In this model, the *environment* is a static object that contains several physical objects. We can think of this environment as Space, or universe that has a defined set of rules, and every object contained are subjected to these rules. For the sake of simplicity, we can say that the environment does not alter its objects, and that the objects do not have an impact on each other. This environment is simply a container for the physical objects that we define. In this sense, this environment is the same as our known universe, and the rules can be interpreted as the mathematical laws that we know today.

Each object in the environment has a simple identification attribute (for example, a name), a category list, and a set of executable functions.

The only purpose of the identifier name of an object is so we can reference the object in the environment.

Given a set of classified objects at the beginning, we expect our agents to be able to process and further classify new objects. Here, we can make two choices about the starting object set: we could leave it empty, and leave every classification process to the agents. The other approach is to create a few labeled objects, which observed by the agents, can guide the agents how to classify, and how to interpret the basic labels.

Naturally, here we can also chose to create such a *set of basic objects*, which contains labeled basic objects (such as: "door", "button", "box", etc.). One characteristic of these basic objects, is that they are fully labeled, meaning that the agents know everything about these objects; in other words, the agents have access to all attributes and methods of these objects. Outside of the set of basic objects, we can define a set of regular objects, which contain only partially labeled objects, and analyze how the agents observe (classify) these partially labelled objects.

As we previously outlined, an object has a list of methods. These methods can be executed on that given object. Each of these methods can be interpreted as a mathematical function, more precisely, an analytically solvable function. Each function can alter the inner state (attributes) of an object, and the agents can query the change of state of that object. However, it is not entirely clear what kind of mathematical function is defined within a method, and what the actual function is trying to solve.

Let us take a look at a complex, mathematical model.

Suppose that our agent would like to wash its hands in a washroom. To reach the sink in the washroom, it must first get to the door of the washroom, and must open it, without having any knowledge of such object. After opening the door, the agent must find the sink, and wash its hands without knowing how the sink actually works. To solve this problem, the agent could try to find all methods of the sink object, and "bombard" the found functions with arbitrary types and arbitrary number of parameters, and try to find a method, that accepts these parameters, and results in a desired outcome. The agent will know that it found the right method for washing its hands by querying the change of state of the object after giving it a certain parameter list. After finding the desired method, the agent must undergo an optimization routine, which will find a reasonable value for the function's parameter, to get an adequate result. The actual process of solving the problem can only begin after all of these have been found out by the agent.

This small example demonstrates that even this small problem requires a lot of planning and technologically interesting fields, both from the agents' and the developers' perspective. This approach is further complicated by the aim of this research, in which we try to observe the evolution of an agent's intelligence. This means that after each step, the agent must classify the observed objects, and try to dynamically create newer classifications while the simulation is running. From the perspective of the implementation, this means that the program must be able to create new data types during run-time, and use the created types. However, the purely mathematical approach requires to dynamically create iso-

morphic relations between data types, the problems we wish to solve, and the mathematical functions.

In order to avoid this great complexity, we propose another model, which is easier to define and implement.

2.2 Object oriented approach using discrete values

In the previous section we saw that the mathematical approach introduces too many complications to our goal. In order to avoid this, we chose a different approach.

In our discrete model, let the *environment* be a container, that holds all the objects and the agents. In this model, we can define a new type objects that have attributes which change with the passing of time. We call this model the *evolutionary environmental model*. In the evolutionary environmental model, we have to account for the objects impact on each other, as well as the passing of an arbitrary unit of time. In this model, we measure the unit of time in *the amount of steps* taken since the beginning of starting the simulation. One step includes all the agents observing some of the objects in the environment. With each step, the step–counter increases, and some objects may change their inner states regarding the counter.

For example, assume that our environment contains an *iron door*. We know that a property of iron is that it rusts if it is exposed to oxygen for long enough. This iron door object might have a function that determines the force which is needed to open it. In the beginning, if the iron is completely fresh, then it will be much easier for an agent to open it – in other words, it can apply a much lower force to open the door. However, after some steps, the door may start to "rust", causing the increase of minimum force to open it. Therefore, we can see that this object will change its inner state without an agent's interaction.

The purpose of the evolutionary environment is to propagate all of its contained objects forward in time. An agent can observe its environment only if all the environmental objects are within the same evolutionary step.

We denote the evolutionary environment with Γ, and denote each discrete state of the environment with Γ_0, Γ_1, ..., Γ_k ($k \in \mathbb{N}_0$). Furthermore, let us denote a step between these discrete state as $\Gamma_k \longmapsto \Gamma_{k+1}$. For several steps, we use the following denotation: $\Gamma_k \longmapsto^* \Gamma_{k+m}$, ($k \in \mathbb{N}_0, m \in \mathbb{N}_0$).

During the execution of the simulation, we would like the agents to deduce the attributes and functions of the objects. Consequently, it is not enough for the agent to observe the objects in the environment, hence practically that would mean that the agent knows everything about that particular object (attributes and methods). This problem can be solved by introducing a new space, where the objects are held.

We place the well defined objects in the *object–space*. We denote the object-space with Ω, and the objects within the object space with ω_1, ω_2, etc. ($\omega_1, \omega_2, .. \in \Omega$)

After we have placed the desired objects in the object-space, we *map* the object-space to an *object representational table*, which stores all the environment

objects as serialized strings in a table. This representational table is the *medium* between the agents and the actual object, and it only represents the information about an object that has been "discovered" by an agent before. We denote the *representational table* with Σ.

In the previous section, we mentioned that it is advised to include base-objects in the environment, which play guiding roles for the agents. The purpose of these base-objects is especially important is this case, since these object are the ones that define how other objects will be interpreted. We understand base-objects as *static objects in the environment*, therefore we denote these objects with κ_1, κ_2, etc., and the set of base objects with K. Clearly, we can see that K $\subseteq \Omega$, and $\forall \kappa \in K : \kappa \in \Omega$.

This object oriented approach that we described above, is very similar to ideas that are outlined in two other papers.[10] [11]

Essentially, our model comes very close to a *abstract state machine*, where the agents represent the changing states within the state machine, and the environment provides the input for the agents.[14]

2.3 Attribute assignment to objects

One of the basic characteristics of intelligence is that the "being" is capable of *categorical thinking*. In other words, the being is able to categorize the objects experienced in its environment, and use the objects according to their categories.

These classifications are not straightforward, and the observed attributes have a great impact on how an agent will classify an object. During the simulation, we would like to achieve for each agent to be able to create their own categorical system for the objects that they observed. During the simulation, we would like to see that the agent's knowledge about the environmental objects get more and more specific with each passing step, consequently, the category-attribute of that object get specific. This is only possible if the agents observe the same object in several different Γ states.

2.4 Comparison with general neuron–networks

There are several questions that can arise from the models explained above. For example, why is it necessary to create these models in the first place, when there are already well defined methods in the field of artificial intelligence, which can optimize any type of mechanically implementable algorithm; and, how is our model actually different from neuron–networks?

First, let us take a look at the well defined methods in the fields of artificial intelligence. One of the elements of creating an AI algorithm is to define the problem on a higher level of abstraction such way, that the problem can be easily mapped to a problem-graph. The starting node of the graph indicates where we initially start our simulation, and we can move to different nodes on the graph based on a set of rules. From these algorithms, we expect to reach a node on the graph, that results in the solution of the problem. There are

many different variations of these *graph search* algorithms, they try to find the solution as optimally as possible – if it exists. However, it is also a possibility that our problem does not have a clear solution. In this case, we are forced to introduce *heuristic approaches* in the algorithms, which help the algorithm in case it gets stuck in a state, or a sequence of states. These heuristic ideas are always defined by the given problem, and often cannot be generalized. During our simulation, and the agents' observation of their environment, we will work with many random numbers, which would greatly complicate our ability to define a problem-space, without it actually benefiting the processes of the simulation. We would like to observe whether the machines are able to create or initialize a hierarchical observation system. Therefore, even if the *graph search* algorithms would be good if we had a clear, mechanical algorithm, this approach does not help much in our case.

Now let us explain why or model is different from neuron–networks. Neuron-networks are used in cases, where we want to teach a program to solve a problem on a general level. The learning process can be guided and unguided. In the case of guided learning, we tell the program what the outcome (result) is, and we expect the program to adjust its parameters if the calculated result is not the expected. By storing these parameters gained, the more sample data we feed the learning process, the more likely our program will be able to solve the given problem outside of its learning phase. The samples given to the program has a great impact on each step of the learning process. For example, if we only show left turns to the automatic steering software of a vehicle, then the vehicle will have almost no chance of being able to correctly take a right-turn. First, it would seem that this problem of learning can be done by giving a relatively huge number of samples of all possible cases that the software could encounter. However, in that case, the software is subjected to the fault of *over-learning*, which means that the software will account for parameters in certain cases, which would be completely irrelevant. In the majority of the cases, this leads to worse results than that of a program, which was given less, but more general samples.

Therefore, we must discard the guided learning for the following reasons: firstly, we cannot tell the agents what outcome we are looking for, since this is what we are essentially trying to obtain. (We hope to see results, which reflect the categorical thinking of a human, however this requirement is very hard to ensure due to the lack of intervention.) On the other hand, the sample data given to the learning process must help in the categorization process, however, the samples do not have any precedence, therefore there is no real "good" sample that we could supply the learning process.

Our model is much more similar to the unguided learning process. In this case, we leave the categorization process to the agents, and we only interfere by telling the agents whether a calculated value is adequate or not. The problem of us not being able to give an adequate set of sample data is still present in this case.

3 Implementation

In the previous chapter, we detailed the models that can be, and should be used for our purposes. In this chapter, we explain how we implemented the *object–oriented* approach.

Even though the object–oriented programming paradigms are well known, it is most appropriate to mention an amazing work by *Zeigler, Bernard P.* about how agent based simulations should be planned, and how the communication between the components flows.[15]

Another fundamental work about object oriented simulation was digested when we planned our implementation.[6] Ideas about implementing simulation systems were inspired by an article written by Isabel A. Nepomuceno-Chamorro[12].

To implement our model, we require the means to reference an object by its name, and query all its attributes and function in run-time. The JAVA programming language is a very strongly object-oriented language, which is beneficial for defining the *environment* and the *environmental objects*. The JAVA 8 (version 1.8), the language supports a service that is so called *reflection*. *Reflection* allows us to extract every information (including attributes, methods, and method parameters) about an object. We can exploit this service to give our agents a sense of "intuition". The JAVA Enterprise Edition (JAVA EE) includes services that support database connections and persistent data management, which is a convenient way for us to store our *environmental objects*, without having to worry about extensive memory usage. With the so called *JPA – Java Persistence API –*, we can *annotate* our objects to define exactly which database and in which schema we would like to store it. By persisting our objects in databases, we will know exactly where to look for them, and more importantly, the memory of the running simulation will only contain the agents, the representational string table, and the object which is currently observed by an agent. The details of the modern Java JPA (also used in our implementation) can be read in a book by M. Keith and M. Schincariol. [8]

However, one problem arises from persistently storing our objects in the database: if an object is an *evolutionary environmental object*, then we expect the object to undergo some inner change of state as we propagate through the steps of the simulation – even if the object is not observed by an agent (therefore, have not been pulled into the memory of the program). We solve this problem by storing the step-counter, in which a given object was last observed by an agent. When the object is observed again, the step-counter is refreshed and the new inner state of the object is calculated. After we calculate the new inner state, we pass the object to the agent for observation.

3.1 Environment implementation

As mentioned before, the environment is responsible for the maintenance of the communication between the agents and the environmental objects. Since we want to store our environmental objects in a database, we need some sort of communication pipe between the agents and the actual objects.

This communication channel is very similar to the message driven concept in P–colonies, where each side has a *sender* and a *receiver* method. [4] [2]

The environment is therefore responsible for receiving the requests from the agents, and send these requests forward to the objects in the object-space.

Beside support the means of communication, the environment has another important task. Our environment should also be responsible for querying the database, and create the representational string table from the queried objects in the object-space. We use *Java Run-time Reflection* to extract all information about the queried objects, and *serialize* these object in the string-table, based on a string representation of the objects.

The reason why this is beneficial for us, is because in the future, we can tell exactly which data in the representational table can be interpreted by the agents.

3.2 Environmental Objects

We implement the environmental objects using Plain Old Java Objects (or POJOs). These object defining classes have the characteristic of having *getters* and *setters* for all of their attributes, in a very well defined format. Let us look at an example, or what we mean by "well defined format". Suppose our object has an attribute called "id". For this attribute, the getter function would be called "getId" and the setter would be called "setId".

Each object class can have numerous attributes. Rather than writing the getters and setters for each attribute, and expanding the code, we use the LOMBOK Java library, which dynamically generates these methods for us. This is also a much safer method of doing so, because it eliminates the possibility of mistyping these methods.

To identify an object class as a persistable entity, we use the annotations included in the Java Persistence API. We use "@Entity" and "@Table(schema = "mySchema", table = "object")" annotations to tell which table and schema we want to persist the object in.

3.3 Agents

As outlined in the previous sections, the agents are objects which can observe (read) its environment, and occasionally change the inner state of the objects via sent messages. We implement our agents such that they have a *list of known categories*, which stores the type of objects the agent has encountered before.

The agents also have *sender* and *receiver* methods, which are generic in a sense that they are capable of sending and receiving different type of objects. The sender function send a list of types and values, and will try to call a function of the object with the parameters given in the list. The purpose of the receiver function is to receive the result of the function, and to interpret the result.

3.4 Implementation with results

We first show the results of the progression of the implementation. For the early tests, we used one type of entity object called *Block*. A *block* is an abstract object with very simple, observable characteristics, such as dimensions, weight, and the ability to be moved. The following Java class defines this object.

```
@Data
@Entity
@Table(schema = "WOAF", name = "BLOCK")
public class Block {

    @Id
    @GeneratedValue(strategy = GenerationType.AUTO)
    @Column(nullable = false)
    private int id_;
    @Column(nullable = false)
    private int step_ = 1;
    @Column(nullable = false)
    private float width_;
    @Column(nullable = false)
    private float height_;
    @Column(nullable = false)
    private float depth_;
    @Column(nullable = false)
    private boolean groundAttached_;
    @Column(nullable = false)
    private boolean hollow;

    public Block() {
    }

    ...
}
```

As we mentioned in the previous section, the annotations above the class definition help the Java compiler and JPA to sort out what to do with the class. The @Data annotation automatically generates the getters and setters for all fields, without manifesting the code itself in the source file. The reason we decided to put an underscore character after each property element, is to avoid accidental name clashing with SQL names. We will use this convention throughout all of our objects.

We used the @Table annotation to tell which database schema and in which table we wish to store our object.

In order to initialize our database – which is essentially our environment with the known objects – we instantiated a few objects of the Block class, and stored the objects in the database.

```
private static void fillDB() {

    List<Object> objects = new ArrayList<>();

    Block b = new Block(10, 20, 15, true, true);
    Block b2 = new Block(40, 20, 30, false, false);

    objects.add(b);
    objects.add(b2);

    EntityManagerFactory emf =
        Persistence.createEntityManagerFactory("ReflectionPU");
    EntityManager em = emf.createEntityManager();
    em.getTransaction().begin();

    for (Object object : objects) {
    em.persist(object);
    }

    em.getTransaction().commit();
    em.close();
}
```

Luckily, we found out that using a list of *Object* typed objects is adequate for storing initializing entities, since the entity annotations will make sure that each entity will be persisted in their corresponding tables. This is especially useful, since we have the ability to store every initializing object in one array, and persist them using one loop.

After we the generation of the basic environment variables and persisting them in our database, we moved on to how we can query these object from a clean, empty environment, without ever mentioning the type of the object which we would like to query.

In order to do so, we first query the names of the tables that are present in our database.

```
private static List<String> setupDatabaseConnection() {
    Connection con;
    String driver = "org.apache.derby.jdbc.EmbeddedDriver";
    String dbName = "//localhost:1527/Reflection;user=woaf;password=123";
    String connectionURL = "jdbc:derby:" + dbName;
    List<String> tableNameList = new ArrayList<>();

    try {
        Class.forName(driver);
        con = DriverManager.getConnection(connectionURL);
        ResultSet sets = con.getMetaData().getTables(null, "WOAF", "%",
            null);
        while(sets.next())
```

```
        {
            if(!sets.getString(3).equals("SEQUENCE"))
                tableNameList.add(sets.getString(3));
        }
        con.close();
    } catch (SQLException | ClassNotFoundException ex) {
        Logger.getLogger(Main.class.getName()).log(Level.SEVERE, null,
            ex);
    }
    return tableNameList;
}
```

This method returns the names of all the tables that are present in our database. However, there is one problem with the return values. Due to conventional reasons, we decided to give all–capital letter names to our tables, but in our queries, we would have to refer to our tables with only the first letter of the name being capitalized. For example, we would write "Block" instead of "BLOCK". The conversion of each name in the result list can be done easily.

In the next few lines of code, we demonstrate how we managed to get all the persisted *Block* objects from the database, without referring explicitly to the Block table or type.

```
public static void main(String[] args) {

    fillDB();
    List<String> tables = setupDatabaseConnection()
            .stream()
            .map(s -> toCamelCase(s))
            .collect(Collectors.toList());
    EntityManagerFactory emf =
        ersistence.createEntityManagerFactory("ReflectionPU");
    EntityManager em = emf.createEntityManager();
    Query q = em.createQuery("SELECT b FROM " + tables.get(0) + " b");
    q.getResultList().forEach(System.out::println);
}
```

Of course, we can iterate through all of the table names, and query the stored objects of each type.

Running the program will give us the following results.

```
Block(id_=2, step_=1, width_=40.0, height_=20.0, depth_=30.0,
    groundAttached_=false, hollow=false)
Block(id_=1, step_=1, width_=10.0, height_=20.0, depth_=15.0,
    groundAttached_=true, hollow=true)
```

As we can see, we managed to get our stored entities without referring to their types.

In the following lines of code we demonstrate how we can extract all of the information about any given class (contained within the names of the tables), including function names, parameter numbers and types, method modifiers, and data members.

```
tables.forEach((name) -> {
    try {
        Class<?> arbitraryClass = Class.forName("base." + name);
        List<Constructor<?>> constructors =
            Arrays.asList(arbitraryClass.getDeclaredConstructors());

        System.out.println(constructors.toString());
        Object o = constructors.get(0).newInstance();

        List<Field> oFields =
            Arrays.asList(o.getClass().getDeclaredFields());
        List<Method> oMethods =
            Arrays.asList(o.getClass().getDeclaredMethods());

        System.out.println("Fileds:");
        oFields.forEach(System.out::println);

        System.out.println("Methods:");
        oMethods.forEach(System.out::println);

    } catch (ClassNotFoundException ex) {
        Logger.getLogger(Main.class.getName()).log(Level.SEVERE, null,
            ex);
    } catch (InstantiationException ex) {
        Logger.getLogger(Main.class.getName()).log(Level.SEVERE, null,
            ex);
    } catch (IllegalAccessException ex) {
        Logger.getLogger(Main.class.getName()).log(Level.SEVERE, null,
            ex);
    } catch (IllegalArgumentException ex) {
        Logger.getLogger(Main.class.getName()).log(Level.SEVERE, null,
            ex);
    } catch (InvocationTargetException ex) {
        Logger.getLogger(Main.class.getName()).log(Level.SEVERE, null,
            ex);
    }
});
```

Here, we exploit the use of Java Reflection that, given a specific name, can observe classes during run-time that has been loaded in the JVM. As it can be seen above, we have added the string "base." to the beginning of each table name. The purpose of this is for Reflection to be able to find the class itself, which is inside the designated package called "base".

This previous section of code gives the following results.

```
[public base.Block(), public
    base.Block(float,float,float,boolean,boolean)]
Fileds:
private int base.Block.id_
private int base.Block.step_
private float base.Block.width_
private float base.Block.height_
private float base.Block.depth_
private boolean base.Block.groundAttached_
private boolean base.Block.hollow
Methods:
public boolean base.Block.equals(java.lang.Object)
public java.lang.String base.Block.toString()
public int base.Block.hashCode()
public boolean base.Block.isGroundAttached_()
public void base.Block.setGroundAttached_(boolean)
public int base.Block.getId_()
public void base.Block.setId_(int)
public void base.Block.setWidth_(float)
public float base.Block.getHeight_()
public void base.Block.setDepth_(float)
public float base.Block.getWidth_()
public void base.Block.setHollow(boolean)
public float base.Block.getDepth_()
public void base.Block.setStep_(int)
public int base.Block.getStep_()
public boolean base.Block.isHollow()
public void base.Block.setHeight_(float)
protected boolean base.Block.canEqual(java.lang.Object)
```

As we can see above, we were able to extract all of the data members and methods of the given (arbitrary) class. Since the result is not in the form of Strings, but Fields and Methods, we have absolute knowledge about the modifiers, the return types, the name, the parameter count, and the parameter types. There are a few methods (such as "equals", and "toString") which are unnecessary in our case, so we will try to filter the methods to only the types of methods that relate to the actual query of the object's data members. Using this extracted collection of information, we can create the *environment representational table* that we explained in the previous chapters of this paper.

The idea of creating the representational string table was to give the sensation or the illusion of the agents having some sort of intuition. By reading from the string table, the agents could extract the name of a class and a method, and based on the methods explained previously, would instantiate that certain object and use it's chosen method. For the sake of simplicity, we will skip implementing the string representation for each element, and define a container class that will store the class information in arrays.

We move on to defining an *Agent* with the minimal amount of knowledge needed. Our agents reflect their knowledge by storing three major lists: one is for object constructors, one is for the object fields, and on is for the object methods. These three lists come in a package called *ClassWrapper*. In our main program, we added a new list, that stores the information gathered from creating classes with the found table names. We then passed this list to our agent. Keeping in mind, that the *Block* class is a base object, so the agent has all rights and means to know every method and field of that object. Here we present our code. The result is the equivalent that was shown earlier, with the exception that now the results are shown by the agent.

```
Agent a = new Agent();
a.setKnownClasses_(resultWrappers);
for(int i = 0; i < a.getKnownClasses_().size(); i++){
    System.out.println("Constructors:");
    a.getKnownClasses_().get(i)
        .getClass_constructors().forEach(System.out::println);
    System.out.println("Fields:");
    a.getKnownClasses_().get(i)
        .getClass_fields().forEach(System.out::println);
    System.out.println("Methods:");
    a.getKnownClasses_().get(i)
        .getClass_methods().forEach(System.out::println);
}
```

Now, let us introduce a new indicator inside the class wrapper, which will tell us whether the extracted class (from the database) is of an evolutionary or of a base object. Since our model only includes base and evolutionary objects, we indicate this attribute as a Boolean data member. This object attribute is to be determined when we try to find the matching class for the name of a table from the database.

In our project hierarchy, we put the base classes in the "base", and the evolutionary classes in the "evo" package. Therefore, any of our objects can be found in base.⟨ClassName⟩ or evo.⟨ClassName⟩ path.

Since we want our Agents to have limited knowledge about the evolutionary objects, we restricted the amount of information gathered from the classes. We decided to store only the constructors of these objects. Based on the parameters of the non–empty constructor, we can make the agents deduce certain properties of the evolutionary objects. We implement this in the class extraction segment with the following structure.

```
Class<?> arbitraryClass = Class.forName(path + name);
ClassWrapper classWrapper = new ClassWrapper();
classWrapper.setClass_constructors(
    Arrays.asList(arbitraryClass.getDeclaredConstructors()));
if(path.equals("base.")){
    classWrapper.setEvolutionary(false);
    Object o = classWrapper.getClass_constructors().get(0).newInstance();
```

```
    classWrapper.setClass_fields(
        Arrays.asList(o.getClass().getDeclaredFields()));
    classWrapper.setClass_methods(
        Arrays.asList(o.getClass().getDeclaredMethods()));
}
else {
    classWrapper.setEvolutionary(true);
}

resultWrappers.add(classWrapper);
```

For our evolutionary object, we created a very simple model of a tree, which has a property of height, and has a logarithmic growth based on the elapsed number of steps.

When the agent stores the constructor of the evolutionary object, it has access to all of the parameters in the constructor. Based on the parameters, the agent can *assume* some functions of the object.

There is one problem, which arises from storing the information about the extracted classes. As soon as the extraction is over, the connection between the database and the main program is closed. We could solve the problem by allowing the agents to gain access into the database, look up a certain evolutionary object, and by observing the input parameters, try to get a method that matches the return type of the parameter. However, this would ruin our model of separating the agents from the object space. We solve this problem by asking the agents to choose an evolutionary object, and ask the environment to connect to the object space and return the methods of that particular object to the agent.

When the environment returns an adequate function of the specific class – requested by the agent – the agent can then *update* its class wrapper of that particular object. In other words, the agent *gains experience and insight* of that object.

We now introduce the tags which is then used to observe the characteristics of an object. We define the set of tags for each object by their method names. There are several methods that are generated by the @Data annotation and are useful for the inner workings of the simulation – such as "toString" and "canEqual" methods – but otherwise are not needed for the agent itself. We expanded the ClassWrapper class with a set attribute that stores only the names of the methods (without modifiers), and filters the class names for only the object specific method names that we defined. We compare two arbitrary objects using this member of each class, and if the agent finds a similar method name in the two set of method names, then the agent is able to create a new type of object using the mixture of the two objects.

Result of our approach of simulating the intelligence evolution of artificially generated intelligence clearly indicate that very simple agents are able to gain knowledge and experience of their surroundings, and also have the ability to combine the objects they find similar.

4 Major results

Using the *Block* as a base, and *Tree* as an evolutionary object, we ran a simple simulation to see whether our model is actually capable of simulating some sort of *evolution of intelligence* of our agents.

As we outlined in the model description, we split each cycle of observation into discrete time intervals, with the duration of 1 second.

We ran the simulation for approximately half a minute, which was adequate for the agent to *learn, or gain experience* about all the functional methods of the Tree evolutionary object.

As expected, at the beginning, the agent did not know anything about the Tree object other than its constructor function and its parameters. By extracting the parameter types, and asking the environment to return a method of that object with the specified parameter type, the agent was able to extend its knowledge of that object.

5 Analyzing the results

Based on the model we have created and the results of its implementation, we are positive that our method is adequate for simulating knowledge, or as we called at the beginning of the paper *intelligence* evolution.

When we deal with *evolutionary objects*, it might occur that the agents take an uncertain amount of time exploring the properties of a particular the object – which inevitable leads of uncertain, possibly hours, or perhaps days long of simulation run-time. In order not to lose the data that the agents already possess, it would be wise to persist the agents as well in a different database. This can be done so with the same *entity managers*, by defining other *persistence units* in the persistence XML file (used by Java for handling entity object and communicating with the database). We will worry about this problem later.

One valid question the reader might ask, is how is the agent actually be able to know the exact properties of a particular object in the database. This can be solved several ways. The first solution is to ask the environment to actually store this information, and pass it on to the agent in another type of wrapper. We will use this approach. Another approach could be a *verifying* approach, where the agent asks if the fed parameter to the object was "good enough" for its needs. To implement this approach, the environmental objects need to consist methods that return either true or false values.

The code-base for this project can be downloaded from GitHub from the following link: `https://github.com/Woaf/Reflection`.

6 Suggestions for improvements

Since this project is very much in its experimental stage, there are plenty of areas where it can be improved.

The agents could be improved to not only combine objects that show some type of similarity, but also objects that could potentially be different in every aspects.

Another major improvement could be to enhance the agents with certain desires, or favours, which have an impact on what kind of methods they are looking for when observing an objects, and how they combine them to make a new object – preferably one that favors them in a certain way. Hence, a more clever decision making system should also be considered for the agents, as they might favor an attribute over another based on their inner states – a similar idea that of multiple attribute decision making. [13] [3]

The simulation was done by using only one agent, however it would be very interesting to see it run with several agents. A type of *system of consensus* could be implemented by the knowledge base of different agents, and how they agree or disagree on certain elements of their possibly different observations. This could be done by storing the knowledge of all agents in one list, and if the cardinality of certain methods reach a threshold, then it could be considered to be accepted by all agents within the simulation. This consensus is called *multi–attribute negotiation*, described in an article by C. M. Jonker et al.[7]

It would also be very interesting to see what would happen, if new agents were to be added to the simulation with no prior knowledge, and what impact would they make on the general knowledge base of the agents – where the multi–attributes used in the communication between the agents are incomplete.[9]

Our model describes a method where the environment plays the central role, however, there are different models where the agents are in the centre of focus. A model focusing only on agents, and their pure interactive abilities would result in a different type of communication between the agent and the environment.[10]

Bibliography

[1] Cristiano Castelfranchi. Simulating with cognitive agents: The importance of cognitive emergence. In *International Workshop on Multi-Agent Systems and Agent-Based Simulation*, pages 26–44. Springer, 1998.

[2] Lucie Ciencialová and Ludëk Cienciala. Variations on the theme: P colonies. In *Proceedings of the 1st International workshop WFM*, volume 6, pages 27–34, 2006.

[3] Michael T Cox. Perpetual self-aware cognitive agents. *AI magazine*, 28(1):32, 2007.

[4] Erzsébet Csuhaj-Varjú. P automata. In *International Workshop on Membrane Computing*, pages 19–35. Springer, 2004.

[5] Michael N Huhns and Munindar P Singh. Cognitive agents. *IEEE Internet computing*, 2(6):87–89, 1998.

[6] Stephen D. Roberts Jeffery A. Joines. Fundamentals of object-oriented simulation. In *Proceedings of the 1998 Winter Simulation Conference*, pages 141–149, 1998.

[7] Catholijn M Jonker and Jan Treur. An agent architecture for multi-attribute negotiation. In *International joint conference on artificial intelligence*, volume 17, pages 1195–1201. LAWRENCE ERLBAUM ASSOCIATES LTD, 2001.

[8] Mike Keith and Merrick Schincariol. *Pro EJB 3: Java Persistence API*. Apress, 2006.

[9] Guoming Lai, Cuihong Li, and Katia Sycara. Efficient multi-attribute negotiation with incomplete information. *Group Decision and Negotiation*, 15(5):511–528, 2006.

[10] Charles M Macal and Michael J North. Tutorial on agent-based modeling and simulation. In *Simulation conference, 2005 proceedings of the winter*, pages 14–pp. IEEE, 2005.

[11] Alexandre Muzy, Eric Innocenti, Antoine Aıello, Jean-François Santucci, Paul-Antoine Santoni, and David RC Hill. Modelling and simulation of ecological propagation processes: application to fire spread. *Environmental Modelling & Software*, 20(7):827–842, 2005.

[12] Isabel A Nepomuceno-Chamorro. A java simulator for membrane computing. *J. UCS*, 10(5):620–629, 2004.

[13] Richard L Oliver. Cognitive, affective, and attribute bases of the satisfaction response. *Journal of consumer research*, 20(3):418–430, 1993.

[14] Robert F Stärk, Joachim Schmid, and Egon Börger. Abstract state machines. In *Java and the Java Virtual Machine*, pages 15–28. Springer, 2001.

[15] Bernard P Zeigler. *Object-oriented simulation with hierarchical, modular models: intelligent agents and endomorphic systems*. Academic press, 2014.