Article

General Environment Description Language

Krzysztof Zatwarnicki [1,*], Waldemar Pokuta [1], Anna Bryniarska [1], Anna Zatwarnicka [1], Andrzej Metelski [2] and Ewelina Piotrowska [1]

1 Department of Computer Science, Opole University of Technology, Proszkowska 76, 45-758 Opole, Poland; w.pokuta@po.edu.pl (W.P.); a.bryniarska@po.edu.pl (A.B.); a.zatwarnicka@po.edu.pl (A.Z.); e.piotrowska@po.edu.pl (E.P.)
2 Department of Mathematics and IT Applications, Opole University of Technology, Proszkowska 76, 45-758 Opole, Poland; A.Metelski@po.edu.pl
* Correspondence: k.zatwarnicki@gmail.com

Abstract: Artificial intelligence has been developed since the beginning of IT systems. Today there are many AI techniques that are successfully applied. Most of the AI field is, however, concerned with the so-called “narrow AI” demonstrating intelligence only in specialized areas. There is a need to work on general AI solutions that would constitute a framework enabling the integration of already developed narrow solutions and contribute to solving general problems. In this work, we present a new language that potentially can become a base for building intelligent systems of general purpose in the future. This language is called the General Environment Description Language (GEDL). We present the motivation for our research based on the other works in the field. Furthermore, there is an overall description of the idea and basic definitions of elements of the language. We also present an example of the GEDL language usage in the JSON notation. The example shows how to store the knowledge and define the problem to be solved, and the solution to the problem itself. In the end, we present potential fields of application and future work. This article is an introduction to new research in the field of Artificial General Intelligence.

Keywords: intelligent systems; autonomous systems; collective intelligence; computers and INFORMATION processing; knowledge management; artificial intelligence; software agents; agent-based modeling; autonomous agents; intelligent agents; knowledge engineering; knowledge representation

check for updates

Citation: Zatwarnicki, K.; Pokuta, W.; Bryniarska, A.; Zatwarnicka, A.; Metelski, A.; Piotrowska, E. General Environment Description Language. *Appl. Sci.* **2021**, *11*, 740. https://doi.org/10.3390/app11020740

Received: 19 November 2020
Accepted: 10 January 2021
Published: 14 January 2021

Publisher’s Note: MDPI stays neutral with regard to jurisdictional claims in published maps and institutional affiliations.

Copyright: © 2021 by the authors. Licensee MDPI, Basel, Switzerland. This article is an open access article distributed under the terms and conditions of the Creative Commons Attribution (CC BY) license (https://creativecommons.org/licenses/by/4.0/).

1. Introduction

For a long time, engineers have been using different techniques to control systems without directly supervising them. Various types of mechanical control systems were perfected by their creators, obtaining the highest forms as windmills, steam machines, locomotives, cars and entire factories. As a result of the development of science, however, new technical possibilities have emerged, enabling the creation of decision-making systems, e.g., using biological, chemical, quantum and electronic systems. Advanced information processing capabilities mean that we are not currently dealing only with simple control systems, but very complex systems, having the ability to acquire information, collect it, transform it into knowledge and even process the knowledge. The indicated capabilities allow us to create more and more advanced systems, such as banking advisory systems, autonomous cars, robots cooking meals or vacuum cleaners cleaning our apartments automatically. Currently, developed and manufactured systems are designed to support very specific tasks with a relatively narrow spectrum of applications. Those systems often use deep learning techniques that achieve very good results, e.g., in image recognition, creation of artificial images or text.

The challenge now is to create general-purpose systems that would enable both learning about the environment in an autonomous manner and the performance of commissioned tasks in a diverse environment. The implementation of general-purpose systems

would not be possible without the development of systems that perform specific tasks, since they are the basis for the correct interpretation of the data flowing from the sensors, and they make it possible to perform the activities that make up the entire large task.

However, it can be stated that technological progress is already at such a high level that it will be possible to develop an intelligent general-purpose autonomous system and implement it into practical solutions. The use of such systems would, of course, be extensive, starting with home applications, in which the robot would perform simple operations such as cooking and cleaning, continuing with applications in industry and agriculture, and ending with applications in the space industry.

Work on general-purpose artificial intelligence has been going on for over 20 years. However, it should be noted that despite the fact that the new proposals and solutions are emerging in this area, unfortunately, no significant and noticeable breakthrough has been made that would enable widespread implementation of the proposed technologies. The vast majority of the AI field today is concerned with what might be called "narrow AI"—concentrated on creating programs that demonstrate intelligence in one specialized area [1]. There is still a need to work on general solutions that would constitute a framework enabling, on the one hand, the integration of already developed "narrow AI" solutions, and on the other hand, general problem-solving.

In the following article, a new proposal in the field of general-purpose artificial intelligence is discussed. We present a new language that may be used to describe almost any environment in which robots or agents can potentially work. In addition, the presented language makes it possible to give meaning to objects observed in the environment and to plan tasks using the objects.

1.1. Literature Review

At first, the definition of artificial intelligence was very much identified with the intelligence present in humans or animals. It was referred to as self-awareness, i.e., having thoughts, feelings, worries, understanding the situation. However, such a definition of artificial intelligence causes its immeasurability (e.g., some systems can imitate human emotions very well through their behavior). Definitions of artificial intelligence are evolving with the emergence of new systems. To assess whether a new system will be considered intelligent or not, we can use the Universal Evaluation of Behavioral Features [2]. Another approach, which assumes that the definition should be simple and lead to fruitful research, states that it is "adaptation with insufficient knowledge and resources" [3]. In other publications, it is important to distinguish intelligent systems according to the specific criteria—general and narrow, individual and collective, biological and artificial, new and old, dispersed and centralized [4]. In yet another approach, artificial intelligence should work with human intelligence in order to complement it [5].

Many solutions concerned with artificial intelligence focus on solving one type of task (e.g., image recognition, creating music, driving a vehicle, etc.). In addition to such partially intelligent systems, there are attempts to develop artificial intelligence of general-purpose. The problem in this approach is the integration of many more straightforward solutions using one Artificial General Intelligence (AGI) tool [1]. Sometimes, however, in order to solve a complex problem, there is a need to integrate many smaller systems. One can then talk about General Collective Intelligence [6–9]. AGI should be able to anticipate certain behaviors or actions that should be taken to achieve the expected result [10]. Another essential feature of AGI is adapting to changing conditions, i.e., using new environmental features while not forgetting what has been developed in previous situations (cumulative learning) [11]. Understanding natural language (drawing conclusions, learning) is also one of the directions of AGI research [12].

Creating an intelligent system to solve any problem can be difficult, which is why systems are designed to address a particular group of tasks, e.g., playing games [13]. A special framework for testing agents (game players) has been created. New rules of the game can be described using the Game Description Language (GDL) in this environment.

In this way, tournaments are played between agents to determine the algorithm that best solves the problems with changing rules [14].

Another area in which AGI is used is automatic planning. The issues of automated planning have widened from toy problems to real applications. Automatic planning is difficult—we need to know the structure of the problem. Automatic planners do not provide good results in many areas. An overview of research on automated planning can be found in the literature [15,16]. The issue of planning in business is a specific subgroup of algorithms for planning. The objective is to find a procedural way of working for a system that is declaratively described, while optimizing performance measures [17,18]. This can be achieved using Formal State Transition Systems [19]. Automatic planning is used in robotics, production, logistics, transport and spaceflight. In an intelligent plan, attention needs to be paid to the analysis phase because the systems can identify and redefine variables, so the accuracy of the model (generated by automatic planners) can be increased [20].

The use of the Non-Axiomatic Reasoning System (NARS) is an interesting proposition in the field of AGI. It assumes that the system should be finished and open, work in real-time and adapt with insufficient knowledge or resources. A language that has semantics based on experience has been introduced. According to this, the value of truth of the judgment is determined on the basis of the previous experience, and the meaning of a term depends on its relationship with other terms [21–23].

Sometimes, systems have limitations related to the way of recording knowledge about the surrounding environment. They are too limited or too complicated. The recording method, which is not general enough, causes the system to be dedicated to one task only. Therefore, an important part of these systems is the way to record knowledge about the environment in which the job must be solved. The system environment can be a world discovered by a robot using sensors. It can also be a policy environment in strategic games or a production facility operation. Many publications have introduced languages that could describe specific contexts in which AGI applications will run.

The game description language is a language to describe the rules of the game [24]. The language consists of concepts such as term, atomic sentence, literal, datalog rule, dependency graph, model and satisfaction. By defining these statements, it is possible to set the rules of the game. An agent (player) operating in this environment must, by following these principles, demonstrate better adaptation to new conditions than the other agents.

Many agent description languages have been created in recent years, e.g., Q [25], JADL [26], ADL [27], JADEL [28]. Q describes the interaction scenarios between agents and users based on external policies. External rules describe the environment in which agents move. These rules can be changed, e.g., in the event of an emergency and evacuation. The "JADL" language (JIAC Agent Description Language) is a language of description in which the Agent environment is defined by means of a goal to be achieved and rules [26]. Rules are implemented quite simply, consisting of a condition and two actions, one of which is executed when the condition becomes true and the other when the condition becomes false. In Architectural Description Language (ADL), the behavioral model consists of eight main project units: agent, knowledge base purpose, ability, beliefs, plan, events, activities and services [27]. An agent needs knowledge of his environment to make good decisions. The knowledge is stored in the agent in the form of one or many knowledge bases making its information state. The knowledge base consists of a set of beliefs that the agent has about his environment. A belief represents the view of the agent's current environment. JADEL (JADE Language) was created as a way to reduce the complexity of building systems based on Java Agent Development framework (JADE), providing support during the implementation of agents, behaviors and ontologies. JADEL is to enable agents to be used as components [28].

Among the languages that use non-axiomatic knowledge, we can distinguish NARS-ESE [29], NARS [30], ALAS [31,32]. NARSESE and NARS, that are languages used to build

a system with learning ability. The system is able to acquire problem-solving skills based on experience, it also is adaptive, and able to distinguish between the external environment and internal knowledge. The following elements are defined in these languages: a judgment—an expression with the value true at the input, which is a representative of a piece of knowledge that the system learns or checks; a question—which the system answers in accordance with the beliefs of the system; a goal—an expression to be implemented by performing certain operations in accordance with the beliefs of the system. The ALAS language uses non-axiomatic logic for distributed inference for agents.

Planning is a branch of artificial intelligence (AI) that attempts to automate reasoning about plans and, above all, reasoning that serves to formulate a plan needed to achieve a specific goal in a given situation. Planning for artificial intelligence is model-based: the planning system takes the description of the initial situation as input, the activities available for its change and the condition of the goal to create a plan consisting of those activities that will achieve the goal after execution from the initial situation. Planning languages include Planning Domain Definition Language (PDDL) [33,34] and Stands for Stanford Research Institute Problem Solver (STRIPS) [35,36] for multi-agent environments. PDDL aims to express the "physics" of the domain, that is, what the predicates are, what actions are possible, what the structure of complex operations is and what the effects of actions are. Most planners also require some kind of "advice" or annotation including information which activities should be used in order to achieve one of the goal, or in what complex actions, under which circumstances. The PDDL language does not provide such advice, making it a neutral tool that can be used in various places. As a result of this neutrality, almost all planners need to expand notation, but this can be done in different ways. The STRIPS language works in a multi-agent environment; each agent tries to achieve its own goals, usually leading to a conflict of objectives. However, there is a group of problems with conflicting goals that can be met at the same time. Such problems can be modeled as a STRIPS system. If the STRIPS planning problem is reversible, planning under uncertainty methodology can be used in order to solve the inverted problem, and then find a plan that solves the problem with multiple agents.

Language grounding is another important issue in the context of language creation for intelligent systems. Grounding means connecting linguistic symbols to perceptual experiences and actions [37]. These issues can be particularly useful e.g., in the communication of people with robots [38]. Robots can learn to correlate natural language with the physical world being sensed and manipulated [39,40]. There are publications concerned with teaching robots to recognize objects by their names and attributes, and demonstrating their learning action [41,42].

1.2. Motivation and Goals

The main objective of the following work is to present a new General Environment Description Language (GEDL) that may describe almost any environment in which robots or programs can potentially operate independently.

The proposed language differs from the languages and solutions described in this subsection A. GEDL can be treated more like a frame that determines the way of perceiving reality and organizing the data and knowledge. We believe that the presented language should use AI techniques developed so far and also presented in subsection A. For example, PDDL or STRIPS languages could potentially be used by individuals using GEDL to solve problems. Likewise, other artificial intelligence techniques and tools, such as deep learning libraries TensorFlow [43], Keras [44], PyTorch [45] or fuzzy logic [46] can be a part of GEDL based autonomous system devoted to building intelligent general-purpose individuals.

It is assumed that individuals using our language will have unlimited resources, including memory and computing power. This assumption is the opposite of that commonly used in the agent systems discussed above. We believe that individuals (robots, programs) using the language will have access to cloud computing or will use future technologies.

The use of concepts that give meaning to the observed elements is a novelty, significantly differentiating the proposed language from similar languages previously described in the literature. The manner of describing the actions that individuals can take is also of great importance. We do not define the manner of their implementation, but only the state of the system before and after execution.

1.3. Paper Organization

The paper is structured in the following way: in Section 2, the overall concept of the proposed GEDL language is introduced. Sections 3–7 contain notation, definitions and examples of the elements of this language. In Section 8, an exemplary individual knowledge with problem solution is presented. The conclusions are presented in Section 10.

2. Introduction to GEDL Language

In this chapter, the general assumptions of the GEDL language are described. It is presented how the elements are included in the GEDL language and how they are further interpreted. It is necessary to be able to define the individual elements of this language later.

Humans are intelligent beings who are able to build conceptual systems in their minds, learn from mistakes, save knowledge, and transfer it to other individuals. Trying to develop a general language for describing the environment that could be used in autonomous general-purpose robots, an attempt can be made to model ourselves by observing our understanding of the environment and behavior. In some ways, we are also doomed in our considerations to recreate our understanding. It is because, in most cases, we are not able to understand other ways sufficiently, e.g., represented by animals.

The presented General Environment Description Language helps to describe the environment, and thus also save knowledge about the environment for the needs of autonomous systems that make decisions and perform tasks in the environment. This language makes it possible to systematize uncertain, vague, and sometimes not fully defined knowledge.

The environment can be a fragment of the physical world in which we live or another reality in which information is processed, for example, a computer game.

There are individuals in the environment, i.e., individuals who possess and collect individual knowledge. Individuals acquire knowledge using a cognitive mechanism. The mechanism may not provide full knowledge, or this knowledge may not be accurate. Therefore, it should be noted that in the described approach, the environment with its objects is distinguished from the knowledge of the environment possessed by the individual. Each individual may have utterly different knowledge of the environment. In extreme cases, when individual knowledge will be empty, the individual will find that the environment does not exist. The way the individual perceives the environment is presented in Figure 1.

An individual in the environment can distinguish an object that is an element of the environment. The individual decides on distinguishing or perceiving the element. To distinguish it, the individual must need it, and the cognitive mechanism must be able to do it. An element called an instance in individual knowledge is the equivalent of an object in the environment. The instance is the knowledge about the objects and it contains features subjectively distinguished by an individual. Features can have values, e.g., color can be green, tastiness can be unpalatable. Instances may also be in relationships with other instances and they may perform actions. It should be noted that an individual is also an instance having its own features, relationships and actions.

An instance can change over time by changing the configuration of its features, relationships and actions. The values of features and relationships of an instance, at a particular moment, are called a state.

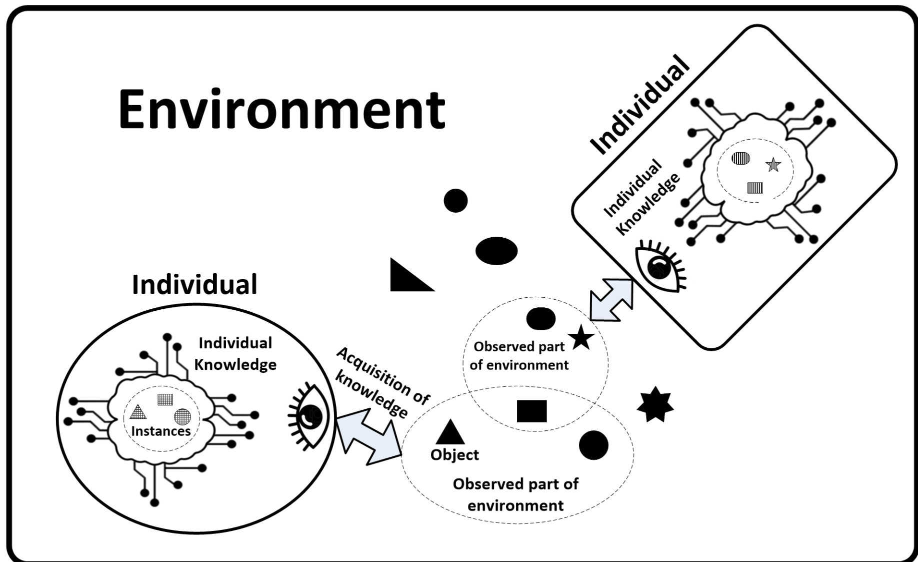

Figure 1. The environment with objects and perception of individuals.

Two objects with the same characteristics should be distinguished in the individual knowledge as two independent instances, in which attributes will be assigned independently. Such an approach can, however, be problematic when making decisions autonomously because a large number of ungrouped instances would be in the individual knowledge then. In connection with the above, we are introducing a new approach in the field of environment description languages, called an instance concept. The instance concept is a set of all instances with features that have assigned values from a specific range. For example, let us take the knife concept, i.e., sets of all instances in which we can distinguish a blade having a length of 1 to 25 cm. At the same time, some instances belonging to the knife concept may belong to the cutlery concept, and others may belong to the concept of a hunting knife.

The idea of the instance concept can also help to formulate problems to be solved by autonomous systems and to improve the communication of these systems with people. Let's consider the following example: the robot is to make pancakes from edible ingredients including flour and eggs. Edible ingredients, flour and eggs, are concepts of instances. To perform the task, the robot has wheat flour and lime flour (used in construction) available. Both types of flour belong to the concept of flour, which has the following features: they combine well with water, form a sticky mass when mixed with water, are white, form dust, etc. However, in the concept of edible ingredients, it is indicated that the instance should have an eatable feature. In this way, the robot will easily be able to classify instances and choose wheat flour, having the eatable feature as opposed to lime flour. A properly developed decision-making mechanism could also ensure that it would not be necessary to indicate that the ingredients of the pancake are edible because the pancake itself is edible. The mechanism can "guess" that an inedible ingredient cannot be used there.

As it was mentioned earlier, we can also distinguish the relationships between instances, e.g., instance 1 is above instance 2 or instance 1 is a mother of instance 2. In order to create a relationship between instances, the individual should notice the relationship between objects in the environment, or it must result from logical premises in the individual knowledge. We also distinguish the concept of a relationship that gives the meaning of a relationship.

Instances can perform actions that can change a fragment of the environment and, more precisely, change the instance states, e.g., by changing existing objects. Actions are performed by instances, and other instances may also be used in their performance. For example, a robot can do laundry at home using a washing machine. Actions are attributed only to those instances that are able to carry out a specific action autonomously (they make decisions on their own). For example, a kitchen knife is used for chopping, but it cannot

perform the chopping on its own. This can be done, however, by a human or possibly by an autonomous robot, and a knife is only a tool when performing actions. The performance of actions by an individual does not always mean that the goal of carrying out actions will be achieved in the environment, e.g., making pancakes may fail for several reasons, often even independently of the quality of the decisions made.

For a set of similar actions, regardless of the instances that implement them, concepts of action can be distinguished. The action concept is a set of all actions that transform instances in such a way that they finally obtain instances being in a similar state. Thanks to the action concept, it will be possible to find all instances being able to perform similar actions. For example, a robot that will be given the task of doing laundry will search, at home or in a wider available environment, for all instances that can wash clothes.

Actions are atomic, and smaller indirect activities are not distinguished for them, although they may consist of such. Actions can be implemented using advanced subsystems, such as artificial neural networks. In this case, we act similarly to biological systems, when, for example, we have to squeeze an object by hand, we do not think which muscles should be used, we just do it.

Actions are used in problem solutions. The solution may consist of a group of actions carried out in a specific order, in parallel or concurrently. An individual can solve the problem by developing several solutions, choosing the best, and implementing it. Developing solutions is time-consuming, which is why they should be stored in the individual knowledge and used when a new similar problem appears.

As it was mentioned earlier, each individual perceives and learns the environment on their own, perhaps without even understanding that there are other individuals in that environment. However, since there may be many individuals in the environment with similar cognitive capabilities, e.g., several robots of the same type, we should consider sharing the individual knowledge or fragments of this knowledge. This would significantly accelerate knowledge growth and enable us to share previously-found solutions. In the humans' world, such knowledge sharing is called education.

Precise definitions of the ideas and concepts outlined above will be presented in the further sections.

3. Notation Used in GEDL Language

The GEDL language allows an individual to describe the environment in which the individual acts. The JSON notation [47] was chosen for storing knowledge according to the GEDL. The JSON has many advantages, including the universality of applications, numerous implementations, ease of storing data in databases. It is easy for humans to read and write, and also easy for machines to parse, generate and interpret. The JSON allows us to use universal data structures and most of the modern programming languages support it. Thanks to this, it would be potentially possible to exchange or even share knowledge by individuals constructed using various technologies. However, it should be mentioned that other notations, such as XML, could also be used to store data according to the GEDL.

The definitions presented in further chapters indicate the meanings of the terms used. They also present the way of storing individual knowledge using the JSON notation. It is worth mentioning that JSON objects are surrounded by curly braces "{}" and are written in key/value pairs. Arrays are surrounded by square braces "[]". Keys must be strings (text) and values must be valid JSON data types: string, number, another JSON object, array, boolean or null defined according to [47]. Besides, in the GEDL language, some of the JSON strings contain a source code written in a programming language. This approach is sometimes used in JSON [48,49], and in our case, it makes the language more flexible in the way of expression.

Further, the JSON objects are defined according to the following example:

An element *S* is a JSON object constructed in the following way:

```
1 "SName": {
2 "name1": JO1,
3 ...
4 "namen": JOn,
5 ...
6 "nameN": JON
7 }
```

In the example, $S, JO_1, JO_2, ..., JO_N$ denote certain structures according to the JSON notation, while the elements $''SName'', ''name_1'', ..., ''name_N''$ indicate the place in the structure of the object and the order of individual elements. They are also human-understandable.

As a part of the description of elements of the GEDL language, the mathematical notation is introduced to clarify the presented definitions. According to the notation, S is a Cartesian product $S = JO_1 \times ... \times JOn \times ... \times JO_N$, where $N \in \mathbb{N}$, and $JO_1, JO_2, ..., JO_N$ are sets constructed in accordance with its definitions, and $JO_n = \{JO^o_{n1}, ..., JO^o_{nm}, ..., JO^o_{nM}\}$, where $n = 1, \ldots, N, N \in \mathbb{N}, m = 1, \ldots, M, M \in \mathbb{N}$ and JO^o_{nm} is an element (a JSON object or another JSON structure). A subset of occurrences of JSON objects (relation) is marked in the following way $S^W \subseteq S, S^W = \{S^o_1, ..., S^o_K\}$, where K is the number of existing JSON objects. The object S^o is defined as a series $S^o = (JO^o_1, ..., JO^o_n, ..., JO^o_N), S^o \in S^W, JO^o_n \in JO_n, n = 1, \ldots, N$. The presented approach is similar to the approach used in the database theory [50].

The upper index ^{o} indicates the occurrence of an element built according to the presented definitions of JSON objects or another element constructed according to the JSON notation (e.g., S^o, S^o_1, JO^o_n). The lower index is added in the case of references to specific elements of sets or series (e.g., JO^o_n is the n-th element of S^o). The lack of a lower index indicates any element of the set or a series (e.g., S^o is an example of any element of the set S^W).

We denote the JSON array $[J^o_1, ..., J^o_p, ..., J^o_P]$ in the mathematical notation as series $aS^o = (J^o_1, ..., J^o_p, ..., J^o_P)$, where $P \in \mathbb{N}$, $J^o_p \in J$, J is the JSON object or another JSON structure.

We use the following definitions:

- Let $S^W \subseteq JO_1 \times ... \times JO_n \times ... \times JO_N$, where $n = 1, ..., N, N \in \mathbb{N}$ be any relation. A projection of the S^W on the set JO_n is denoted as $\pi_{JOn}(S^W)$ and defined in the following way:

$$\pi_{JOn}(S^W) = \{JO^o_n : \exists JO^o_1 ... \exists JO^o_{n-1} \exists JO^o_{n+1} ... \exists JO^o_N (JO^o_1, ..., JO^o_n, ..., JO^o_N) \in S^W\}. \quad (1)$$

- Let $S^W \subseteq JO_1 \times ... \times JO_n \times ... \times JO_N$, be a relation and $S^o = (JO^o_1, ..., JO^o_n, ..., JO^o_N), S^o \in S^W$, where $n = 1, ..., N, N \in \mathbb{N}$. The function *elem* is defined as follows:

$$elem(S^o, JO_n) = JO^o_n, \quad (2)$$

  where $JO_n \in \{JO_1, ..., JO_N\}$.
  For example:
  $S^W \subseteq JO_1 \times JO_2 \times JO_3, S^o \in S^W, S^o = (1, 2, 3), elem(S^o, JO_2) = 2.$
- Let S^o be a series $S^o = (JO^o_1, ..., JO^o_n, ..., JO^o_N)$, where $n = 1, \ldots, N, N \in \mathbb{N}$. The function *pos* is defined as follows:

$$pos(S^o, n) = JO^o_n, \quad (3)$$

  where $JO^o_n \in \{JO^o_1, ..., JO^o_n, ..., JO^o_N\}$.
  For example: $S^o = (3, 8, 5)$, $pos(S^o, 2) = 8$.
- Let S^o be a series $S^o = (JO^o_1, ..., JO^o_n, ..., JO^o_N)$, where $n = 1, ..., N, N \in \mathbb{N}$. The set of values of the series S^o is defined as follows:

$$\widehat{S^o} = \{JO^o_1, ..., JO^o_n, ..., JO^o_N\}. \quad (4)$$

Due to a large number of definitions describing the language, they are divided into four sections presenting:

- environment and the individual building the knowledge of the environment,
- a conceptual system storing concepts of elements which can potentially occur in the environment,
- occurrences of observed elements of the environment,
- experience of the individual containing problems to solve and the solutions.

The elements describing the GEDL language are complex, and there are many connections and dependencies between them, making it impossible to arrange the definitions from the simplest to the most complex one. The order of definitions is arranged in this way to facilitate reading and understanding of the text.

4. Environment and Individual

This chapter defines basic elements used in the GEDL language: the environment, the individual and the individual knowledge.

Definition 1. *An* ***environment*** *is a set* $E = \{O_1, ..., O_m, ..., O_M\}$ *containing objects* $O_m, m = 1, ..., M, M \in \mathbb{N}$*. The environment can be a fragment of the physical world, information space (e.g., computer game, computer network, stock exchange) or mixed physical and information space.*

Definition 2. *An* ***individual*** *IND is an entity building knowledge about the environment or having such knowledge about a fragment of the environment that is in its sphere of interest, and/or learning about it, and/or performing tasks in the environment. The individual can be an object* O_m *in the environment or can only observe it.*

Definition 3. *Any* ***individual knowledge*** *IK is a systematized knowledge about the environment possessed by an individual. Individual knowledge can be built using various techniques:*

- *an observation with the usage of sensors,*
- *an adaptation that is the improvement of knowledge about the environment which is the effect of subsequent observations or activities that bring new information,*
- *obtaining information stored outside the individual, e.g., previously collected by other individuals,*
- *a deduction or inference (knowledge, in this case, can be uncertain),*
- *other.*

The IK contains the following elements:

```
1 {
2 "conceptualSystem": oCS,
3 "occurrences": oOC,
4 "experiences": aE
5 }
```

- ***oCS***—*is a JSON object containing a conceptual system defined in Definition 4,*
- ***oOC***—*is a JSON object containing occurrences of instances and relationships, defined in Definition 12,*
- ***aE***—*is a JSON array containing a finite set of solved problems oE, defined in Definition 22,* $aE^o = (oE^o_1, ..., oE^o_j)$*,*

  $IK = oCS \times oOC \times aE.$

The individual knowledge $IK^o \in IK$ can change over time. We distinguish moments of time $t_1, ..., t_n, ..., t_N$, $N \in \mathbb{N}$, when the knowledge is updated, and in this way, we get a series $(IK^o_{t1}, ..., IK^o_{tn}, ..., IK^o_{tN})$ of subsequent versions of knowledge. In the definitions presented further in the article, we consider only one version of the knowledge IK^o_{tn} in one moment t_n.

5. Conceptual System and Its Elements

The conceptual system of the GEDL language is presented. Later, all elements of this system are described. Firstly, features and feature sets are introduced. Then, there are definitions of the instance, relationship, state and action concepts. Additionally, the instance concept variable definition is presented.

Definition 4. *A conceptual system oCS is a JSON object containing concepts of entities, concepts of relationships, concepts of activities. The conceptual system is:*

```
1 "conceptual system": {
2 "features": aF,
3 "featureSets": aFS,
4 "instanceConcepts": aIC,
5 "relationshipConcepts": aRC,
6 "actionConcepts": aAC
7 }
```

- ***aF***—*is a JSON array containing a finite set of elements of the following type: feature oF,* $aF^o = (oF^o_1, ..., oF^o_K)$,
- ***aFS***—*is a JSON array containing a finite set of feature sets oFS,* $aFS^o = (oFS^o_1, ..., oFS^o_L)$,
- ***aIC***—*is a JSON array containing a finite set of instance concepts oIC,* $aIC^o = (oIC^o_1, ..., oIC^o_M)$,
- ***aRC***—*is a JSON array containing a finite set of relationship concepts oRC,* $aRC^o = (oRC^o_1, ..., oRC^o_N)$,
- ***aAC***—*is a JSON array containing a finite set of action concepts oAC,* $aAC^o = (oAC^o_1, ..., oAC^o_O)$,

$oCS = aF \times aFS \times aIC \times aRC \times aAC$, *where, oF is defined in Definition 5, oFS is defined in Definition 6, oIC is defined in Definition 7, oRC is defined in Definition 8, and oAC is defined in Definition 11.*

5.1. Features

Definition 5. *A feature oF is a JSON object representing a distinguished property of an object in the environment. The feature is:*

```
1 {
2 "name": oFName,
3 "description": oFDescription,
4 "domain": oFDomain,
5 "default": oFDefault,
6 "unit": oFUnit,
7 "normalizationFunction": oFNormalizationFunction,
8 "initFunction": oFInitFunction
9 }
```

where:

- ***oFName***—*a JSON string containing the name of a feature uniquely identifying the feature,*
- ***oFDescription***—*a JSON string containing the description defining the meaning of the feature (optional),*
- ***oFDomain***—*a JSON object describing a set of values that the feature can take. The domain contains:*

  ```
  1 {
  2 "set": oFDSet,
  3 "min": oFDMin,
  4 "max": oFDMax
  5 },
  ```

  where:

- ***oFDSet***—*a set of values:*
  * *for measurable values (quantitative) it can be a JSON string containing the name of an earlier defined set, for example: real, integer, float numbers etc.,*
  * *for non-measurable values it can be a JSON array containing a set of acceptable values of the feature, where aFDSet is a finite set of values used as a dictionary, for example for a blood type* $aFDSet^o = ('0', 'A', 'B', 'AB')$.
- ***oFDMin***—*a JSON string or number containing a minimal value of this feature (optional),*
- ***oFDMax***—*a JSON string or number containing a maximal value of this feature (optional),*

$oFDomain = oFDSet \times oFDMin \times oFDMax$.
Let $oFDomain^o = (oFDSet^o, oFDMin^o, oFDMax^o)$ *and* $oFDomain^o \in oFDomain$. *A set of values* $oFD_{oFDomain^o}$ *is defined by* $oFD_{oFDomain^o} = \{x : x \in oFDSet^o \wedge x \geq oFDMin^o \wedge x \leq oFDMax^o\}$. *If* $oFDMin^o$ *and* $oFDMax^o$ *are not fixed (defined), then* $oFD_{oFDomain^o} = \{x : x \in oFDSet^o\}$.

- ***oFDefault***—*a JSON string or number containing a default value of the feature if it is not specified (optional),*
- ***oFUnit***—*a JSON string containing a unit of the value (optional),*
- ***oFNormalizationFunction***—*a JSON string containing a source code written in a programming language, containing a normalization function that returns a normalized value of the feature; it can be useful while choosing the optimal solution (optional),*
- ***oFInitFunction***—*a JSON string containing a source code written in a programming language, containing a function that can calculate the value of a feature, based on the value of signals from an individual sensor or based on the value of other features (optional).*

$oF = oFName \times oFDescription \times oFDomain \times oFDefault \times oFUnit \times oFNormalizationFunction \times oFInitFunction$,
$oFo \in oF, oF^o \in \widehat{aF^o}, aF^o = elem(oCS^o, aF), oCS^o = elem(IK^o, oCS), IK^o \in IK$.

Examples of two features—tastiness and bloodGroup:

```
1  {
2  "name": "tastiness"
3  "description": "the sensation of flavour perceived in the
      mouth
4  and throat on contact with a substance",
5  "domain":{
6  "set": "Real",
7  "min": 0,
8  "max": 1,
9  }
10 "default": 0
11 },
12 {
13 "name": "bloodGroup"
14 "description": "any of various classes into which human blood
15 can be divided according to immunological compatibility, based
16 on the presence or absence of specific antigens on red blood
17 cells.",
18 "domain":{
19 "set": ["0", "A", "B","AB"]
20 },
21 "default": "0"
22 }
```

5.2. Feature Sets

Definition 6. *A feature set oFS is a JSON object grouping a collection of features under one name. The feature set groups features. It can be written:*

```
{
"name": oFSName,
"features": oFNames
}
```

where:

- ***oFSName***—*a JSON string containing the name of the feature set uniquely identifying the feature set,*
- ***oFNames***—*is a JSON array containing a finite ordered set of feature names oFName,* $aFNames^o = (oFName_1^o, ..., oFName_m^o, ..., oFName_M^o)$, *the feature having the name oFName has to be defined in the conceptual system,* $\forall oFName^o \in \widehat{aFNames^o}\ \exists oF^o : aFNames^o = elem(oFS^o, aFNames) \wedge oFS^o \in \widehat{aFS^o} \wedge aFS^o = elem(oCS^o, aFS) \wedge oCS^o = elem\ (IK^o, oCS) \wedge oF^o \in \widehat{aF^o} \wedge aF^o = elem(oCS^o, aF) \wedge oFName^o = elem\ (oF^o, oFName) \wedge IK^o \in IK$,
  $oFS = oFSName \times aFNames$.

  For example:

```
{
"name": colour,
"features": [
"red",
"green",
"blue"
]
}
```

5.3. Instance Concepts

Definition 7. *An instance concept oIC is a JSON object defining a set of all instances that are similar in some aspects. The instance concept is defined as follows:*

```
{
"name": oICName,
"features": oICFeatures,
"relationshipConcepts": oICRelationshipConcepts,
"actionConcepts": oICActionConcepts,
"instanceConcepts": oICInstanceConcepts,
}
```

where:

- *textbfoICName—a JSON string containing the name of an instance concept uniquely identifying the concept,*
- ***oICFeatures***—*a JSON array containing a set of elements called a* $oICFeature$ *(*$oICFeatures$ *is optional).*
  The $oICFeature$ *is a JSON object defining a narrowed set of values for the existing feature* oF*. The object is constructed as follows:*

```
1 {
2 "name": oFName,
3 "range": oICFRange,
4 }
```

where:

- **oFName**— is a JSON string containing a name of an existing feature, $\exists oF^o : oFName^o = elem(oF^o, oFName) \wedge oF^o \in \widehat{aF^o} \wedge aF^o = elem(oCS^o, aF) \wedge oCS^o = elem(IK^o, oCS) \wedge IK^o \in IK$,
- **oICFRange**—is a JSON object defining a range of values from the domain of the feature to which a value of an instance feature has to belong to be in the set of the instance concept,

```
1 {
2 "set": oICFRSet,
3 "min": oICFRMin,
4 "max": oICFRMax,
5 }
```

  - **oICFRSet**—is a JSON string or an array defining a set of values from the domain of the feature (optional),
  - **oICFRMin**—is a JSON string or decimal containing a minimal value of the range (optional),
  - **oICFRMax**—is a JSON string or decimal containing a maximal value of the range (optional),

$oICFRange = oICFRSet \times oICFRMin \times oICFRMax$
Let $oICFRange^o = (oICFRSet^o, oICFRMin^o, oICFRMax^o)$, *moreover* $oICFRange^o \in oICFRange$.
A set of values $oICFR_{oICFRange^o}$ *is defined as* $oICFR_{oICFRange^o} = \{x : x \in oICFRSet^o \wedge x \geq oICFRMin^o \wedge x \leq oICFRMax^o\}$.*If* $oICFRMin^o$ *and* $oICFRMax^o$ *are not fixed, then* $oICFR_{oICFRange^o} = \{x : x \in oICFRSet^o\}$.

- **oICRelationshipConcepts**—is a JSON array containing a set of elements oICRC, $oICRelationshipConcepts^o = (oICRC^o_1, ..., oICRC^o_p)$.
The oICRC is a JSON object indicating a relationship concept. The instance belonging to the instance concept should be in a relationship belonging to the relationship concept. The object is constructed as follows:

```
1 {
2 "relationshipConcept": oICRName,
3 "role": oICRRole
4 }
```

where:

- **oICRName**—is a JSON string containing a name of an existing relationship concept (defined in Definition 12) to which a relationship should belong, in which there is an instance belonging to the instance concept. The following roles should be met: $\forall oICRC^o \in \widehat{oICRelationshipConcepts}\ \exists oRC^o : oICRelationshipConcepts^o = elem(oIC, oICRelationshipConcepts) \wedge oRC^o \in \widehat{aRC^o} \wedge aRC^o = elem(oCS^o, aRC) \wedge elem(oICRC^o, oICRName) = elem(oRC^o, oRCName) \wedge oCS^o = elem(IK^o, oCS) \wedge IK^o \in IK$,
- **oICRRole**—is a JSON string containing a role that should have an instance in the relationship belonging to the relationship concept. Possible values: role1, role2.

oICRelationshipConcepts is optional.

- ***oICActionConcepts***—*is a JSON array containing a finite set of oACNames identifying action concepts (defined in Definition 11).* $oICActionConcepts^o = (oACName^o_1, ..., oACName^o_R)$. *An instance belonging to a given instance concept must contain actions belonging to all action concepts with identifiers contained in oICActionConcepts.*
  $\forall oACName^o \in \widehat{oICActionConcepts^o}\ \exists oAC^o : oAC^o \in \widehat{aAC^o} \wedge aAC^o = elem(oCS^o, aAC) \wedge oACName^o = elem(oAC^o, oACName) \wedge oCS^o = elem(IK^o, oCS) \wedge IK^o \in IK$.
  oICActionConcepts is optional.
- ***oICInstanceConcepts***—*is a JSON array containing a finite a set of oICName identifying other instance concepts to which an instance has to belong to be in a given instance concept.* $oICInstanceConcepts^o = (oICName^o_1, ..., oICName^o_S), \forall oICName^o \in \widehat{oICInstanceConcepts^o}\ \exists oIC^o : oIC^o \in \widehat{aIC^o} \wedge aIC^o = elem(oCS^o, aIC) \wedge oICName^o = elem(oIC^o, oICName) \wedge oCS^o = elem(IK^o, oCS) \wedge IK^o \in IK$. *The set of oICNames indicates the instance concept that narrows down a set of instances belonging to the given instance concept.*

$oIC = oICName \times oICFeatures \times oICFeatures \times oICRelationshipConcepts \times oICActionConcepts \times oICInstanceConcepts, oCS^o = elem(IK^o, oCS), IK^o \in IK$.

The instance concepts group instances that are similar in some way or belong to the same kind. It can be said that the instance is in the instance concept when the instance is in the set ICM defined in Definition 18. In order to distinguish concept names from instance names, all concept names start with a prefix c. Definition 18 accurately determined the belonging of instance to the instance concept.

For example:

```
1  {
2  "name": "c_apple",
3  "features":[
4  {
5  "name": "tastiness",
6  "range":{
7  "min": 0.3,
8  "max": 1
9  }
10 },
11 {
12 "name": "red",
13 "range":{
14 "min": 100,
15 "max": 255
16 }
17 },
18 {
19 "name": "green",
20 "range":{
21 "min": 100,
22 "max": 255
23 }
24 },
25 {
26 "name": "blue",
27 "range":{
28 "min": 100,
29 "max": 255
30 }
31 }
```

```
32 ]
33 }
```

The concept can also be narrowed down to another concept, for example:

```
1 {
2 "name": "c_tastyApple",
3 "oICInstanceConcepts": ["c_apple"],
4 "features":[
5 {
6 "name": "tastiness",
7 "range":{
8 "min": 0.3,
9 "max": 1
10 }
11 }]
12 }
```

5.4. Relationship Concepts

Definition 8. *A relationship concept oRC is a JSON object defining a set of all relationships that have the same meaning. The relationship concept represents relationships between instances belonging to two instance concepts. The concept gives meaning to the relationship. It can be defined as follows:*

```
1 {
2 "name": oRCName,
3 "description": oRCDescription,
4 "role1": oRCRole1,
5 "role2": oRCRole2,
6 "conditionsEstablishingRelation":
    oRCConditionsEstablishingRelation,
7 "conditionsRemovingRelation": oRCconditionsRemovingRelation
8 }
```

The concept of a relationship includes:

- ***oRCName**—is a JSON string containing a name uniquely identifying the relationship concept,*
- ***oRCDescription**—is a JSON string containing a description of the meaning of the relationship (optimal),*
- ***oRCRole1**—is a oFName identifying the name of the first instance concept to which instances being a part of a relationship belong,*
  $\exists oIC^o : aIC^o = elem(oCS^o, aIC) \wedge oIC \in \widehat{aIC^o} \wedge oRCRole1^o = elem(oIC^o, oICName) \wedge oCS^o = elem(IK^o, oCS) \wedge IK^o \in IK,$
- ***oRCRole2**—is a oFName identifying the name of the second instance concept to which instances being a part of a relationship belong,*
  $\exists oIC^o : aIC^o = elem(oCS^o, aIC) \wedge oIC \in \widehat{aIC^o} \wedge oRCRole2^o = elem(oIC^o, oICName) \wedge oCS^o = elem(IK^o, oCS) \wedge IK^o \in IK,$
- ***oRCconditionsEstablishingRelation**—is a JSON string containing a set of conditions that have to be fulfilled in order to build a relationship according to the relationship concept. Conditions should be written in a source code in a programming language (optional),*
- ***oRCconditionsRemovingRelation**—is a JSON string containing a set of conditions that have to be fulfilled to remove a relationship built according to the relationship concept. Conditions should be written in a source code in a programming language (optional).*

The relationship concept describes a one-direction relation (but it is not the relationship itself) between the first element ($oRCRole1^o$) and the second one ($oRCRole2^o$). The order of those two parameters of the relationship is important.

$oRC = oRCName \times oRCDescription \times oRCRole1 \times oRCRole2 \times oRCConditionsEstablishingRelationship \times oRCConditionsRemovingRelationship, oRC^o \in oRC, oRC^o \in \widehat{aRC^o}, aRC^o = elem(oCS^o, aRC), oCS^o = elem(IK^o, oCS), IK^o \in IK.$

Examples of two relationship concepts c_liesOn and c_isMother:

```
1  {
2  "name": "c_liesOn",
3  "description": "First thing lies on the second thing",
4  "role1": "c_thing",
5  "role2": "c_thing",
6  "conditionsEstablishingRelation":[{
7  "role1.y>role2.y"
8  }],
9  "conditionsRemovingRelation":[{
10 "role1.y<role2.y"
11 }]
12 },
13 {
14 "name": "c_isMother",
15 "describtion": "first person is a woman and parent of the
       second
16 person, represent the relation (parent -> child)",
17 "role1": "c_person",
18 "role2": "c_person",
19 "conditionsEstablishingRelation":[{
20 "role1.age>role2.age"
21 }]
22 }
```

Definition 9. *An **instance concept variable** oICV is a JSON object representing an exemplary instance (not existing in occurrences in IK) about which we do not have any knowledge besides the knowledge about belonging to instance concepts. It consists of the following elements:*

```
1 {
2 "name": oICVName,
3 "instanceConcept": oICName
4 }
```

- ***oICVName***—*is a JSON string containing a name uniquely identifying the instance concept variable,*
- ***oICName***—*is the oICName identifying an existing instance concept,*

$\exists oIC^o : aIC^o = elem(oCS^o, aIC) \land oIC \in \widehat{aIC^o} \land oICName^o = elem(oIC^o, oICName) \land oCS^o = elem(IK^o, oCS) \land IK^o \in IK, oICV = oICVName \times oICName.$

Definition 10. *A state concept oSC is a JSON object describing the possible state of the fragment of the IK. It consists of the following elements:*

```
1 {
2 "name": oSCName,
3 "instanceConceptVariables": oSCInstanceConceptVariables,
4 "relationships": oSCRelationships
```

```
5 }
```

- ***oSCName***—*is a JSON string containing a name uniquely identifying the state concept (optional),*
- ***oSCInstanceConceptVariables***—*is a JSON array containing a finite set of instance concept variables* $oICV$ *(optional),*
  $oSCInstanceConceptVariables^o = (oICV_1^o, ..., oICV_M^o)$,
- ***oSCRelationships***—*is a JSON array containing a finite set of relationship* oR *(Definition 17) occurring between instance concept variables indicated in instanceConceptVariables (optional),*
  $oSCRelationships^o = (oSCRelationship_1^o, ..., oSCRelationship_N^o)$,

$\forall oSCRelationship^o \in \widehat{oSCRelationships^o} \exists oRC^o : oSCRelationship^o \in oR \wedge oSCRelationships^o = elem(oSC^o, oSCRelationships) \wedge oSC^o \in oSC \wedge oRC^o \in \widehat{aRC^o} \wedge aRC^o = elem(oCS^o, aRC) \wedge elem(oSCRelationship^o, oRCName) = elem(oRC^o, oRCName) \wedge oCS^o = elem(IK^o, oCS) \wedge IK^o \in IK$.

$oSC = oSCName \times oSCInstanceConceptVariables \times oSCRelationships$.

Example of a state concept:

```
1 {
2 "instanceConceptVariables": [
3 {
4 "name": "thing1",
5 "instanceConcept": "c_thing"
6 },
7 {
8 "name": "thing2",
9 "instanceConcept": "c_thing"
10 }],
11 "relationships": [
12 {
13 "concept": "c_liesOn",
14 "role1": "thing1",
15 "role2": "thing2"
16 }]
17 }
```

5.5. Action Concepts

Definition 11. *An action concept oAC is a JSON object defining a set of actions, each of which can transform a specified state of a fragment of the IK into another specified state of a fragment of the IK.*

```
1 {
2 "name": oACName,
3 "initialStateConcept": oACinputStateConcept,
4 "finalStateConcept": oACoutputStateConcept
5 }
```

The action concept consists of:

- ***oACName***—*is a JSON string containing a name that uniquely identifies the action concept,*
- ***oACinputStateConcept***—*a state concept oSC before the start of the action, regarding instances to be transformed as a part of the implementation of the action,*
- ***oACoutputStateConcept***—*a state concept oSC expected to be after completion of the action, regarding instances to be transformed as a part of the implementation of the action,*

$oAC = oACName \times oACinputStateConcept \times oACoutputStateConcept, oAC^o \in oAC$, $oAC^o \in \widehat{aAC^o} = elem(oCS^o, aAC), oCS^o = elem(IK^o, oCS), IK^o \in IK$.

The action concept can be compared to the abstract method or interface in the programming language like C# or Java.

Example of the action concept:

```
1 {
2 "name": "putOn",
3 "initialStateConcept": {
4 "instanceConceptVariables": [
5 {
6 "name": "thing1",
7 "instanceConcept": "c_thing"
8 },
9 {
10 "name": "thing2",
11 "instanceConcept": "c_thing"
12 }
13 ],
14 "relationships": []
15 },
16 "finalStateConcept": {
17 "instanceConceptVariables": [
18 {
19 "name": "thing1",
20 "instanceConcept": "c_thing"
21 },
22 {
23 "name": "thing2",
24 "instanceConcept": "c_thing"
25 }],
26 "relationships": [
27 {
28 "concept": "c_liesOn",
29 "role1": "thing1",
30 "role2": "thing2"
31 }]
32 }
33 }
```

6. Occurrences in GEDL Language

In this section, the following elements of the GEDL language are defined: occurrences, the feature usage, the feature set usage, the action, the instance, the relationship, and the instance concept membership.

Definition 12. *Occurrences oOC—is an JSON object containing a finite set of instances and relationship occurrences observed by the individual. Occurrences include:*

```
1 {
2 "instances": aIN,
3 "relationships": aRS
4 },
```

- ***aIN***—*is a JSON array containing a finite set of instances oI* $aIN^o = (oI^o_1, ..., oI^o_J)$,
- ***aRS***—*is a JSON array containing a finite set of relationships oR,* $aRS^o = (oR^o_1, ..., oR^o_K)$,

where oI is defined in Definition 16, and oR is defined in Definition 17.

$$oOC = aIN \times aRS, oOC^o = elem(IK^o, oOC), IK^o \in IK.$$

Definition 13. *A feature usage* oFU *is an JSON object representing values assigned to the existing feature* oF *of the instance* oI*. The* oFU *is composed of the following elements:*

```
1 {
2 "name": oFName,
3 "value": oFUValue,
4 }
```

where:

- ***oFName***—*is a JSON string representing the name of the feature* oF *existing in the conceptual system* oCS,
  $\exists oF^o : oFName^o = elem(oF^o, oFName) \wedge oF^o \in \widehat{aF^o} \wedge aF^o = elem(oCS^o, aF) \wedge oCS^o = elem(IK^o, oCS), IK^o \in IK.$
- ***oFUValue***—*is a JSON value assigned to the feature from the* $oFDomain$ *of* oF,
  $\exists oF^o : oFName^o = elem(oF^o, oFName) \wedge oFDomain^o = elem(oF^o, oFDomain) \wedge oFUValue^o \in oFDoFDomain^o \wedge oF^o \in aF^o \wedge aF^o = elem(oCS^o, aF) \wedge oCS^o = elem(IK^o, oCS) \wedge IK^o \in IK.$

  $oFU = oFName \times oFUValue$

  The example of the usage of three features is presented below:

```
1 "features":[{
2 "name": "tastiness",
3 "value": 0.3
4 },
5 {
6 "name": "weight",
7 "value": 50
8 },
9 {
10 "name": "bloodGroup",
11 "value": "A"
12 }]
```

Definition 14. *The feature set usage* $oFSU$ *is an JSON object representing values assigned to the existing feature set* oFS*. The* $oFSU$ *is:*

```
1 {
2 "name": oFSName,
3 "values": aFSUvalues
4 },
```

where:

- ***oFSName***—*is a JSON string containing a name of the feature set* oFS *existing in* oCS,
  $\exists oFS^o : oFSName^o = elem(oFS^o, oFSName) \wedge oFS^o \in \widehat{aFS^o} = elem(oCS^o, aFS) \wedge oCS^o = elem(IK^o, oCS) \wedge IK^o \in IK,$
- ***aFSUvalues***—*is a JSON array containing an ordered set of values assigned to features in the feature set,*
  $aFSUvalues^o = (oFSUValue_1^o, ..., oFSUValue_p^o, ..., oFSUValue_P^o).$
  All of the values should belong to the domains of the features,
  $\forall_{p \in \{1,...,P\}} \exists oF_p^o \exists oFS^o : oF_p^o \in aF^o = elem(oCS^o, aF) \wedge oFS^o \in aFS^o = elem(oCS^o, aFS) \wedge oFSName^o = elem(oFS^o, oFSName) \wedge oFName_p^o \in aFNames^o = elem(oFS^o, aFNames) \wedge oFName_p^o = elem(oF_p^o, oFName) \wedge oFDomain_p^o = elem(oF_p^o, oFDomain) \wedge$

$oFUValue_p^o \in oFD_{oFDomain_p^o} \wedge oCS^o = elem(IK^o, oCS) \wedge IK^o \in IK$, *where* $P = card(\widehat{aFSUvalues^o})$, *function* $card()$ *return number of elements of a set.*
$oFSU = oFSName \times aFSUvalues.$

The example of the feature set usage is presented below:

```
1 {
2 "name: "color",
3 "values": [255, 20, 60]
4 }
```

Definition 15. *An action oA is a JSON object containing a description of an action, which can be potentially performed by the instance to which it is assigned, and result in a change in the state of the fragment of IK^o. An instance uses its own capabilities to carry out an action. It can also use other instances and their actions. The action consists of:*

```
1 {
2 "name": oAName,
3 "actionConcept": oACName,
4 "parameters": oAParameters,
5 "initialConditions": oAInitialConditions,
6 "successProbability": oASuccessProbability
7 }
```

- ***oAName**—is a JSON string containing a name uniquely identifying the action,*
- ***oACName**—name $oACName$ of the existing action concept oAC that precisely defines the transformation of the concepts of states within the action, $\exists oAC^o : oACName^o = elem(oAC^o, oACName) \wedge oAC^o \in \widehat{aAC^o} \wedge aAC^o = elem(oCS^o, aAC) \wedge oCS^o = elem(IK^o, oCS) \wedge IK^o \in IK$,*
- ***oAParameters**—is a JSON array containing instances, values, relationships and any information necessary for the action accomplishment. The parameters can contain, for example, instances that are necessary tools or instances changed while taking the action (optional).*
- ***oAInitialConditions**—is a JSON array containing strings built of a source code written in a programming language, having initial conditions specifying what must be fulfilled to perform the action (optional). The conditions must relate to the instances and the environment in which the action operates. The conditions include:*
  - *a state concept of a fragment of the IK which has to be attained to start the action,*
  - *ranges of the feature values (except the values indicated in the instance concepts) for the instances,*
  - *relationships to which instances must belong or not.*
- ***oASuccessProbability**—is a JSON number containing probability of success. It determines how likely it is that the activity will be carried out correctly in the environment (optional). $oASuccessProbability \in \mathbb{R}$, $oASuccessProbability \in [0, 1]$.*

$oA = oAName \times oACName \times oAParameters \times oAInitialConditions \times oASuccessProbability$

Actions are assigned only to those instances which are able to perform actions autonomously. For example, a chopping knife will not have a slice action because it cannot carry out operations autonomously. However, a bread slicer machine, automatically cutting bread, will have a slicing action only if, after starting the operation, it can make its own decision to end the action.

Actions are atomic and we do not specify any sub-actions for them. Complex actions are problems with their solutions.

Example of an action:

```
1 {
2 "name": "putOn",
3 "actionConcepts": "c_putOn",
4 "parameters": [
5 {
6 "name": "thing1", "instanceConcept": "c_thing"
7 },
8 {
9 "name": "thing2", "instanceConcept": "c_thing"
10 }
11 ],
12 "initialConditions": [
13 {"return {concept: c_liesOn,
14 role1: thing1,
15 role2: thing2
16 } not in relationships"}
17 ]
18 }
```

Definition 16. *An instance oI is the knowledge about an object observed and distinguished in the environment by the individual. The instance consists of the following elements:*

```
1 {
2 "name": oIName,
3 "features": oIFFeatures,
4 "featureSets": oIFFeatureSets
5 "actions": oIActions
6 }
```

- ***oIName***—*a JSON string containing the name of an instance oI uniquely identifying the instance (optional),*
- ***oIFFeatures***—*is a JSON array containing a finite set of features usage oFU (optional),* $oIFFeatures^o = (oFU_1^o, ..., oFU_R^o)$,
- ***oIFFeatureSets***—*is a JSON array containing a finite set of feature sets usage oFSU (optional),* $oIFFeatureSets^o = (oFSU_1^o, ..., oFSU_S^o)$,
- ***oIActions***—*is a JSON array containing a finite set of actions oA distinguished for the instance by the individual (optional),* $oIActions^o = (oA_1^o, ..., oA_T^o)$.

$$oI = oIName \times oIFeatures \times oIActions,$$
$$oI^o \in oI, oI^o \in \widehat{aIN^o} = elem(oOC^o, aIN), oOC^o = elem(IK^o, oOC), IK^o \in IK.$$

Example of an instance apple:

```
1 {
2 "name": "apple",
3 "features":[
4 {
5 "name": "tastiness",
6 "value": 0.9
7 }
8 ],
9 "featureSets":[
10 {
11 "name": "color",
12 "values": [255, 20, 60]
```

```
13 }
14 ]
15 }
```

As it was mentioned in Definition 2, the individual is also an instance and can have all of the attributes belonging to the instance. The name of the instance representing the individual is "myself".

Definition 17. *A relationship oR is a JSON object representing a connection distinguished by the individual between two instances oI. The relationship includes:*

```
1 {
2 "relationshipConcept": oRCName,
3 "role1": oRRole1,
4 "role2": oRRole2
5 }
```

where:

- ***oRCName***—*is a JSON string containing a name of an existing relationship concept oRC. The relationship concept defines the relationship.* $\exists oRC^o : oRC^o \in \widehat{aRC^o} \wedge aRC^o = elem(oCS^o, aRC) \wedge oRCName^o = elem(oRC^o, oRCName) \wedge oOC^o = elem(IK^o, oOC) \wedge IK^o \in IK$,
- ***oRRole1***—*is a JSON string containing* $oIName^o$ *of* oI^o *which is the first parameter in this relationship. The instance has to belong to the first instance concept* oIC^o*, which name* $oRCRole1^o$ *is indicated in the relationship concept,*
  $\exists oI^o \exists oRC^o \exists oIC^o : oRRole1^o = elem(oI^o, oIName) \wedge oRC^o \in \widehat{aRC^o} \wedge aRC^o = elem(oCS^o, aRC) \wedge oRCName^o = elem(oRC^o, oRCName) \wedge oOC^o = elem(IK^o, oOC) \wedge oRCRole1^o = elem(oRC^o, oRCRole1) \wedge aIC^o = elem(oCS^o, aIC) \wedge oIC^o \in \widehat{aIC^o} \wedge oRCRole1^o = elem(oIC^o, oICName) \wedge oI^o \in ICM(oIC^o) \wedge IK^o \in IK$, *where function* $ICM()$ *is defined in Definition 18.*
- ***oRRole2***—*is a JSON string containing* $oIName^o$ *of* oI^o *which is the second parameter in this relationship. The instance has to belong to the second instance concept* oIC^o*, which name* $oRCRole2^o$ *is indicated in the relationship concept,*
  $\exists oI^o \exists oRC^o \exists oIC^o : oRRole2^o = elem(oI^o, oIName) \wedge oRC^o \in \widehat{aRC^o} \wedge aRC^o = elem(oCS^o, aRC) \wedge oRCName^o = elem(oRC^o, oRCName) \wedge oOC^o = elem(IK^o, oOC) \wedge oRCRole2^o = elem(oRC^o, oRCRole2) \wedge oIC^o \in \widehat{aIC^o} \wedge aIC^o = elem(oCS^o, aIC) \wedge oRCRole2^o = elem(oIC^o, oICName) \wedge oI^o \in ICM(oIC^o) \wedge IK^o \in IK$,

  $oR = oRCName \times oRRole1 \times oRRole2$,
  $oR^o \in \widehat{aRS^o} = elem(oOC^o, aRS), oOC^o = elem(IK^o, oOC), IK^o \in IK$.

  For example:

```
1 {
2 "relationshipConcept": "c_isMother",
3 "role1": "Anna",
4 "role2": "Peter"
5 },
6 {
7 "relationshipConcept": "c_isCitizen",
8 "role1": Andrew,
9 "role2": Poland
10 }
```

Definition 18. *Let* $oIC^o \in oIC$*. An instance concept membership set* $ICM(oIC^o)$ *of the instance concept* oIC^o *is a set defined as follows:*

$$ICM(oIC^o) = \{oI^o \in oI : \forall oICFeature^o \in oICFeatures^o = elem(oIC^o, oICFeatures) \exists oFName^o \exists oFUValue^o \exists oFU^o \exists oFSU^o \exists oFS^o [(oFU^o \in \widehat{oIFFeatures^o} = elem(oI^o, oIFFeatures) \wedge oFName^o = elem(oFU^o, oFName) \wedge oICFeature^o \in \widehat{oICFeatures^o} \wedge oFName^o = elem(oICFeature^o, oFName) \wedge oICFRange^o = elem(oICFeature^o, oICFRange) \wedge oFUValue^o = elem(oFU^o, oFUValue) \wedge oFUValue^o \in oICFR_{oICFRange^o})$$

$$\vee$$

$$(oIFFeatureSets^o = elem(oI^o, oFSU) \wedge oFSU^o \in \widehat{oIFFeatureSets^o} \wedge aFSUvalues^o = elem(oFSU^o, aFSUvalues) \wedge oFS^o \in \widehat{aFS^o} = elem(oCS^o, aFS \wedge elem(oFSU^o, oFSName) = elem(oFS^o, oFSName \wedge \exists u \in \{1, ..., card(aFSUvalues^o)\}(oFName^o = pos(elem(oFS^o, aFNames), u) \wedge oFUValue^o = pos(elem(oFSU^o, aFSUvalues), u)) \wedge oFName^o = elem(oICFeature^o, oFName) \wedge oICFRange^o = elem(oICFeature^o, oICFRange) \wedge oFUValue^o \in oICFR_{oICFRange^o})]$$

$$\wedge$$

$$\forall oICRC^o \in \widehat{oICRelationshipConcepts^o} = elem(oIC^o, oICRelationshipConcepts) \exists oICRName^o \exists oICRRole^o \exists oR^o [oICRName^o = elem(oICRC^o, oICRName) \wedge oICRRole^o = elem(oICRC^o, oICRRole) \wedge oR^o \in \widehat{aRS^o} = elem(oOC^o, aRS) \wedge oICRName^o = elem(oR^o, oRCName) \wedge ((oICRRole^o =' role1' \wedge elem(oR^o, oRRole1) = elem(oI^o, oIName)) \vee (oICRRole^o =' role2' \wedge elem(oR^o, oRRole2) = elem(oI^o, oIName))]$$

$$\wedge$$

$$\forall oACName^o \in \widehat{oICActionConcepts} = elem(oIC^o, oICActionConcepts) \exists oA^o [OA^o \in \widehat{oIActions^o} = elem(oI^o, oIActions) \wedge oACName^o = elem(oA^o, oAActionConcept)]$$

$$\wedge$$

$$\forall oICName^o \exists oIC^o [oICName^o \in^o ICInstanceConcepts^o = elem(oIC^o, oICInstanceConcepts) \wedge oIC^o \in oIC \wedge oICName^o = elem(oIC^o_k, oICName \wedge oI^o \in ICM(oIC^o)]\}$$

The defined set $ICM(oIC^o)$ allows us to determine whether a particular instance oI^o belongs to the instance concept oIC^o or not. We check in the definition several conditions the instance oI^o has to fulfill in order to belong to the set $ICM(oIC^o)$. At first, values of the features and feature sets of oI^o are checked in order to determine whether they are within the ranges of features and feature sets of the instance concepts oIC^o or not. Then, it is checked if the oI^o has relationships belonging to appropriate relationship concepts. Next, actions of the oI^o are tested. In the end, we check if the oI^o is in the $ICM()$ sets of other listed instance conceps.

7. Experience in GEDL Language

The experiences in the IK contain problems and their solutions. A solution to the problem is composed of actions performed by an individual or other instances used as tools.

Definition 19. *A problem oP is a JSON object containing a description of a complex task that the individual should perform. The problem may be ordered to be solved by an entity that is not represented in the IK^o. The problem must be formulated using terms understandable to the individual, i.e., those belonging to the IK^o. The problem consists of:*

```
1 {
2 "name": oPName,
3 "individualKnowledgeFragment": oPIndividualKnowledgeFragment,
4 "initialStateConcept": oPInitialStateConcept,
5 "finalStateConcept": oPFinalStateConcept,
6 "goalFunction": oPGoalFunction
7 }
```

- ***oPName***—*is a JSON string containing a name uniquely identifying the problem (optional),*
- ***oPIndividualKnowledgeFragment***—*is a JSON array containing names of instances from occurrences oOC in IK which can be used to solve the problem oP,*

$oPIndividualKnowledgeFragment^o = (oI_1^o, ..., oI_N^o)$,
$oPIndividual\widehat{KnowledgeFragment^o} \subseteq \widehat{aIN^o}$,
$aIN^o = elem(oOC^o, aIN), oOC^o = elem(IK^o, oOC), IK^o \in IK$,

- ***oPInitialStateConcept***—*is a state concept oSC before solving the problem,* $oPInitialStateConcept^o \in oSC$,
- ***oPFinalStateConcept***—*is a state concept oSC after solving the problem,* $oPFinalStateConcept^o \in oSC$,
- ***oPGoalFunction***—*a goal function oGF (optional).*

$oP = oPName \times oPIndividualKnowledgeFragment \times oPInitialStateConcept \times oPFinalStateConcept \times oPGoalFunction$.

Section 8 contains an example of a problem and a solution to the problem.

Definition 20. *A problem solution oPS is a JSON object containing a solution to a given problem. The solution contains a list of actions to be performed in a specific order or concurrently/parallelly, and instances that are used to enable the transition from the initial state concept to the final state concept.*

The problem solution oPS is constructed in the following way:

```
1 {
2 "name": oPSName,
3 "actions": aPSActions,
4 "goalFunctionValue": oPSGoalFunctionValue
5 }
```

- ***oPSName***—*is a JSON string containing a name uniquely identifying the problem solution,*
- ***oPSActions***—*is a JSON array containing strings built of a source code written in a programming language, containing actions oA of a given individual or other instances indicated in oPIndividualKnowledgeFragment. Values of parameters or objects should be added to the listed actions. An exemplary code written in an object language may look as follows:*

```
1 myself.putOn(gumdropSnake,cake)
```

- ***goalFunctionValue***—*is a JSON number containing a value of the performed goal function oGF for a given solution.*

A problem can have many solutions. A description of the implementation of the action can be made in a specific programming language. The implementation may consist of actions of the myself instance or actions of other instances used in the solution.

$oPS = oPSName \times oPIndividua \times oPSActions \times oPSGoalFunctionValue$

Definition 21. *A goal function oGF takes the form of JSON strings built of a source code written in a programming language and contains a method for calculating a value eval* $\in \mathbb{R}$ *that enables the evaluation of solutions to the problem,* $oGF^o(oPS^o) = eval$. *The bigger the eval is, the better the solution is evaluated.*

Definition 22. *An experience oE is a JSON object containing problems and solutions developed by the individual. The experience is defined as follows:*

```
1 {
2 "problem": oEProblem,
3 "problemSolutions": oEProblemSolutions,
4 "bestSolution":oEBestSolution
5 }
```

where:

- ***oEProblem***—*a problem oP*, $oEProblem^o \in oP$,
- ***oEProblemSolutions***—*is a JSON array containing a finite set of different problem solutions oPS*, $oEProblemSolutions^o = (oPS_1^o, ..., oPS_K^o)$,
- ***oEBestSolution***—*is a oPSName of the best solution to the problem, for which the goal function obtains the highest rated value*, $oEBestSolution^o = elem(oPS_p^o, oPSName)$, *where* $oEProblemSolutions^o = elem(oE^o, oEProblemSolutions)$, $p = max_k = 1^K oGF^o(oPS_k^o)$, $oPS_p^o = pos(oEProblemSolutions^o, p)$, $K = card(\widehat{oEProblemSolutions^o})$.

$oE = oEProblem \times oEProblemSolutions \times oPSActions \times oEBestSolution, oE^o \in \widehat{aE^o}, aE^o = elem(IK^o, aE), IK^o \in IK$.

8. Exemplary Individual Knowledge

This chapter presents an exemplary content of the IK. It can be divided into three sections: conceptualSystem oCS, occurrences oOC and experience aE.

8.1. The Conceptual System

The conceptual system oCS is the first part of the IK. The listing below presents a part of the IK containing the oCS. At the beginning, feature definitions (their possible ranges and default values) and feature sets are introduced. Next, definitions of concepts: instance concepts, relationship concepts, and action concepts, are presented based on previously defined features and feature sets. These concepts are a part of the IK, which helps to understand the surrounding environment and how to influence it.

```
1  {
2  "conceptualSystem":{
3  "features": [
4  {
5  "name": "sweetness",
6  "description": "Describes how sweet the thing is",
7  "domain":{
8  "set": "Real",
9  "min": 0,
10 "max": 1
11 },
12 "default": 0
13 },
14 {
15 "name":"edible",
16 "description": "Determines if the item is edible",
17 "domain":{
18 "set": [true,false]
19 },
20 "default": false
21 },
22 {
23 "name": "solid",
24 "description": "Describes how solid thing is",
25 "domain":{
26 "set": "Real",
27 "min": 0,
28 "max": 1
29 },
30 "default": 0
31 },
32 {
```

```
33 "name": "red",
34 "description": "Intensity of red color",
35 "domain":{
36 "set": "Integer",
37 "min": 0,
38 "max": 255
39 },
40 "default": 0
41 },
42 {
43 "name": "green",
44 "description": "Intensity of green color",
45 "domain":{
46 "set": "Integer",
47 "min": 0,
48 "max": 255
49 },
50 "default": 0
51 },
52 {
53 "name": "blue",
54 "description": "Intensity of blue color",
55 "domain":{
56 "set": "Integer",
57 "min": 0,
58 "max": 255
59 },
60 "default": 0
61 },
62 {
63 "name": "beauty",
64 "description": "Describes the beauty",
65 "domain":{
66 "set": "Real",
67 "min": 0,
68 "max": 1
69 },
70 "default": 0
71 },
72 {
73 "name": "softness",
74 "description": "Describes the softness of material,
75 1 means very soft",
76 "domain":{
77 "set": "Real",
78 "min": 0,
79 "max": 1
80 },
81 "default": 0
82 }
83 ],
84 "featureSets":[
85 {
86 "name": "color",
```

```
87 "features":[
88 "red",
89 "green",
90 "blue"
91 ],
92 "featureSets":[]
93 }
94 ],
95 "instanceConcepts": [
96 {
97 "name": "c_sweets",
98 "features": [
99 {
100 "name": "sweetness",
101 "range": {
102 "min": 0.7,
103 "max": 1
104 }
105 },
106 {
107 "name": "edible",
108 "range": {
109 "set": [true]
110 }
111 }
112 ]
113 },
114 {
115 "name": "c_thing",
116 "features": [
117 {
118 "name": "solid",
119 "range": {
120 "min": 0.6,
121 "max": 1
122 }
123 }
124 ]
125 },
126 {
127 "name": "c_solidSweets",
128 "contain": [
129 "c_sweets",
130 "c_thing"
131 ]
132 },
133 {
134 "name": "c_sweetDecoration",
135 "contain": [
136 "c_solidSweets"
137 ],
138 "features":[
139 {
140 "name": "beauty",
```

```
141 "range": {
142 "min": 0.6,
143 "max": 1
144 }
145 }
146 ]
147 }
148 ],
149 "relationshipConcepts": [
150 {
151 "name": "c_liesOn",
152 "description": "First thing lies on second thing",
153 "role1": "c_thing",
154 "role2": "c_thing"
155 }
156 ],
157 "actionConcepts": [
158 {
159 "name": "c_putOn",
160 "initialStateConcept": {
161 "instanceConceptVariables": [
162 {
163 "name":"thing1",
164 "instanceConcept": "c_thing"
165 },
166 {
167 "name":"thing2",
168 "instanceConcept": "c_thing"
169 }
170 ],
171 "relationships": []
172 },
173 "finalStateConcept": {
174 "instanceConceptVariables": [
175 {
176 "name":"thing1"
177 "instanceConcept": "c_thing"
178 },
179 {
180 "name":"thing2"
181 "instanceConcept": "c_thing"
182 }
183 ],
184 "relationships": [
185 {
186 "concept": "c_liesOn",
187 "role1": "thing1",
188 "role2": "thing2"
189 }
190 ]
191 }
192 },
193 {
194 "name": "c_boilWater",
```

```
195 "initialStateConcept": {
196 "instanceConceptVariables": [
197 {
198 "name":"myWater",
199 "instanceConcept": "c_water"
200 }
201 ]
202 },
203 "finalStateConcept": {
204 "instanceConceptVariables": [
205 {
206 "name":"myWater",
207 "instanceConcept": "c_boilingWater"
208 }
209 ]
210 }
211 }
212 ]
213 },
```

The conceptual system oCS contains instance concepts (c_sweets, c_thing, c_solidSweets, and c_sweetDecoration). These concepts have features with restrictions on the range of their values. The individual classifies the observed instances based on their features. If a given instance has the given features and their values are within certain ranges, then according to the individual knowledge, this instance belongs to this concept. The c_solidSweets concept contains the c_sweets and c_thing concepts, thus contains all their features with appropriate ranges of values. If a concept contains features from two other concepts that have the same features, the resulting feature in this concept has restrictions that are an intersection of the limits of these two concepts.

The next part of the listing describes the relationship concept c_liesOn. This concept defines the relationship between two instances. In the beginning, there is a JSON string with a description of the relationship in a natural language. Then, the two instance concepts (to which the instances must belong) are defined.

The action concepts (c_putOn and c_boilWater) are placed further in the individual conceptual system oCS. It should be noted that in their structure there is no description of how they work. On the input (initialStateConcept), there is a description of a part of the environment that is found before the action is carried out. On the output (finalStateConcept), there is a description of the same part of the environment with all the changes made by this action. Knowledge of the result that should be obtained allows the individual to verify whether the action has been carried out correctly and to take appropriate steps in case of failure. The part initialStateConcept lists the instances (and instance concepts to which they belong) and their relationships. The part finalStateConcept consists of the same concepts (instance concepts or relationship concepts) that were listed in the part initialStateConcept, and which will not be liquidated as a result of the action, and also consists of concepts that will arise as a result of the performed action. For example, the action concept c_putOn needs two instances belonging to the c_thing concept as input. After completing this action, these two instances should still exist but should be in the c_liesOn relationship.

8.2. The Occurrences

The next part of IK includes occurrences oOC of instances (chocolateStar, fish, cake, hotChocolate, gumdropSnake, raisins) and their relationships (currently an empty set). These are instances representing objects recognized by the individual in the surrounding environment. Instances have specific features with specific values and may have actions that can be performed with these instances. In a conceptual system oCS, there may be many actions that implement the same task, but depending on the instance that implements

it, additional assumptions (initialConditions) or parameters may be needed. One of the instances is myself—it defines the individual in which the current IK is located. The putOn action described in the myself instance can be used by the individual to perform the action by itself.

```
213 "occurrences":{
214 "instances": [
215 {
216 "name":"myself",
217 "actions": [
218 {
219 "name": "putOn",
220 "actionConcept": "c_putOn",
221 "parameters": [
222 {"name":"thing1", "instanceConcept":"c_thing"},
223 {"name":"thing2", "instanceConcept":"c_thing"}
224 ],
225 "initialConditions": [],
226 "successProbability" : 0.95
227 }
228 ]
229 },
230 {
231 "name": "chocolateStar",
232 "features":[
233 {"name": "sweetness", "value": 0.9},
234 {"name": "edible", "value": true},
235 {"name": "solid", "value": 0.9},
236 {"name": "beauty", "value": 0.9}
237 ],
238 "featureSets":[
239 {"name": "color", "values": [165, 42, 42]}
240 ]
241 },
242 {
243 "name": "fish",
244 "features":[
245 {"name": "edible", "value": true},
246 {"name": "sweetness", "value": 0.0},
247 {"name": "solid", "value": 1},
248 {"name": "beauty", "value": 0.2}
249 ]
250 },
251 {
252 "name": "cake",
253 "features":[
254 {"name": "sweetness", "value": 0.8},
255 {"name": "edible", "value": true},
256 {"name": "solid", "value": 0.7},
257 {"name": "softness", "value": 0.7}
258 ]
259 },
260 {
261 "name": "hotChocolate",
262 "features":[
```

```
263 {"name": "sweetness", "value": 0.9},
264 {"name": "edible", "value": true},
265 {"name": "solid", "value": 0.1},
266 {"name": "temperature", "value": 60}
267 ]
268 },
269 {
270 "name": "gumdropSnake",
271 "features":[
272 {"name": "sweetness", "value": 1},
273 {"name": "edible", "value": true},
274 {"name": "solid", "value": 0.8},
275 {"name": "beauty", "value": 0.7}
276 ]
277 },
278 {
279 "name": "raisins",
280 "features":[
281 {"name": "sweetness", "value": 1},
282 {"name": "edible", "value": true},
283 {"name": "solid", "value": 1},
284 {"name": "beauty", "value": 0.6}
285 ]
286 }
287 ],
288 "relationships": []
289 },
```

8.3. The Experience

The listing below presents a part of the IK representing experience aE. It consists of problems to be solved by the individual and solutions. In order to initiate the search for solutions, the problem should be defined—the part of the environment that can be used (individualKnowledgeFragment) and the transformation that we want to achieve (from initialStateConcept to finalStateConcept) must be indicated. To achieve the optimal solution, the goal function should also be provided.

```
290 "experience": [
291 {
292 "problem" : {
293 "name" : "Decorate cake",
294 "individualKnowledgeFragment": {
295 "instances": [
296 "chocolateStar",
297 "fish",
298 "hotChocolate",
299 "gumdropSnake",
300 "cake"
301 ]
302 },
303
304 "initialStateConcept": {
305 "instanceConceptVariables": [
306 {"name":"myCake", "instanceConcept":"c_cake"},
307 {"name":"thing1", "instanceConcept":"c_sweetDecoration"},
```

```
308 {"name":"thing2", "instanceConcept":"c_sweetDecoration"}
309 ],
310 "relationships": []
311 },
312 "finalStateConcept": {
313 "instanceConceptVariables": [
314 {"name":"myCake", "instanceConcept":"c_cake"},
315 {"name":"thing1", "instanceConcept":"c_sweetDecoration"},
316 {"name":"thing2", "instanceConcept":"c_sweetDecoration"}
317 ],
318 "relationships": [
319 {
320 "concept": "c_liesOn",
321 "role1": "thing1",
322 "role2": "myCake"
323 },
324 {
325 "concept": "c_liesOn",
326 "role1": "thing2",
327 "role2": "myCake"
328 }
329 ]
330 },
331 "goalFunction":{
332 "function": "{
333 let val=0;
334 for(let i=0; i<relationships.length; i++) {
335 val+=(r.role1.beauty + r.role1.sweetness)/2;
336 });
337 return val/i;
338 }"
339 }
340 },
341 "problemSolution":[
342 {
343 "actions":[
344 "myself.putOn(chocolateStar, cake)",
345 "myself.putOn(raisins, cake)"
346 ],
347 "goalFunctionValue":"0.85"
348 },
349 {
350 "actions":[
351 "myself.putOn(chocolateStar, cake)",
352 "myself.putOn(gumdropSnake, cake)"
353 ],
354 "goalFunctionValue":"0.875"
355 },
356 {
357 "actions":[
358 "myself.putOn(raisins, cake)",
359 "myself.putOn(gumdropSnake, cake)"
360 ],
361 "goalFunctionValue":"0.825"
```

```
362 }
363 ],
364 "bestSolution":{
365 "actions":[
366 "myself.putOn(chocolateStar, cake)",
367 "myself.putOn(gumdropSnake, cake)"
368 ],
369 "goalFunctionValue":"0.875"
370 }
371 }
372 ]
373 }
```


Our objective is to have two sweet decorations on the cake, in other words, we should have two sweet decorations in the relationship "lies on" with the cake. Therefore, the individual analyses all action concepts that can lead to the creation of such relationships. The c_putOn action concept is the only one in this case. Then, the individual checks all actions in the system that belong to the given action concept. The only action that fulfills this condition belongs to the myself instance. Therefore, the individual must carry out this action itself. Then, it analyses the prerequisites needed to perform this action. In this case, the existence of three instances is the condition—one belonging to the c_cake concept and two belonging to the c_sweetDecoration concept. In the IK, three instances belong to the c_sweetDecoration concept: gumdropSnake, chocolateStar, and raisins. They can be used in potential solutions. The problem has a goal function (goalFunction) that helps to determine which of the solutions implements the given challenge most effectively. Although JSON does not allow a string on many lines, the goalFunction is divided into several lines in the listing for readability reasons. The goal function chooses the best solution based on the beauty and sweetness of concepts in the presented example. Three solutions with assessments are the result of the algorithm. As can be seen, the best solution is to decorate the cake with a chocolate star and a gel snake by performing action c_putOn twice.

8.4. The More Complex Example

The presented example has a straightforward solution because all the required instances were available and needed only the putOn action. However, we can imagine a situation in which the instance belonging to the c_sweetDecoration concept is missing, but there is a sugar instance belonging to the c_sweets concept, cocoaLion instance belonging to the c_edibleDecoration concept and a sprinkle action that will combine these instances into one that will meet the requirements of the c_sweetDecoration concept. In this way, a newly created instance (and a physically made object that has been created in the environment) can be put together with other cake decorations.

8.5. Summary of the Example

When solving tasks, the individual may decide to perform actions that may change the properties of the instance, add or remove the relationship between them, or decide to create a non-existent instance (e.g., making pancake dough or cooking an egg for a sandwich). There can be many ways to solve the problem.

Many factors can be used to assess a given solution. These may be: expected results, costs incurred, time of implementation, the uncertainty of receiving the intended solution, assessment of previous similar actions, or subjective assessment made with the help of another individual (e.g., a human being).

After the physical execution of the action in the surrounding environment, an attempt may be made to undertake the final assessment of the task. The assessment can be made using available sensors. The solution with input and output parameters and the final grade

can be saved in the IK and become the main solution in future tasks. The problem of evaluation of the adopted solution will be the topic of discussion in future articles.

9. Applications of GEDL Language and Directions of Future Work

The language presented in the article allows an individual possessing a cognitive mechanism to describe almost any environment. The way the individual will describe the environment is not imposed in advance, and two individuals of a similar type (e.g., two identical robots) can build very different IK for the same environment. These differences may be caused by different initial conditions (different time or place to start work), the stochastic nature of the environment, or a difference in tasks performed in the environment.

The GEDL language will help to combine various data and knowledge processing technologies, including artificial intelligence techniques. For example, the cognitive mechanism, apart from sensors, will require the use of techniques that enable data interpretation so that objects can be distinguished in the environment, and characterized. In this case, cameras and appropriate image recognition techniques could be used to distinguish objects and give them features such as color, shape etc.

In addition, the instance can carry out our atomic actions in the GEDL language. The way the instance processes actions is not precisely defined, and different complex techniques of artificial intelligence may also be used to carry them out. The description of actions in the new language consists of a description of the state before and after the action, so that the way the actions are executed is not narrowed down to the use of strictly defined techniques for their implementation.

So far, in the article, the definitions forming the basis of the new language have been presented, and it was shown how it may work on a simple example of IK along with the problem and its solution. The further articles will include a description of the basic operation of the product. They will sum up our research into concepts and concept states. It will be shown how to store historical states of instances (marked as memory aM) in the IK. In the future, an attempt will be made to deal with the issue of inference based on historical data. An effective way of storing IK in databases currently used will be presented.

Subsequent publications will also deal with the issue of building a conceptual system that defines the understanding and perception of the environment by an individual. It is going to be determined how a conceptual system should be constructed in an individual that starts their activities in the environment, and how such a system can be developed automatically. Building an IK shared by a larger number of individuals, so that the conceptual system remains coherent, is also a separate research problem for the future.

10. Summary

In this article, a new language called the General Environment Description Language (GEDL) was presented. The language is dedicated to being used in autonomous systems (robots, programs) working in a diverse environment. It enables the description of the environment in which the system acts and helps to make autonomous decisions.

According to the adopted conception, the autonomous system called an Individual collects its knowledge in three sections: Conceptual System, Occurrences and Experience. The Conceptual System contains concepts which, in a way, group occurrences observed in the environment. The section Occurrences presents knowledge about objects observed in the environment. Spotted objects are called instances in the Occurrences. The Experience in GEDL language section contains problems to be solved and their solutions. The Individual Knowledge in the GEDL language is described using the JSON notation.

In this article, an example was provided showing how to store the knowledge, define the problem and find the problem solutions.

In further articles, an attempt will be made to deal with the problem of describing and solving many issues that were only mentioned. Among other things, methods will be proposed for optimal problem solving, language grounding, automatic development of the conceptual system, and others.

Author Contributions: Conception of GEDL language: K.Z., W.P., A.B., A.Z. and E.P.; Matematical formalization: K.Z., A.M., A.B., and W.P.; Examples: W.P. and K.Z.; Original Draft Preparation: K.Z., A.B., W.P., A.M. and A.Z.; Editing: A.B.; Writing Review: A.B. and K.Z.; Supervision: K.Z. All authors have read and agreed to the published version of the manuscript.

Funding: This research received no external funding.

Conflicts of Interest: The authors declare no conflict of interest.

References

1. Goertzel, B.; Pennachin, C. *Artificial General Intelligence*; Springer: Berlin/Heidelberg, Germany, 2007.
2. Hernández-Orallo, J. *The Measure of All Minds. Evaluating Natural and Artificial Intelligence*; Cambridge University Press: Cambridge, UK, 2017.
3. Wang, P. On Defining Artificial Intelligence. *J. Artif. Gen. Intell.* **2019**, *10*, 1–37. [CrossRef]
4. Bhatnagar, S. Mapping Intelligence: Requirements and Possibilities. In *3rd Conference on" Philosophy and Theory of Artificial Intelligence*; Müller, V., Ed.; Springer: Cham, Switzerland, 2018; Volume 44, pp. 117–135.
5. Bassenne, M.; Lozano-Durán, A. Computational Model Discovery with Reinforcement Learning. *ArXiv* **2019**. Available online: https://arxiv.org/abs/2001.00008v1 (accessed on 10 December 2020).
6. Williams, A. The Relationship Between Collective Intelligence and One Model of General Collective Intelligence. In Proceedings of the 11th International Conference on Computational Collective Intelligence ICCCI 2019, Hendaye, France, 4–6 September 2019; Part I, Lecture Notes in Artificial Intelligence; Nguyen, N., Chbeir, R., Exposito, E., Aniorte, P., Trawiński, B., Eds.; Springer: Berlin/Heidelberg, Germany, 2019; pp. 589–600.
7. Poole, D.; Mackworth, A.K. *Artificial Intelligence. Foundations of Computational Agents*; Cambridge University Press: Cambridge, UK, 2010.
8. Poole, D.; Mackworth, A.K. *Artificial Intelligence: Foundations of Computational Agents*, 2nd ed.; Cambridge University Press: Cambridge, UK, 2017.
9. Ghallab, M.; Nau, D.; Traverso, P. *Automated Planning and Acting*; Cambridge University Press: Cambridge, UK, 2016.
10. Hutter, M. Universal Artificial Intelligence: Sequential Decisions Based On Algorithmic Probability. In *EATCS Series: Texts in Theoretical Computer Science*; Springer: Berlin/Heidelberg, Germany, 2005.
11. Thórisson, K.R.; Bieger, J.; Li, X.; Wang, P. Cumulative Learning. In *Artificial General Intelligence*; AGI 2019; Lecture Notes in Computer Science; Hammer, P., Agrawal, P., Goertzel, B., Iklé, M., Eds.; Springer: Cham, Switzerland, 2019; Volume 11654, pp. 198–208.
12. Wang, P. Natural Language Processing by Reasoning and Learning. In *Artificial General Intelligence, Lecture Notes in Computer Science*; Kühnberger, K.U., Rudolph, S., Wang, P., Eds.; Springer: Berlin/Heidelberg, Germany, 2013; Volume 7999, pp. 160–169.
13. Świechowski, M.; HyunSoo, P.; Mańdziuk, J.; Kyung-Joong, K. Recent Advances in General Game Playing. *Sci. World J.* **2015**, *2015*, 986262. [CrossRef] [PubMed]
14. Cropper, A.; Evans, R.; Law, M. Inductive general game playing. *Mach. Learn.* **2020**, *109*, 1393–1434. do:10.1007/s10994-019-05843-w.
15. Jiménez, S.; Rosa, T.D.L.; Fernández, S.; Fernández, F.; Borrajo, D. A review of machine learning for automated planning. *Knowl. Eng. Rev.* **2012**, *27*, 433–467. [CrossRef]
16. Nau, D.S. Current Trends in Automated Planning. *AI Mag.* **2007**, *28*, 43–58.
17. Sohrabi, S. AI Planning for Enterprise: Putting Theory Into Practice. In Proceedings of the Twenty-Eighth International Joint Conference on Artificial Intelligence (IJCAI-19), Macao, China, 10–12 August 2019; pp. 6408–6410. [CrossRef]
18. Cimatti, A.; Pistore, M.; Traverso, P. Automated Planning. In *Handbook of Knowledge Representation*; van Harmelen, F., Lifschitz, V., Porter, B., Eds., Elsevier B.V.: Amsterdam, The Netherlands, 2008. [CrossRef]
19. Gregory, P.; Schumann, P.H.C.; Björnsson, Y.; Schiffel, S. The GRL System: Learning Board Game Rules with Piece-Move Interactions. In *Workshop on Computer Games. International Workshop on General Intelligence in Game-Playing Agents. Communications in Computer and Information Science*; Cazenave, T., Winands, M., Edelkamp, S., Schiffel, S., Thielscher, M., Togelius, J., Eds.; Springer: Cham, Switzerland, 2016; Volume 614, pp. 130–148.
20. Basbaum, R.T.; Vaquero, T.S.; Silva, J.R. Requirements and Work Domain Analysis in Automated Planning Systems. In Proceedings of the 23rd International Conference on Automated Planning and Scheduling (ICAPS 2013), Workshop on Knowledge Engineering for Planning and Scheduling (KEPS), Rome, Italy, 10–14 June 2013.
21. Wang, P. *Non-Axiomatic Reasoning System-Exploring the Essence of Intelligence*; Indiana University: Indianapolis, IN, USA, 1996.
22. Ivanović, M.; Ivković, J.; Bădică, C. Role of Non-Axiomatic Logic in a Distributed Reasoning Environment. In *Proceedings of the International Conference on Computational Collective Intelligence*; Lecture Notes in Computer Science; Nguyen, N.T., Papadopoulos, G., Jedrzejowicz, P., Trawinski, B., Vossen, G., Eds.; Springer: Cham, Switzerland, 2017; Volume 10448, pp. 381–388.
23. Hammer, P. Adaptive Neuro-Symbolic Network Agent. In *Artificial General Intelligence, AGI 2019, Lecture Notes in Computer Science*; Hammer, P., Agrawal, P., Goertzel, B., Iklé, M., Eds., Springer: Cham, Switzerland, 2019; Volume 11654, pp. 80–90.
24. Love, N.; Hinrichs, T.; Haley, D.; Schkufza, E.; Genesereth, M. General Game Playing: Game Description Language Specification. 2008. Available online: http://logic.stanford.edu/classes/cs227/2013/readings/gdl_spec.pdf (accessed on 2 December 2019).
25. Ishida, T. Q: A Scenario Description Language for Interactive Agents. *Computer* **2002**, *35*, 42–47. [CrossRef]

26. Konnerth, T.; Hirsch, B.; Albayrak, S. JADL–An Agent Description Language for Smart Agents. In *Declarative Agent Languages and Technologies IV, DALT 2006, Lecture Notes in Computer Science*; Baldoni, M., Endriss, U., Eds.; Springer: Berlin/Heidelberg, Germany, 2006; Volume 4327, pp. 141–155.
27. Faulkner, S.; Kolp, M.; Wautelet, Y.; Achbany, Y. A Formal Description Language for Multi-Agent Architectures. In *Agent-Oriented Information Systems IV, AOIS 2006, Lecture Notes in Computer Science*; Kolp, M., Henderson-Sellers, B., Mouratidis, H., Garcia, A., Ghose, A.K., Bresciani, P., Eds.; Springer: Berlin/Heidelberg, Germany, 2008; Volume 4898, pp. 143–163.
28. Bergenti, F.; Iotti, E.; Poggi, A. Core Features of an Agent-Oriented Domain-Specific Language for JADE Agents. Trends in Practical Applications of Scalable Multi-Agent Systems, the PAAMS Collection. *Adv. Intell. Syst. Comput.* **2016**, *473*, 213–224.
29. Wang, P. *Non-Axiomatic Logic: A Model Of Intelligent Reasoning*; World Scientific: Singapore, 2013.
30. Wang, P.; Li, X.; P, P.H. Self in NARS, an AGI System. *Front. Robot. AI* **2018**, *5*, 20. [CrossRef]
31. Sredojević, D.; Vidaković, M.; Okanović, D.; Mitrović, D.; Ivanović, M. Conversion of the agent-oriented domain-specific language ALAS into JavaScript. In Proceedings of the AIP Conference, Symposium on Computer Languages, Implementations and Tools (SCLIT), Rhodes, Greece, 23–29 September 2015. [CrossRef]
32. Sredojević, D.; Vidaković, M.; Ivanović, M.; Mitrović, D. Extension of Agent-oriented Domain-specific language ALAS as a support to Distributed Non-Axiomatic Reasoning. In Proceedings of the ICIST 2017, Kopaonik, Serbia, 12–15 March 2017; Volume 2, pp. 368–372.
33. Aeronautiques, C.; Howe, A.; Knoblock, C.; McDermott, I.D.; Ram, A.; Veloso, M.; Weld, D.; SRI, D.W.; Barrett, A.; Christianson, D.; et al. *PDDL—The Planning Domain Definition Language*; Technical Report; CVC TR98003/DCS TR1165; Yale Center for Computational Vision and Control: New Haven, CT, USA, 1998.
34. Haslum, P.; Lipovetzky, N.; Magazzeni, D.; Muise, C. An Introduction to the Planning Domain Definition Language. In *An Introduction to the Planning Domain Definition Language*; Morgan & Claypool: San Rafael, CA, USA, 2019.
35. Fikes, R.E.; Nilsson, N.J. STRIPS: A New Approach to the Application of Theorem Proving to Problem Solving. *Artif. Intell.* **1971**, *2*, 189–208. [CrossRef]
36. Galuszka, A.; Swierniak, A. Planning in Multi-agent Environment Using Strips Representation and Non-cooperative Equilibrium Strategy. *J. Intell. Robot. Syst.* **2010**, *58*, 239–251. [CrossRef]
37. Harnad, S. The symbol grounding problem. *Phys. D Nonlinear Phenom.* **1990**, *42*, 335–346. [CrossRef]
38. Das, A.; Kottur, S.; Moura, J.M.; Lee, S.; Batra, D. Learning cooperative visual dialog agents with deep reinforcement learning. In Proceedings of the IEEE International Conference on Computer Vision (ICCV), Venice, Italy, 22–29 October 2017; pp. 2970–2979.
39. Sun, Y.; Singla, A.; Fox, D.; Krause, A. Building hierarchies of concepts via crowdsourcing. In Proceedings of the 24th International Conference on Artificial Intelligence, IJCAI, Buenos Aires, Argentina, 25–31 July 2015; pp. 844–853.
40. Chattopadhyay, P.; Yadav, D.; Prabhu, V.; Chandrasekaran, A.; Das, A.; Lee, S.; Batra, D.; Parikh, D. Evaluating visual conversational agents via cooperative human-ai games. In Proceedings of the Fifth Conference on Human Computation and Crowdsourcing (HCOMP), Québec City, QC, Canada, 24–26 October 2017; pp. 2–10.
41. Matuszek, C.; Bo, L.; Zettlemoyer, L.; Fox, D. Learning from unscripted deictic gesture and language for human-robot interactions. In Proceedings of the Twenty-Eighth AAAI Conference on Artificial Intelligence, AAAI'14, Québec City, QC, Canada, 27–31 July 2014, pp. 2556–2563.
42. Krishnaswamy, N.; Pustejovsky, J. A Formal Analysis of Multimodal Referring Strategies Under Common Ground. In Proceedings of the Language Resource and Evaluation (LREC 2020), Marseille, France, 11–16 May 2020.
43. Comprehensive Open Source Machine Learning Platform. Availabel online: https://www.tensorflow.org/ (accessed on 3 December 2020).
44. Keras API Libraries. Availabel online: https://keras.io/ (accessed on 3 December 2020).
45. An Open Source Machine Learning Framework. Availabel online: https://pytorch.org/ (accessed on 3 December 2020).
46. Zadeh, L.A.; Aliev, R.A. *Fuzzy Logic Theory and Applications*; World Scientific Publishing Co Pte Ltd.: Singapore, 2018.
47. Introducing JSON. Available online: https://www.json.org (accessed on 10 December 2020).
48. JsonLogic. Available online: http://jsonlogic.com (accessed on 10 December 2020).
49. JSON Application Development for IBM® Data Servers. Available online: https://www.ibm.com/support/knowledgecenter/SSEPEK_11.0.0/json/src/tpc/db2z_jsonappdev.html (accessed on 3 December 2020).
50. Ullman, J.D.; Widom, J. *A First Course in Database Systems*, 3rd ed.; Pearson Education Limited: London, UK, 2008; ISBN 9780136006374.