https://orcid.org/0000-0002-4168-7190

UDC 623.764

P.S. SAPATY[*]

A LANGUAGE TO COMPREHEND AND MANAGE DISTRIBUTED WORLDS

[*]Institute of Mathematical Machines and Systems Problems of the National Academy of Sciences of Ukraine, Kyiv, Ukraine

Анотація. *У статті представляється і більш детально розглядається Мова просторового захоплення (МПЗ), зокрема, її філософія, методологія, синтаксис, семантика, а також реалізація у розподілених системах. Як ключовий елемент розробленої моделі і Технології просторового захоплення МПЗ використовувалася в численних програмах і описувалася в ряді публікацій, включаючи і сім книг. Це надихнуло нас зосередити свою увагу у цій статті виключно на основних особливостях даної мови і її порівнянні з іншими мовами, таким чином показавши її вплив на реалізацію Проєкту розподіленого управління в Національній академії наук за активного міжнародного втручання і потужної підтримки. При порівнянні з іншими мовами програмування МПЗ відзначається високим рівнем, простотою і компактністю отриманих рішень, що пояснюється тим, що вона працює безпосередньо з розподіленими мережевими тілами у цілісному, паралельному, самонавігаційному режимі, здатному узгоджувати патерни. Це фактично перевертає з ніг на голову усталену практику і думку про те, що паралельні і розподілені обчислення є набагато складнішими, ніж послідовне програмування. У порівнянні зі спеціалізованими мовами управління бойовими задачами, орієнтованими на командування та управління військовими кампаніями, приклад програмування за допомогою МПЗ демонструє високу чіткість і ефективність для кампаній, в рамках яких здійснюється переміщення у просторі й обслуговування необхідних ресурсів, а також підтверджується її універсальна здатність для використання у розширених програмах у подібних сферах. У роботі показується відношення МПЗ до природних мов, а саме, що її глобально рекурсивну організацію можна безпосередньо використовувати для опису і аналізу структур природної мови будь-якого обсягу і складності, просто додавши для цього нові правила до визначення МПЗ. Ця мова також може ефективно замінити природні мови для високорівневого і швидкого визначення складних просторових проблем і вирішувати їх завдяки своїй потужності, компактності і формулоподібній рекурсивній природі.*

Ключові слова: *Мова просторового захоплення, паралельні і розподілені обчислення, мережа, Мова управління бойовими задачами, обробка природної мови.*

Abstract. *The paper presents and discusses the details of the Spatial Grasp Language (SGL), including its philosophy, methodology, syntax, semantics, and implementation in distributed systems. As a key element of the developed Spatial Grasp Model and Technology, SGL has been used in numerous applications and publications, including seven books. This inspired us to devote the current paper exclusively to the main features of this language and comparison with other languages as a tribute to its impact on the Distributed Management Project at the National Academy of Sciences, with strong international participation and support. The comparison with other programming languages shows high level, simplicity, and compactness of the obtained solutions, which are explained by the fact that SGL operates directly on distributed networked bodies in a holistic, parallel, self-navigation and pattern matching mode. This effectively puts upside down the established practice and opinion that parallel and distributed computing is essentially more complex than sequential programming. In comparison with specialized battle management languages oriented on command and control of military campaigns, the SGL programming example shows high clarity and efficiency for expressing campaigns with spatial movement and maintenance of the needed resources, also confirming its universal ability for extended applications in similar areas. The relation of SGL to natural languages is shown where globally recursive SGL organization can be directly used for describing and analyzing natural language structures of any volume and complexity, just by adding new*

ISSN 1028-9763. Математичні машини і системи. 2022. № 3

rules to the SGL definition. SGL can also effectively substitute natural languages for a high-level and quick definition of complex spatial problems and their solutions due to its power, compactness, and formula-like recursive nature.
Keywords: *Spatial Grasp Language, parallel and distributed computing, networking, Battle Management Language, Natural Language Processing.*

DOI: 10.34121/1028-9763-2022-3-9-27

1. Introduction

The aim of this paper is to describe and analyze the main features of the Spatial Grasp Language (SGL), which is the key element of the developed Spatial Grasp Model and Technology oriented on high-level simulation and management of large distributed systems. The other goal is to compare SGL capabilities with other existing languages: popular programming languages especially oriented on parallel, distributed and mobile computing; special command and control languages for the management of military campaigns; and natural languages in general. SGT and SGL have been investigated in numerous applications in diverse areas, including graph and network theory, artificial intelligence, distributed network management, large systems management and control, crises and disaster management, withstanding global pandemics, battlefield command and control, collective robotics, social systems, international security, gestalt philosophy and theory, global awareness and consciousness, space-based systems, collective behavior of animals, complex terrain investigation, city transport problems, etc. Trial implementations of previous versions of SGL named WAVE were made and practically used in different countries. Also, they were available as the public domain on the Internet. The central philosophical, theoretical and methodological point of all the mentioned investigations, applications, and related publications was the same basic high-level, universal and globally recursive language SGL having very efficient and unified network implementation which influenced the main orientation of this paper as a particular tribute to it.

The rest of the paper is organized as follows. Section 2 describes the basics of the Spatial Grasp Model and Technology, including the types of worlds SGT operates with, SGL universal and global recursive structure, some details of how SGL scenarios evolve in distributed spaces, and the repertoire of SGT control states allowing effective management of any distributed scenarios in SGL.

Section 3 provides the details of SGL, including its constants which may reflect virtual or physical objects, different types of variables being stationary or mobile, and an extendable repertoire of SGL rules comprising a variety of descriptive, data processing and space navigation operations, as well as local and global management and control.

Section 4 gives some details of distributed and networked SGL interpretation, as well as the basic components of the SGL interpreter and its dynamically created spatial tracking system allowing for effective distributed management and control of parallel self-spreading SGL scenarios.

Section 5 presents a short review of other existing programming languages, among which there are currently the most popular and also the ones oriented on parallel, distributed and mobile computing. The section also compares their features with SGL capabilities and orientation. An example is shown how programming with the use of traditional languages fundamentally differs from programming with SGL. The example is Dijkstra's Shortest Path Algorithm in C++ with an extremely clear and compact solution in SGL obtained due to its natural parallel and distributed self-navigation of a distributed network body. Additionally, there are shown similar examples of the use of SGL, including outputting all the paths between nodes, finding network articulation points, and matching graph patterns with network topologies.

Section 6 compares SGL with battle management languages oriented on practical description and management of military campaigns, which can be executed by both manned and unmanned components. The section reviews Battle Management Language (BML) and its variant as

Geospatial Battle Management Language (GeoBML) for terrain reasoning. Moreover, it shows how to use SGL to describe a controlled synchronized campaign evolving through a sequence of physical locations and performing planned actions there, and delivering the needed supplies. This confirms the potential applicability of SGL for such classes of problems, which may also serve as a universal rather than specialized language for even broader applications.

Section 7 discusses the possible relation of SGL to natural languages, where knowledge of their structure and syntax is helpful in many areas. It reviews Natural Language Processing (NLP) as the process of determining the syntactic structure of a text consisting of words and their combinations as phrases which may form complex hierarchical syntactic structures. It shows how the global recursive organization of SGL can naturally describe any natural language structures, imaginable and even so far unimaginable, and of any volume and complexity, just by adding new rules within the SGL syntax.

Section 8 concludes the paper by listing the main SGL advantages for dealing with distributed dynamic systems and its ability to effectively cover areas of other languages, even as a natural language itself for quick high-level grasping of complex spatial problems.

2. Spatial Grasp Technology and Language

2.1. The basic idea of SGT

Within Spatial Grasp Technology (SGT), a high-level scenario for any task to be performed in a distributed world is represented as an active self-evolving pattern rather than a traditional program, sequential or parallel [1–11]. This pattern, written in a high-level Spatial Grasp Language (SGL) and expressing the top semantics of the problem to be solved, can start from any point of the world. Then it spatially propagates, replicates, modifies, covers, and matches the distributed world in a parallel wave-like mode while echoing the reached control states and data found or obtained for making decisions at higher levels and further space navigation, as symbolically shown in Fig. 1.

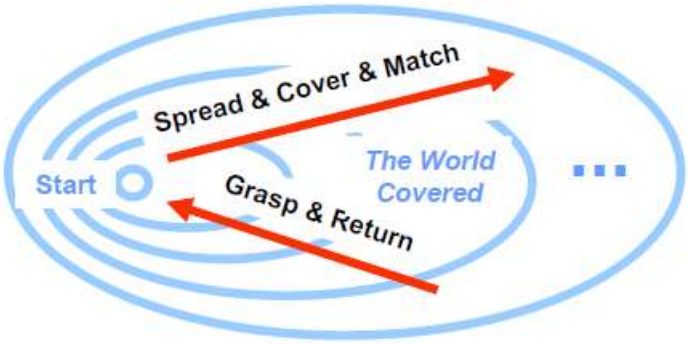


Figure 1 – The main idea of the Spatial Grasp Model

2.2. The worlds SGT operates with

SGT allows us to directly operate with the following world representations (more details in [10]): Physical World (PW) considered as continuous and infinite, with different points accessed by coordinates; discrete Virtual World (VW) consisting of nodes and semantic links between them; and Executive world (EW) consisting of active "doers" with can communiucate. Different kinds of combinations of these worlds can also be possible within the same formalism, as follows: Virtual-Physical World (VPW), Virtual-Execution World (VEW), Execution-Physical World (EPW), and Virtual-Execution-Physical World (VEPW) combining all features of the previous cases.

2.3. SGL global recursive structure

The language top-level universal and recursive organization can be expressed just by a single string in a formula-like mode, as follows:

grasp → *constant* | *variable* | *rule* ({ *grasp*, })

From this definition, an SGL scenario, called grasp, supposedly applied at some point of the distributed space, can just be a constant. It can be a variable whose content, assigned to it previously, provides the result in the application point too. It can also be a rule (expressing certain action, control, description, or context) optionally accompanied with operands and embraced in parentheses. These operands can be of any complexity and defined recursively as grasp, too. The main SGL elements can be summarized as follows, where a constant and a rule can also be represented as composed elements, defined recursively as grasp again:

constant → *information* | *matter* | *custom* | *special* | *grasp*
variable → *global* | *heritable* | *frontal* | *nodal* | *environmental*
rule → *type* | *usage* | *seeing* | *movement* | *creation* | *echoing* | *verification* | *assignment* | *advancement* | *branching* | *transference* | *exchange* | *timing* | *qualifying* | *extension* | *grasp*

2.4. How SGL scenarios evolve

Below there are provided some details on how SGL scenarios self-evolve in distributed environments.

1. SGL scenario develops in steps, potentially parallel, with new steps produced on the results of previous ones.

2. Any step is always associated with a certain point or points of the world (physical, virtual, executive, or combined) from which the scenario is currently developing.

3. Each step provides a resultant value (single or multiple) representing information, matter, or both, and a resultant control state.

4. Different scenario parts may evolve from the same points in an ordered, unordered, or parallel manner, as independent or interdependent branches.

5. Different parts of the scenario can spatially succeed each other, with new parts evolving from final positions reached by the previous ones.

6. This potentially parallel and distributed scenario may proceed in synchronous or asynchronous modes. Any of their combinations are also possible.

7. SGL operations and decisions can use control states and values returned from other, subsequent, scenario parts, combining forward and backward scenario evolution.

8. Different steps from the same or different scenarios may be temporarily associated with the same world points, sharing persistent or provisional information in them.

9. Staying with different world points, whether physical or virtual, it may be possible to change local parameters in them, thus impacting the worlds via these locations.

10. Scenarios navigating distributed spaces can create active distributed physical or virtual infrastructures in them operating on their own. They can also be shared with other scenarios.

11. Overall organization of the world creation, coverage, modification, analysis, and processing can be provided by a variety of SGL rules which may be arbitrarily nested.

12. The evolving SGL scenarios can lose the utilized parts if not needed anymore, also self-modify and self-replicate during space navigation, automatically adjusting to the distributed environments.

2.5. SGT control states

The following control states appear after the completion of different scenario steps, indicating their progress or failure. They can be used for effective control of multiple distributed processes with proper decisions at different levels. These control states are the following:

– thru reflects the full success of the current branch with the capability of further scenario development;

– done indicates the success of the current scenario step with its planned termination;

– fail indicates non-revocable failure of the current branch and no possibility of further development from the location reached;

– fatal reports of terminal failure in the location reached while triggering massive abortion of the currently evolving scenario processes, as well as in other branches, with deletion of associated temporary data in them. The globality of such removal process can be controlled by special higher-level rules.

3. SGL details

A full explanation of all SGL constructs can be found in [4–8]. Below there are provided only their brief descriptions.

3.1. SGL constants

The main constant types are as follows:

constant → *information* | *matter* | *special* | *custom* | *grasp*

Constants are self-identifiable by the way they are written. They can also be explicitly defined by special embracing rules for arbitrary textual representations.

– Information – as arbitrary strings in single quotes (which may be nested, if multiple) and numbers in traditional representations;

– Physical matter or physical objects – as any texts in double quotes (which can be nested, too) adequately identifying these elements;

– Special or reserved *constants* which may be used as standard parameters or modifiers in different language rules:

special → thru | done | fail | fatal | infinite | nil | any | all | other | allother | current | passed | existing | neighbors | direct | forward | backward | neutral | synchronous | asynchronous | virtual | physical | executive | engaged | vacant | firstcome | unique

– Custom – Special self-identifiable names/words established additionally by customers to serve specific classes of problems (to be adjusted with SGL implementations);

– Grasp – Constants can also be compound ones, using the recursive *grasp* definition in SGL syntax, which allows for nested hierarchical structures consisting of multiple (elementary or compound again) objects.

3.2. SGL variables

Their repertoire is as follows, all being self-identifiable or declared by special rules for any names:

variable → *global* | *heritable* | *frontal* | *nodal* | *environmental*

Global Variables

This is the most expensive type of SGL variables with their names starting with capital `G` and followed by arbitrary sequences of alphabetic letters and/or digits:

global → `G`{*alphameric*}

These variables can exist only in single copies with particular names, being common for both read and write operations to all processes of the same scenario, regardless of their physical or virtual distribution and world points they may cover.

Heritable Variables

The names of these variables should start with capital `H` if not defined by a special rule:

heritable → H{*alphameric*}

Heritable variables, being created by the first assignment to them at some scenario development stage, are becoming common for read-write operations for all subsequent scenario operations (generally multiple, parallel and distributed) evolving from this particular point and wherever is space they happen to be.

Frontal Variables

These are mobile-type variables with names starting with capital F, which are propagating in distributed spaces while keeping their contents at the forefront of evolving scenarios:

frontal → F{*alphameric*}

Each of these variables is serving only the current scenario branch operating in the current world point. They cannot be shared with other branches evolving in the same or other world points, always accompanying the scenario control.

Nodal Variables

Variables of this type (their identifiers starting with capital N) are a temporary and exclusive property of the world points visited by SGL scenarios, not of the processes covering these nodes, which can, however, create, change or even remove them.

nodal → N{*alphameric*}

Capable of being shared by all scenario branches, they are created by the first assignment to them and stay in the node until removed explicitly. These nodes can be visited by different scenarios.

Environmental Variables

These are special variables with reserved names which allow us to have access to physical, virtual and execution worlds when they are navigated by SGL scenarios, also to some important internal parameters of the language interpretation system itself.

environmental → TYPE | NAME | CONTENT | ADDRESS | QUALITIES | WHERE | BACK | PREVIOUS | PREDECESSOR | DOER | RESOURCES | LINK | DIRECTION | WHEN | TIME | STATE | VALUE | IDENTITY | IN | OUT | STATUS

3.3. SGL rules

Rules can organize navigation of the world sequentially, in parallel, or in any combinations. They can result in the same application point or cause movement to other world points with obtained results. The reached points can become starting points for other rules. The rules, due to recursive language organization, can form arbitrary operational and control infrastructures expressing any sequential, parallel, hierarchical, centralized, localized, mixed and up to fully decentralized and distributed algorithms. The concept of the rule is dominant in SGL not only for diverse activities on data, knowledge, and physical matter but also for overall management and control of any SGL scenarios. This provides an integral and unified capability for expressing everything that might take place or even come to mind in large dynamic spaces, worlds, and systems, and generally in holistic, highly parallel, and fully distributed mode.

Type

These rules explicitly assign types to different constructs and variables, with their existing repertoire following them.

type → global | heritable | frontal | nodal | environmental | matter | number | string | scenario | constant

Usage

These rules explain how to use the information units they embrace with their main variants:

usage → `address|coordinate|content|index|time|speed|name|center|range|doer|node|link`

Seeing

These rules allow us to see and observe spaces of interest by given parameters:

seeing → `observe|detail|collect`

The observation can just confirm the presence or absence of certain features or objects, or additionally collect and return positions of the areas or objects found, like physical coordinates or virtual addresses, to be processed by other rules. Such observation may use the built-in capabilities of the SGL interpreters supplied with proper sensors or radars. For more complex cases, proper external systems may be accessed (see *transference* rules).

Movement

The movement rules have the following options:

movement → `hop|hopfirst|hopforth|move|shift|follow`

They result in virtual hopping to the existing nodes or real movement to new physical locations, subsequently starting the remaining scenario, if any (with current frontal variables and control) of the points is reached.

Creation

These rules have the following options:

creation → `create|linkup|delete|unlink`

They create or remove nodes and/or links leading to them during distributed world navigation. After the rule termination, their resultant values will correspond to the names of the reached or starting nodes, depending on the result, with termination states `thru` in them.

Echoing

This class of rules implements different variants of data and knowledge processing, some related to physical matter, too. The rules may use both local and remote values for operations.

echoing → `state | rake | order | unit | sum | count | first | last | min | max | random | average | element | sortup | sortdown | reverse | fromto | add | subtract | multiply | divide | degree | separate | unite | attach | append | common|withdraw|increment|decrement|access|invert|apply|location|distance`

The listed rules use world positions reached by the embraced scenario with their associated values, with the resultant position and value in their starting points, with state `thru` or `fail` depending on the success of the operation.

Verification

This class of rules has the following main variants:

verification → `equal | nonequal | less | lessorequal | more | moreorequal | bigger | smaller | heavier | lighter | longer | shorter | empty | nonempty | belong|notbelong|intersect|notintersect|yes|no`

As a result, these rules provide a control state `thru` or `fail` reflecting the outcome of the concrete verification procedure, also `nil` as its own resultant value, while remaining after the completion in the same position from which it started.

Assignment

There are two rules of this class:

assignment → `assign|assignpeers`

These rules assign the result of the right scenario operand (that may be arbitrarily remote and also represent a list of values that can be nested) to the variable or set of variables directly named or reached by the left scenario operand, which may be remote too.

Advancement

This class of rules has the following variants:

advancement → `advance|slide|repeat|align|fringe`

These rules can organize forward or "in-depth" advancement in space and time of the embraced scenarios separated by a comma. They can evolve within their sequence in a synchronous or asynchronous mode by using `synchronous` or `asynchronous` modifiers (the second one is optional as asynchronous is a default mode).

Branching

The rules from this class are as follows:

branching → `branch | sequence | parallel | if | or | and | choose | quickest | cycle|loop|sling|whirl|split|or_sequence|or_parallel|replicate`

These rules allow the embraced set of scenario operands to develop "in breadth", each from the same starting position, with the resultant set of positions and order of their appearance depending on the logic of a concrete rule. The rest of the SGL scenario will be developing from all or some of the positions and nodes reached on the rule.

Transference

There are two rules of this class:

transference → `run|call`

They organize transference of control in distributed scenarios. This may be to an SGL code as a procedure that may be local or the result of the invocation of the embraced scenario. They can also activate external systems by transferring control into them.

Exchange

They provide the exchange of information or physical matter (objects) with the external world on the initiative of the SGL scenario, including broadcasting over the whole distributed area.

exchange → `input|output|send|receive|emit|get`

Timing

The following options establish conditions for further scenario evolution from the current point, with proper time limits for scenario suspension or its further activation.

timing → `sleep|allowed`

Qualification

This class contains the following rules which provide certain qualities or abilities and set proper constraints or restrictions on the scenarios they embrace.

qualification → `contain|release|trackless|free|blind|quit|abort|stay|lift|seize|exit`

Extension

Any new types of rules can be incorporated into the language and its interpreter, being oriented on specific classes of problems but obeying the general language culture and its organization.

Grasping

The rule identifier can also be provided as the result produced by a scenario of any complexity, the latter standing in its place. It can also be a compound one, integrated from multiple names provided by different embraced scenarios.

4. SGL spatial interpretation

4.1. Organization issues of the SGL interpreter

An SGL interpreter (the details can be found in [2–8]) consists of a number of specialized functional processors working with specific data structures. These include Communication Processor (CP), Control Processor (CoP), Navigation Processor (NP), Parser (P), different Operation Processors (OP), and a special World Access Unit (WU). The main data structures comprise Grasps Queue (GQ), Suspended Grasps (SG), Track Forest (TF), Activated Rules (AR), Knowledge Network (KN), Grasps Identities (GI), Global Variables (GV), Heritable Variables (HV), Fontal Variables (FV), Nodal Variables (NV), Environmental Variables (EV), Incoming Queue (IQ), and Outgoing Queue (OQ). Each interpreter can support and process multiple SGL scenario codes which happen to be its responsibility at different moments of time (using CoP, NP, P, OP, WU, GQ, SG, TF, AR, KN, GI, GV, HV, FV, NV, EV, IQ, OQ) and also exchange scenario code and data with other SGL interpreters (using CP, CoP, NP, WU, TF, KN, HV, FV, EV, IQ, OQ).

Communicating interpreters of SGL can be in any number of copies and effectively integrate with existing systems and communications. They can simultaneously execute many tasks-scenarios without central control. The hardware of software SGL interpreters (including those created and replicated at runtime), as universal computational, control and management units U (see Fig. 2), can be stationary or mobile and installed in proper physical or virtual world points on the request of the SGL scenarios.

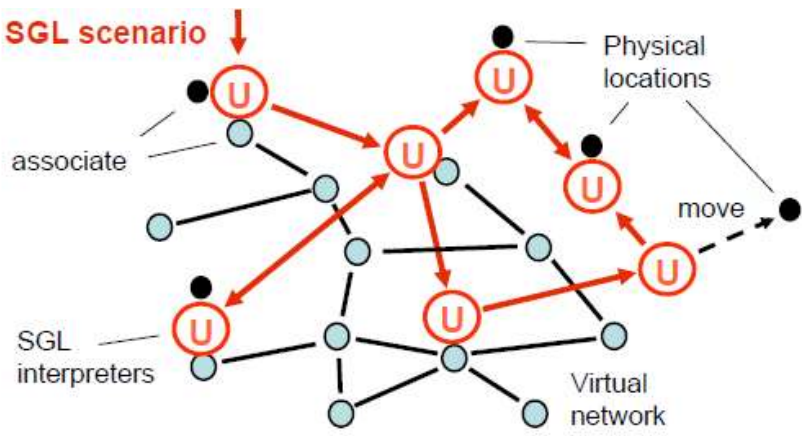


Figure 2 – SGL distributed interpretation in physical and virtual environments

4.2. Spatial tracking system

As both the backbone and nerve system of the distributed interpreter, its self-optimizing spatial track system [4–8] provides hierarchical command and control as well as remote data and code access. It also supports spatial variables and merges distributed control states for decisions at higher organizational levels. The track infrastructure is automatically distributed between active components (humans, robots, computers, smartphones, satellites, etc.) during scenario self-spreading in distributed environments. It integrates the following stages of operation.

– Forward grasping. In the forward process, the next steps of scenario development form new track nodes connected to the previous ones by the track links. Reflecting the history of scenario evolution, this growing track structure effectively supports heritable, nodal and frontal variables associated with the track nodes.

– Echoing. After completing the forward stage of the SGL scenario, the track system can return to the starting track node the generalized control state based on termination states in all fringe nodes, also marking the passed track links with the states returned via them. The track system, on the request of the higher-level scenario rules, can also collect local data obtained at its fringe nodes and merge them into a resultant list of values echoed to the starting node. The track echoing process also optimizes the track system by deleting already used and not needed any more items.

– Further forward development. The echo-modified and optimized track system can route further grasps to the world positions reached by the previous grasps. Heritable variables created in certain track nodes can be also accessed from the subsequent nodes in the track system for both reading and writing operations.

5. Comparison with other programming languages

A programming language [12–13] is any set of rules that converts strings or graphical program elements in the case of visual programming languages to various kinds of machine code output. Thousands of different programming languages have been created, and more are being created every year. Many programming languages are written in an imperative form (i.e. as a sequence of operations to perform) while other languages use the declarative form (i.e. the desired result is specified, not how to achieve it). The description of a programming language is usually split into two components of syntax (form) and semantics (meaning), which are usually defined by a formal language. Most popular programming languages include Python, JavaScript, Java, C#, C, C++, Go, R, Swift, and PHP, with some others (Dart, Kotlin, MATLAB, Perl, Ruby, Rust, and Scala) currently used as well.

A particular class of programming languages is oriented on distributed computing systems [14–15], with the following three main approaches to the distributed languages: distributed shared memory, actor/object model, and dataflow model. Other orientation is on concurrent and parallel programming languages [16], covering coordination languages, dataflow programming, distributed computing, event-driven and hardware description, functional programming, logic programming, monitor-based, multi-threaded, object-oriented programming, partitioned global address space (PGAS), and message passing.

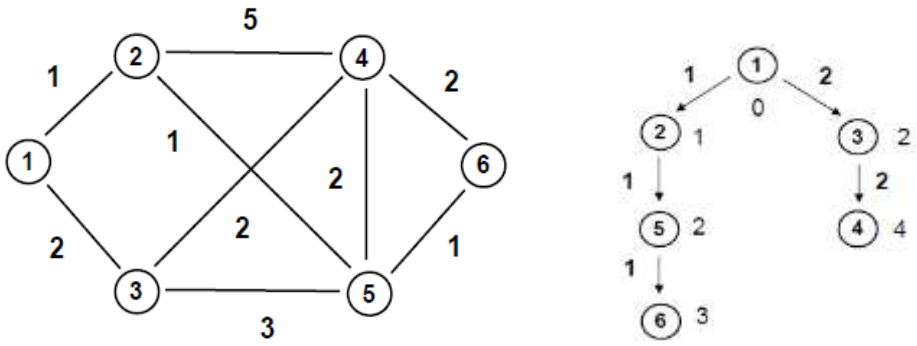
There are also languages oriented to mobile agents and mobile computing [17–19]. In mobile computing, mobile agents are the composition of computer software and data that can autonomously move from one computer to another and continue its execution on the destination computer. In other words, a mobile agent is an autonomous program capable of moving from host to host in a network and interacting with resources and other agents. In this process, the chance of data loss is scarce because the state of the running program is saved and then transported to the new host. It allows the program to continue execution from where it set off before migration. The most significant advantage of mobile agents is the possibility of moving complex processing

functions to a location with enormous amounts of data that have to be processed. Mobile agents can travel from computer to computer in a network, at times and to places they choose by themselves. The state of the running program is saved since it is transmitted to the destination. The program is resumed at the destination continuing its processing with the saved state. Mobile agents have several advantages in the development of various services in smart environments in addition to distributed applications. These are reduced communication costs, asynchronous execution, direct manipulation, dynamic deployment of software, and easy development of distributed applications.

SGL covers almost all functionality of the mentioned languages, combining their different features in a unified and integral way. Let us consider some examples.

An example of shortest paths finding

Let us consider a well-known problem of finding the shortest paths in a network, where a network with weighted links is shown in Fig. 3 *a*, and its shortest-path tree – in Fig. 3 *b*.



a *b*

Figure 3 – Finding the shortest path tree in a network

The most famous method for solving this task is Dijkstra's Shortest Path Algorithm, for which a solution in C++ (taken from [20]) is copied in Fig. 4 (which is oriented to a single computer, being strictly sequential, and with a matrix-based representation of the network).

But the creation of the shortest path tree from a node in a network, covering it completely, can be expressed in SGL as follows (being equivalent to the solution provided in Fig. 4). Its enormous simplicity and compactness are due to the used method that is fully distributed and parallel, where an active SGL code self-spreads through the body of a distributed network, automatically bringing into its nodes the growing shortest distance from the start (including redefining this distance in nodes, and a better solution appears possible after reaching these nodes again). So the SGL solution integrates the mentioned above features of other languages as parallel and distributed computing, as well as mobile code or agents, with everything appearing within the same universal and recursive language syntax and semantics.

```
frontal(Far); nodal(Distance = 0, Before); hop(a);

repeat(hop_links(all); Far += LINK;
       or(Distance == nil, Distance > Far);
       Distance = Far; Before = PREVIOUS)
```

For further development of this solution with the use of SGL capabilities, after finding and fixing the shortest path tree in the distributed network structure, we may add collecting and outputting the shortest path from starting node `a` to any other node `e` as follows (with printing the result in the starting node `a`):

```
frontal(Far); nodal(Distance = 0, Before); hop(a);
```

```
sequence(
  repeat(hop_links(+all); Far += LINK;
         or(Distance == nil, Distance > Far);
         Distance = Far; Before = PREVIOUS),
  (hop(e); repeat(append(Path, NANE); hop(Before));
   output(Path)))
```

Different variants for finding and fixing the shortest paths in distributed networks using SGL and its predecessor variants can be found in [4–7]. Any other path examples in a distributed network can also be expressed in SGL in a very simple and compact way.

```
#include<iostream>
#include<climits>
using namespace std;

int miniDist(int distance[], bool Tset[]) // finding minimum distance
{
    int minimum=INT_MAX,ind;

    for(int k=0;k<6;k++)
    {
        if(Tset[k]==false && distance[k]<=minimum)
        {
            minimum=distance[k];
            ind=k;
        }
    }
    return ind;
}

void DijkstraAlgo(int graph[6][6],int src) // adjacency matrix
{
    int distance[6]; // // array to calculate the minimum distance for each node
    bool Tset[6];// boolean array to mark visited and unvisited for each node


    for(int k = 0; k<6; k++)
    {
        distance[k] = INT_MAX;
        Tset[k] = false;
    }

    distance[src] = 0;   // Source vertex distance is set 0

    for(int k = 0; k<6; k++)
    {
        int m=miniDist(distance,Tset);
        Tset[m]=true;
        for(int k = 0; k<6; k++)
        {
            // updating the distance of neighbouring vertex
            if(!Tset[k] && graph[m][k] && distance[m]!=INT_MAX && distance[m]+graph[m][k]<distance[k])
                distance[k]=distance[m]+graph[m][k];
        }
    }
    cout<<"Vertex\t\tDistance from source vertex"<<endl;
    for(int k = 0; k<6; k++)
    {
        char str=65+k;
        cout<<str<<"\t\t\t"<<distance[k]<<endl;
    }
}

int main()
{
    int graph[6][6]={
        {0, 1, 2, 0, 0, 0},
        {1, 0, 0, 5, 1, 0},
        {2, 0, 0, 2, 3, 0},
        {0, 5, 2, 0, 2, 2},
        {0, 1, 3, 2, 0, 1},
        {0, 0, 0, 2, 1, 0}};
    DijkstraAlgo(graph,0);
    return 0;
}
```

Figure 4 – C++ code for Dijkstra's Shortest Path Algorithm

All paths between network nodes

Another example of simplicity and compactness of SGL solutions may be finding and outputting all simple paths from a node `a` to another node `f`, which can be written as follows (with more on different paths in [4–7]):
a) with printing all found simple paths in the starting node `a`:

```
frontal(Path); hop(a);
output(repeat(notbelong(NAME, PATH); append(NAME, PATH);
              if(equal(NAME, f), blind(Path), hop(neighbors))))
```

b) with printing all found simple paths in the final node `f`:

```
frontal(Path); hop(a);
repeat(notbelong(NAME, PATH); append(NAME, PATH);
       if(equal(NAME, f), done(output(Path)), hop(neighbors)))
```

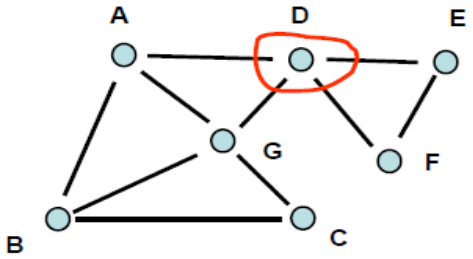


Figure 5 – An articulation point in a network

An example of articulation points

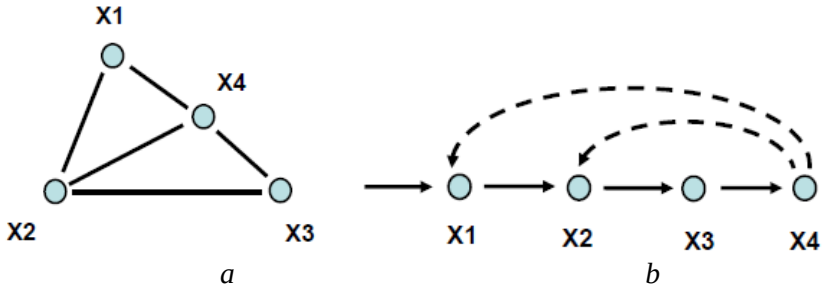
Parallel, distributed and mobile SGL nature allows us to describe solutions to many other network-related problems in an extremely compact and straightforward way, incomparable with any other programming languages mentioned above. For example, to find all the articulation points in a distributed network (i.e. the nodes which, when deleted, split the network into disjoint parts like node D in Fig. 5), the following SGL scenario originally applied in all the network nodes will be sufficient:

```
hop_first(nodes(all)); COLOR = NAME;
and_sequence(
   (hop_first(any(link)); repeat(hop_first(all(links))),
   hop_first(all(links)),
   output(NAME))
```

The result will be D. The details of this and other related SGL scenarios for finding articulation points in a network can be found in [4–7].

Finding graph patterns in a distributed network

Other solutions in distributed networks may relate to finding in them certain graph structures by their spatial pattern-matching with the network topology. An example of a network pattern is shown in Fig. 6 *a* and its spatial matching template – in Fig. 6 *b*.



a *b*

Figure 6 – Graph pattern and its network matching template

The following SGL scenario applying the self-evolving matching template of Fig. 6 *b* to some distributed network, starting originally in all the network nodes, may be as follows:

```
hop(all); frontal(X) = NAME;
repeat(3)(
  (hop(neighbors); notbelong(NAME, X); append(X, NAME));
  yes(hop_and(X[1,2]); output_unit(X)
```

By applying this scenario to the network presented in Fig. 5, we will obtain the following results for variables X1 to X4:

(A, B, C, G), (B, G, D, A), (C, G, A, B), (D, G, B, A), ...

Duplicates of the same found structures are explained by the fact that Xi can match different nodes of the same structure due to its symmetrical shape, as of Fig. 6a. More details on solving graph-pattern matching problems in SGL can be found in [4–7] which confirm the clarity, simplicity, and compactness of the solutions found in comparison with any other existing programming languages.

The following note concerns the SGL code in the above and subsequent examples. Sometimes, SGL allows us to use traditional styles in expressions similar to other languages if this simplifies the code but always remains within the global recursive syntactic formula of subsection 2.3. For example, the sequential application of constructs Ci by rule `advance`, i.e. one after the other, can be substituted by just using a semicolon as a separator between them, the rule explicit mentioning omitted:

`advance(C1, C3, C3)` → C1; C2; C3

6. Comparison of SGL with Battle Management Language

In the increasingly complex worlds of battlefield, anti-terrorism, peace-keeping, and disaster relief operations there is a clear need for rapid, useful, precise and unambiguous exchange of information between military units within a given theater of activity [21]. Orders, requests, and reports have to be expressed in a formal language sufficiently standardized to be unambiguous and sufficiently expressive to convey a commander's intent or to report enemy or civilian activities. This language must also support network-centric communication requirements by facilitating automatic processing and dissemination of information. If military communication can be processed automatically, it can be exchanged not only among the forces and their C2 systems but also between commanders, C2 systems, and simulation systems. Then commanders can directly command simulated forces, and the command Intent is currently incorporated in a number of decision models. Battle Management Language (BML) [22–23] is being developed as an open standard that unambiguously communicates command and control information, including orders that express command intent. Its version is Geospatial Battle Management Language (GeoBML) for terrain reasoning [24]. Terrain and weather effects represent a fundamental, enabling piece of battlefield information, supporting situation awareness and the decision-making processes within the domain of C2. These effects can both enhance or constrain force tactics and behaviors, platform performance (ground and air), system performance (e.g. sensors), and the soldier. Typical command and control order in BML [22] looks the following way:

OrderParagraph3 → CI OB* C_Sp* C_T*.

In this production rule, the task expression consists of the command intent (CI), the basic order expression outlining specific tasks (OB), spatial coordination expression (C_Sp), and temporal coordination expression (C_T). The asterisk indicates that arbitrarily many of the respective expressions can be concatenated. The basic order expression may look the following way:

OB → Verb Tasker Taskee (Affected|Action) Where Start-When (End-When) Why Label (Mod)*
With concrete examples like:
OB → advance Tasker Taskee Route-Where Start-When (End-When) Why Label (Mod)*
OB → defend Tasker Taskee Affected At-Where Start-When (End-When) Why Label (Mod)*

SGL naturally operating with the distributed systems can be effectively used as a high-level language for expressing any battlefield operations, which may include moving through physical spaces with complex terrain, carrying the needed ammunition, troops, and supplies, and making proper local and global decisions on the campaign development and management. The following example in SGL describes a synchronized movement from the Start point and then through the given Pi positions in physical space with feedback control, with needed resources and tasks to be performed in the reached positions, also taking into account the planned campaign start and end time (see Fig. 7).

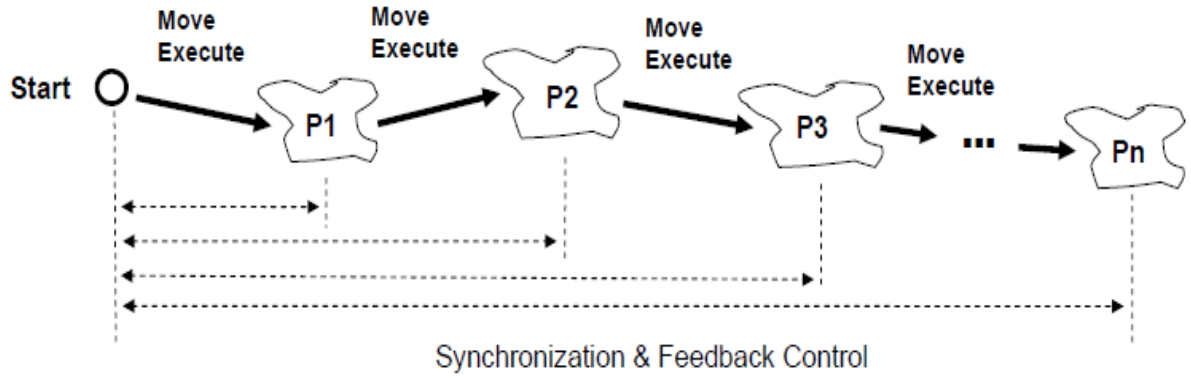


Figure 7 – Distributed campaign management scenario

```
frontal(Start = ..., Route = (P1, P2, P3, ..., Pn),
        Resources = (fighters, ammunition, food, ...),
        Tasks = (seize, destroy, defend, restore, ...),
        Start_When = ..., End_When = ...);
move(Start); wait_for(TIME == Start_When));
if(
  repeat_synchronous(
     TIME < End_When; move(withdraw(Route));
     execute(Tasks, Resources)),
  output("Mission successful"),
  output("Mission failed"))
```

The campaign resources can be gradually changed, especially diminished, after passing each stage and operating at it, which, through the use of frontal variables carrying these resources, can be properly taken into account. The frontal variable carrying the set of tasks to be performed can also reflect their possibly changing repertoire after performing operations in the passed positions. SGL can effectively describe any military campaign, national or international, on any territory, and with making decisions at any level. These high-level descriptions can be collectively executed by any manned and unmanned equipment. A detailed comparison of the traditional battle management scenario in BML and its implementation in SGL can be also found in [4].

7. Relation of SGL to natural languages

For any language, its syntax and structure usually go hand in hand, where a set of specific rules, conventions, and principles govern the way words are combined into phrases, phrases – into clauses, and clauses – into sentences [25]. Knowledge about the language structure and syntax is helpful in many areas like text processing, annotation, and parsing for further operations such as text classification or summarization. Language analysis is carried out at different levels, starting from single words and then covering structures from different words which may be hierarchical, i.e. may include other structures. Usually, words can fall into one of the following major categories: Nouns that depict some object or entity*;* Verbs are words used to describe certain actions, states, or occurrences*;* Adjectives are words used to describe or qualify other words; Adverbs *t*hat usually act as modifiers for other words including nouns, adjectives, verbs, or other adverbs. Besides these four major categories of parts of speech, there are other ones (pronouns, prepositions, interjections, conjunctions, determiners, and many others) that occur frequently in the English language.

Groups of words are combined into phrases, and there are five major categories of phrases: Noun phrase where a noun acts as the headword, Verb phrase with a verb acting as the headword, Adjective phrase with an adjective as the headword, Adverb phrase which acts like adverbs since the adverb acts as the headword in the phrase, and Prepositional phrase which usually contains a preposition as the headword and other lexical components like nouns, pronouns, etc. Natural Language Processing (NLP) [26–28] is the process of determining the syntactic structure of a text by analyzing its constituent words, the way for computers to analyze, understand, and derive meaning from the human language in a smart and useful way. By utilizing NLP, developers can organize and structure knowledge to perform tasks such as automatic summarization, translation, named entity recognition, relationship extraction, sentiment analysis, speech recognition, and topic segmentation.

SGL has a very simple but universal syntax (see subsection 2.3) which allows us to directly describe any natural language structure just by adding new rules within the same recursive SGL organization. Such rules with their abbreviations, identifying the subject, separate words, and categories of phrases, may be as follows: Subject (S), Noun (N), Verb (V), Determiner (DT), Preposition (P), Noun Phrase (NP), Verb Phrase (VP), and Preposition Phrase (PP).

subject → *word* | *rule* ({***subject***,})
rule → S | N | V | DT | P | NP | VP | PP

For example, the natural language sentence “Peter saw a car in the road” can be structured as follows (see Fig. 8):

S(NP(N(**Peter**)), VP(V(**saw**), NP(DT(**a**), N(**car**)), PP(P(**in**), NP(DT(**the**), N(**road**)))))

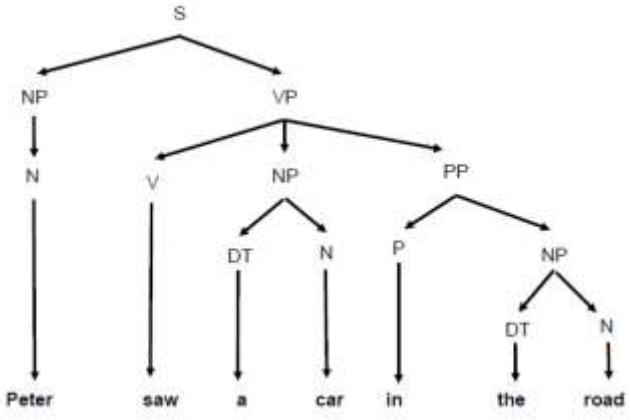


Figure 8 – Example of a syntactic structure of a natural language sentence

Using the same SGL syntax with new rules added, another natural language sentence “A man behind the window saw a car in the road” as an extension of the previous one, can be easily structured, too (see Fig. 9).

S(NP(NP(DET(**a**), N(**man**)), PP(P(**behind**), DT(**the**), N(**window**))), VP(V(**saw**), NP(DT(**a**), N(**car**)), PP(P(**in**), NP(DT(**the**), N(**road**)))))

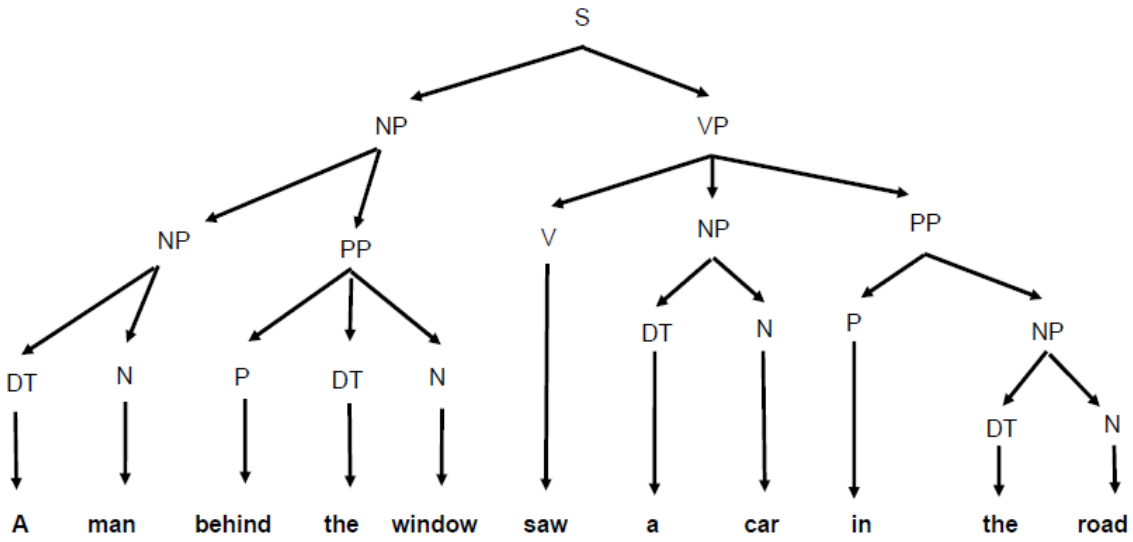


Figure 9 – Extended structure of a natural language sentence

The latest example can also be written without commas if spaces are always provided between different elements:

S(NP(NP(DET(**a**) N(**man**)) PP(P(**behind**) DT(**the**) N(**window**))) VP(V(**saw**) NP(DT(**a**) N(**car**)) PP(P(**in**) NP(DT(**the**) N(**road**)))))

We have considered the use of the SGL syntax only for describing natural language syntactical structures, which may be arbitrary complex and without limits due to the recursive SGL organization. It may allow us to describe any imaginable and even so far unimaginable language structures, which may not always be easily comprehended by the human brain due to their complexity and areas of coverage, but be rather analyzed and executed by special AI systems. SGL parallel, distributed and mobile philosophy may also influence the preparation of natural language texts in the way they can be partitioned and used by teams of readers in a parallel and distributed way, also with parts associated with certain world resources and locations transferred directly to these locations, analyzed and executed there. Such possibilities of using natural language texts may be worth further investigation, but the use of specialized programming languages, including SGL, can always be more practical for specific applications.

8. Conclusions

These are the main results obtained and discussed throughout the current paper:

– Theoretical and technological details of the recursive Spatial Grasp Language have been summarized and discussed together with the mechanisms of its parallel and distributed implementation, which allow us to obtain powerful spatial machines operating without centralized resources in any terrestrial and celestial environments.

– SGL has been compared with other programming languages with concrete programming examples showing high level, simplicity, and compactness of the obtained solutions, which are explained by the fact that SGL operates directly on distributed networked bodies in a holistic,

parallel, self-navigation and pattern matching mode. This effectively puts upside down the established opinion that parallel and distributed computing is essentially more complex than sequential programming.

– SGL has been compared with specialized battle management languages oriented to command and control of real military campaigns, which can be executed by manned and unmanned components. An SGL programming example showed clarity and efficiency of expressing large campaigns with spatial movement and maintenance of the needed resources, also overall management and control, which can confirm universal ability of SGL for extended applications in this one and similar areas.

– The relation of SGL to natural languages has also been shown where globally recursive SGL organization can be used for describing and analyzing any natural language structures, imaginable and even so far unimaginable, and of any volume and complexity, just by adding new rules within the SGL definition. SGL can also be often used instead of natural language for a high-level and quick definition of complex spatial problems and their solutions due to its compactness and formula-like nature.

REFERENCES

1. Sapaty P.S. A distributed processing system, European Patent N 0389655. Publ. 10.11.93. European Patent Office. 35 p.
2. Sapaty P.S. Mobile Processing in Distributed and Open Environments. New York: John Wiley & Sons, 1999. 410 p.
3. Sapaty P.S. Ruling Distributed Dynamic Worlds. New York: John Wiley & Sons, 2005. 255 p.
4. Sapaty P.S. Managing Distributed Dynamic Systems with Spatial Grasp Technology. Springer, 2017. 284 p.
5. Sapaty P.S. Holistic Analysis and Management of Distributed Social Systems. Springer. 2018. 234 p.
6. Sapaty P.S. Complexity in International Security: A Holistic Spatial Approach. Emerald Publishing. 2019. 160 p.
7. Sapaty P.S. Symbiosis of Real and Simulated Worlds under Spatial Grasp Technology. Springer, 2021. 251 p.
8. Sapaty P.S. Spatial Grasp as a Model for Space-based Control and Management Systems. CRC Press, 2022. 280 p.
9. Sapaty P.S. Comprehending Distributed Worlds with the Spatial Grasp Paradigm. *Mathematical machines and systems*. 2022. N 1. P. 12–30. URL: http://www.immsp.kiev.ua/publications/articles/2022/2022_1/01_22_Sapaty.pdf.
10. Sapaty P.S. Holistic spatial analysis of distributed worlds. *Mathematical machines and systems*. 2022. N 2. P. 3–12. URL: http://www.immsp.kiev.ua/publications/articles/2022/2022_2/02_22_Sapaty.pdf.
11. Sapaty P.S. A Brief Introduction to the Spatial Grasp Language (SGL). *Journal of Computer Science & Systems Biology*. 2015. N 09 (02). URL: https://www.researchgate.net/publication/307744279_A_Brief_Introduction_to_the_Spatial_Grasp_Language_SGL.
12. Programming language. URL: https://en.wikipedia.org/wiki/Programming_language.
13. Eastwood B. The 10 Most Popular Programming Languages to Learn in 2022. 2022. June 18. URL: https://www.northeastern.edu/graduate/blog/most-popular-programming-languages/.
14. Bal H.E., Steiner J.G., Tanenbaum A.S. Programming Languages for Distributed Computing Systems. *ACM Computing Surveys*. 1989. Vol. 21, N 3. URL: https://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.145.7873&rep=rep1&type=pdf.
15. Zanin C. Distributed Programming Languages, Programming Models for Distributed Computing. URL: http://dist-prog-book.com/chapter/4/dist-langs.html.
16. List of concurrent and parallel programming languages. URL: https://en.wikipedia.org/wiki/List_of_concurrent_and_parallel_programming_languages.
17. Mobile Agents in Mobile Computing. URL: https://www.javatpoint.com/mobile-agents-in-mobile-computing.
18. Satoh I. Mobile Agents. URL: https://research.nii.ac.jp/~ichiro/papers/satoh-mobile-agent.pdf.

19. Cao J., Das S.K. Mobile Agents in Networking and Distributed Computing (Wiley Series in Agent Technology Book 3), Wiley-Interscience, 2012. July 11.
20. Dijkstra's Algorithm in C++ | Shortest Path Algorithm. 2021. May 20. URL: https://favtutor.com/blogs/dijkstras-algorithm-cpp.
21. Hieb M.R., Schade U. Formalizing Command Intent Through Development of a Command and Control Grammar. 2007. June. URL: https://www.researchgate.net/publication/228817220_Formalizing_Command_Intent_Through_Development_of_a_Command_and_Control_Grammar.
22. Kruger K., Schade U. Battle Management Language: Military Communication with Simulated Forces. 2007. October. URL: https://www.researchgate.net/publication/253246607_Battle_Management_Language_Military_Communication_with_Simulated_Forces.
23. Schade U., Hieb M.R. Development of Formal Grammars to Support Coalition Command and Control: A Battle Management Language for Orders, Requests and Reports. 2006. September. URL: https://www.researchgate.net/publication/237778366_Development_of_Formal_Grammars_to_Support_Coalition_Command_and_Control_A_Battle_Management_Language_for_Orders_Requests_and_Reports.
24. Hieb M., Pullen M., Kleiner M. A Geospatial Battle Management Language (GeoBML) for Terrain Reasoning, Presented to the 11th International Command and Control Research and Technology Symposium. URL: http://www.dodccrp.org/events/11th_ICCRTS/html/presentations/110.pdf.
25. Sarkar D. Understanding Language Syntax and Structure: A Practitioner's Guide to NLP. 2018. August 10. URL: https://www.kdnuggets.com/2018/08/understanding-language-syntax-and-structure-practitioners-guide-nlp-3.html.
26. Rakesh H. Natural Language Processing, What is Natural Language Processing? 2018. November 25. URL: https://becominghuman.ai/natural-language-processing-in-a-nutshell-a784b9fea849.
27. What is parsing in NLP? 2019. September 30. URL: https://forum.huawei.com/enterprise/en/what-is-parsing-in-nlp/thread/571685-100429.
28. AI – Natural Language Processing. URL: https://www.tutorialspoint.com/artificial_intelligence/ artificial_intelligence_natural_language_processing.htm.

Стаття надійшла до редакції 04.04.2022