AIVP ADVANCES IN IMAGE AND VIDEO PROCESSING
SOCIETY FOR SCIENCE AND EDUCATION UNITED KINGDOM
ISSN 2054-7412

Spatial Grasp Language (SGL)

Peter Simon Sapaty
Institute of Mathematical Machines and Systems, National Academy of Sciences, Ukraine;
peter.sapaty@gmail.com, sapaty@immsp.kiev.ua

ABSTRACT

A full description of a high-level language for solving arbitrary problems in heterogeneous, distributed and dynamic worlds, both physical and virtual, will be presented and discussed. The language is based on holistic and gestalt principles representing semantic level solutions in distributed environments in the form of self-evolving patterns. The latter are covering, grasping and matching the distributed spaces while creating active distributed infrastructures in them operating in a global-goal-driven manner but without traditional central resources. Taking into account the existing sufficient publications on the approach developed, the paper will be showing only elementary examples using the Spatial Grasp Language and key ideas of its networked implementation.

Keywords: gestalt psychology; spatial intelligence; spatial pattern matching; Spatial Grasp Language; self-evolving scenarios; parallel networked interpretation; hybrid operations; integral solutions; distributed control.

1 Introduction

We are witnessing a dramatic change in the character of national and international activity, especially in crisis and conflict areas, with the use of asymmetric, unconventional, and hybrid solutions. They may simultaneously involve economy, ecology, international relations, ethnicity, culture, law, religion, etc., defense and military too, occupying both physical and virtual environments. And these solutions may need to be multidimensional and highly integral in order to succeed, aiming at the whole from start rather than parts in hope to achieve this whole.

A new philosophy, methodology, and supporting high-level networking technology are being developed oriented on effective management of distributed, dynamic and hybrid systems [1-6], which may be useful within the context mentioned above. They are based on holistic and gestalt ideas [7-9] rather than traditional communicating agents stemming from [10].

The approach (called over-operability [11] rather than traditional interoperability) allows for integral global-goal-driven solutions in distributed environments. It has certain psychological background in trying to follow existing ideas of how human mind operates by solving complex problems (like in waves, streams, states, etc. [12]) and inherit them by information technologies [13].

The resultant Spatial Grasp Technology (SGT) with Spatial Grasp Language (SGL) as its key element has been prototyped and tested with numerous researched applications [14-34]. In the most general terms it operates as shown in Figure 1.

DOI: 10.14738/aivp.41.1922
Publication Date: 29st,February 2016
URL: http://dx.doi.org/10.14738/aivp.41.1922

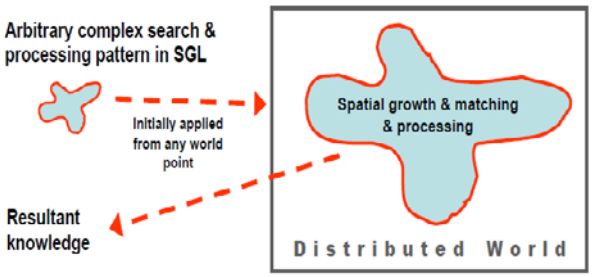

Figure 1: How SGT Operates in General

A high-level scenario for any task in a distributed world is represented as an active self-evolving pattern rather than traditional program, sequential or parallel. This pattern, expressing top semantics and key decisions of the problem to be solved spatially propagates, replicates, modifies, covers and matches the world, creating distributed operational infrastructures throughout it, with the final results retained in the environments or returned as high level knowledge to the starting point.

The current paper describes, first time, the full specification of the latest, updated and improved, version of SGL being currently used in a number of projects related to intelligent management and control of large distributed dynamic systems with both civil and defence applications. It also serves as an exemplary reference in a new patent on parallel and distributed mechanisms for SGL types of languages, which is currently in progress (succeeding the previous patent on the approach [14]).

SGL is the latest and most advanced version in a sequence of spatial languages using free however globally controlled movement of program code in networks, with the previous ones named as WAVE [1], WAVE-WP (World Processing) [2] and DSL (Distributed Scenario Language) [15].

2 SGL Orientation and Peculiarities

SGL differs fundamentally from traditional programming languages. Rather than working with information in a computer memory it allows us to directly *move through*, *observe*, and make *any actions and decisions* in fully distributed environments, whether physical or virtual. In general, the *whole distributed world*, which may be dynamic and active, is considered in SGL as a substitute to traditional computer memory, with multiple "processors" (humans, robots, any manned or unmanned units or devices, etc.) directly operating in it in a cooperative or competitive manner. An SGL program (called *scenario*) can be viewed from different angles:

- As the first linguistic means towards describing and formalizing the notion of *gestalt* [7], often allowing us to grasp top semantics, integrity and super-summative features of large complex systems.
- As an active recursive *self-matching pattern* which if applied against distributed physical, virtual, executive, or combined worlds, can cover, rule and change these worlds in the way required.
- As a sort of a universal *genetic mechanism* expressed in a special integral formalism and allowing any distributed systems, whether passive or active, to be created, grown, extended, evolved, and modified.
- As a symbolic *"soul"* implanted into the distributed world and self-spread throughout it, providing local and global awareness and control, also the world's meaning, sense, life, and consciousness.

- As a powerful and globally controlled *super-virus* which when injected from any point into the world's body can cause different effects on it, from full control and direction of evolution to complete destruction, if required.

3 The SGL Worlds

SGL directly operates with:

- *Physical World* (PW), continuous and infinite, where each point can be identified and accessed by physical coordinates expressed in a proper coordinate system (terrestrial or celestial) and with the precision given.
- *Virtual World* (VW), which is discrete and consists of nodes and semantic links between them, both nodes and links capable of containing arbitrary information, of any nature and volume. VW may be considered as finite as regards the volume of information the mankind accumulated by today, but taking into account its continuing and rapid growth, also possible existence of other civilizations in space, it may potentially be treated as infinite too.
- *Executive world* (EW), consisting of active doers with communication channels between them, where doers may represent any devices or machinery capable of operating on the previous two worlds and include humans, robots, mainframes, laptops, smartphones, etc.
- Different kinds of combination of these worlds can also be possible within the same formalism. For example, Virtual-Physical World (VPW) may allow not only for a mere mixture of the both worlds but also their deep integration, where individually named VW nodes can be associated with certain PW coordinates, thus allowing for their presence in physical reality too. On the other side, the whole regions of PW of arbitrary shape and size may have certain virtual names identifying them, and this naming can be hierarchical. Another possibility is Virtual-Execution World (VEW), where doer nodes may be associated with virtual nodes (say, in the form of special names or nicknames) assigned to them, with semantic relations in between, similarly to pure VW nodes. Execution-Physical World (EPW) can pin some or all doer nodes to certain PW coordinates and consider them inseparable of each other, and Virtual-Execution-Physical World (VEPW) can combine all features of the previous cases.

4 Top Sgl Syntax

SGL has a recursive structure with its top level shown in Figure 2. Such organization allows us to express any spatial algorithm, create and manage any distributed structures and systems, static or dynamic, passive or active, also solve any problem in, on, and over them, and this often can be expressed in a compact, transparent and unified way.

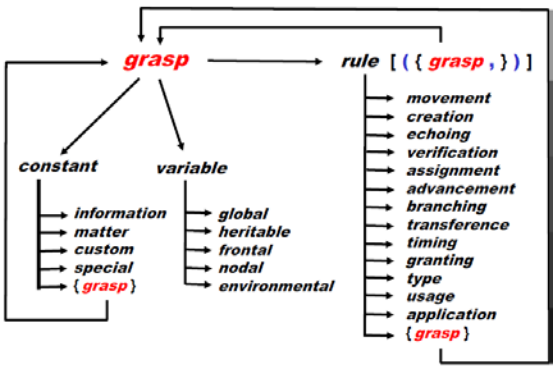

Figure 2: SGL Recursive Syntax

Let us explain the language basics in a stepwise top-down manner. The SGL topmost definition with scenario named as grasp (reflecting the spatial navigation-grasp-conquest model explained in previous chapters, rather than the usual program) can be as follows:

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

where syntactic categories are shown in italics, vertical bar separates alternatives, square brackets identify optional constructs, and parentheses and commas being the language symbols. Braces indicate repetitive parts with the delimiter (here comma) at the right.

As follows from this notation, an SGL scenario, or *grasp* (applied from a certain world point, i.e. of PW, VW, EW or their combination) in its simplest form can be just a *constant* presenting the result explicitly. It can also be a *variable* containing data assigned to it previously, say, by another SGL scenario branch which visited this point before (otherwise empty, or `nil`). The third variant is called a *rule*, which can be optionally supplied with parameters (enclosed in parentheses and separated by comma if more than one). These parameters, due to recursion, can generally be arbitrary grasps again (as constants or variables in the simplest cases, as above, up to scenarios of any complexity and space-time coverage).

The rules, starting their influence in the current world positions, can be of different natures and levels -- from local matter or information processing to full depth management and control. They can produce results which may reside in the same or other world positions. The results obtained and world positions reached by rules may become operands and/or starting positions for other rules, with new results and new positions (single or multiple) obtained after their completion, and so on.

The SGL scenario can dynamically *spread* & *process* & *match* the world or its parts needed, with the scenario code capable of virtually or physically splitting, replicating, and moving in the distributed spaces (accompanied with transitional data). This movement can take place in single or multiple scenario parts dynamically linked with each other under the overall control, the latter (both forward and backward) spreading and covering the navigated world too.

SGL constants can represent *information*, physical *matter* (physical objects including), self-identifying *custom* items (relating to information, matter or both), or *special* words used throughout the language as standard parameters or modifiers for different constructs:

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

The word “constant” is used rather symbolically in SGL definition, as the last option is recursively defined as *grasp* again. This capable of representing any objects (passive or with embedded activities) and with any structures within the recursive SGL syntax for their further processing by SGL rules.

SGL variables, called “spatial”, containing information and/or matter and serving different features of distributed scenarios, can be stationary or mobile. They are classified as *global* (with residence and mobility usually undefined), *heritable* (event-born and remaining stationary to it, being shared by all subsequent events), *frontal* (accompanying evolution, mobile), *nodal* (temporarily associated with, and stationary to, accessed world nodes), and *environmental* (external and internal world-accessing, stationary or mobile):

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

And rules belonging to the following classes:

rule → *movement* | *creation* | *echoing* | *verification* | *assignment* | *advancement* |

branching | *transference* | *timing* | *granting* | *type* | *usage* | *application* | *grasp*

The final rule's option, grasp, brings another level of recursion into SGL where operations may not only be explicitly set up in advance but rather represent results of spatial development of SGL scenarios (of any world coverage and complexity), also act in aggregates with other rules and modifiers or data on the same operands.

5 SGL Main Features

5.1 How Scenarios Evolve

In order to explain main SGL features, we will show how its scenarios generally evolve in distributed worlds, with the points following.

- SGL scenario is considered developing in *steps*, which can be *parallel*, with new steps produced on the basis of previous steps.
- Any step, including the starting one, is always associated with a certain *point* or position of the world (i.e. physical, virtual, executive, or combined) in which the scenario (or its particular part, as there may be many parts working simultaneously) is currently developing.
- Each step provides a resultant *value* (which may be single, multiple, and/or structured) representing information, matter or both, and a resulting control *state* (as one of possible states, ranging by their strength), in the same or other world point (or points) reached.
- Different scenario parts may evolve from the same step in *ordered*, *unordered*, or *parallel* manner, providing new independent or interdependent steps.
- Different scenario parts can also *succeed* each other, with new parts evolving from final steps produced by the previous parts.
- This (potentially parallel and distributed) scenario evolution may proceed in *synchronous* or *asynchronous* mode, also their any combinations.
- SGL operations and decisions in evolving scenario parts can use control states and values *returned* from other scenario parts whatever complex and remote they might be, thus combining *forward* and *backward* scenario evolution in distributed spaces.
- Different steps from the same or different scenario parts can be associated with the same world points, sharing persistent or temporary information in them.
- Staying with world points, it is possible to *change* local parameters in them, whether physical or virtual, thus *impacting* the worlds via these locations.
- Scenarios navigating distributed spaces can create arbitrary distributed physical or virtual infrastructures in them, which may operate on their own after becoming active, with or without external control. They can also subsequently (or even during their creation) be navigated, updated, and processed by the same or other scenarios.
- Overall organization of the world creation, navigation, coverage, modification, analysis, and processing can be provided by a variety of SGL rules which may be arbitrarily *nested*.

As will be shown throughout this book, any sequential or parallel, centralized or distributed, stationary or mobile algorithm operating with both information and physical matter can be written in SGL at any levels and their combinations. These can range from top semantic (like setting global goals, basic

operations, and key decisions only) to those detailing system partitioning, composition, subordination between components and overall management and control.

6 Sense and Nature of Rules

In explaining the language basics further, let us shed some light on the general sense and nature of rules, to be explained later in detail. A rule representing in SGL any action or decision may, for example, belong to the following categories:

- Elementary arithmetic, string, or logic operation.
- Move or hop in a physical, virtual, execution or combined space.
- Hierarchical fusion and return of (potentially remote) data.
- Distributed control, both sequential and parallel, and in breadth or depth.
- A variety of special contexts detailing navigation in space and the character of embraced operations and decisions.
- Type and sense of a value or its chosen usage, guiding automatic language interpretation.
- Creation or removal of nodes and/or links in distributed knowledge networks, allowing us to work with arbitrary structures, including their initial creation and any modification.
- A rule can also be a compound one integrating other rules whether elementary or compound again, due to recursion.

All rules, regardless of their nature, sense or complexity, are pursuing the same ideology and organization, as follows.

- They start from a certain world position, being initially linked to it.
- Perform or control the needed operations in a distributed space, which may be stepwise, parallel, and arbitrarily complex.
- Produce concluding results by the final steps, expressed by control states and values there.

These final steps may associate with the same (where the rule started) or new world positions, reached by the rule's activity.

This uniformity allows us to effectively compose integral and transparent spatial algorithms of any complexity and world coverage, operating altogether under unified and automatic (generally parallel and distributed) control.

6.1 Spatial Variables

Let us consider some more details on the nature and sense of spatial variables, stationary or mobile, which can be used in fully distributed physical, virtual or executive environments, effectively serving multiple cooperative processes under the unified control. They are created upon declaration by special rules, see later, or by first assignment to them.

- *Global variables* – the most expensive ones, which can serve any SGL scenarios and can be shared by their different branches. Their locations, mobility capabilities, and life span can depend on the features of distributed environments and SGL implementations.

- *Heritable variables* – stationary, appearing within a scenario step and serving only all subsequent steps, generally multiple and parallel (not from other branches), which can share them in both read and write operations.
- *Frontal variables* – mobile, temporarily associated with the current step and not shared with other parallel steps; they are following scenario evolution being transferred between subsequent steps. These variables replicate if from a step a number of other steps directly emerge. (The replication procedure, also physical mobility, may have implementation peculiarities if working with physical matter rather than information.)
- *Environmental variables* – these allow us to access, analyze, and possibly change different features of physical, virtual and execution words during their navigation. Most of them are stationary, associated with the world positions reached, but some, especially related to the language execution, can be mobile, some even global like the absolute time.
- *Nodal variables* – stationary, being a sole property of the world positions reached by the scenarios. Staying at world nodes, they can be accessed and shared by all activities having reached these nodes under the same scenario identity, and at any time.

These types of variables, especially when used together, allow us to create advanced algorithms working directly in space, actually *in between* components of distributed systems rather than *in* them, providing flexible, robust and self-recovering solutions (stealthy as well if needed). Such algorithms can freely self-replicate, partition, spread and migrate in distributed environments (partially or as an organized whole), while always preserving overall awareness and global goal orientation.

6.2 Control States and Their Hierarchical Merge

The following control states can appear after performing different scenario steps. Indicating local progress, they can be used for distributed control of multiple processes, allowing us to make proper decisions at a variety of levels.

- `thru` – reflects *full success* of the current branch of the scenario with capability of further development (i.e. indicating successful operation not only *in* but also *through* this step of control). The following scenario steps, if any, will be allowed to proceed from the current step.
- `done` – indicates success of the current scenario step as its *planned termination*, after which no further development of this branch from the current step will be possible. This state can, however, be subsequently changed to *thru* at higher levels by a special rule, as explained later.
- `fail` – indicates non-revocable failure of the current branch, with no possibility of further development. This state directly relates to the current branch and step only. It, however, can influence decisions at higher levels by rules concerning engagement of other branches (same can be said about the previous two states).
- `fatal` – reports *fatal, terminal failure* with nonlocal effect, triggering abortion of all currently evolving scenario processes and removal of all associated temporary data, regardless of their current world locations and operational success. The scope of this spreading termination process may be the whole scenario, by default, or it may be restricted by a certain rule explained later (supervising the scenario part in which this state may potentially occur).

These control states appearing in different branches of a parallel and distributed scenario at bottom levels can be used to obtain generalized control states for higher levels, up to the whole scenario, for making proper decisions. The hierarchical bottom-up merge and generalization of states is based on

their comparative importance, or power, where the stronger state will always dominate when ascending towards the decision root.

For example, merging states `thru` and `done` will result in `thru`, thus generally classifying successful development at a higher scenario level with possibility of further expansion from at least some of its branches. Merging `thru` and `fail` will result in `thru` too, indicating general success with possibility of further development despite some branch (or branches) terminated with failure, while the others remaining open to further evolution. Merging `done` and `fail` will result in `done` indicating generally successful termination while ignoring local failures, however, without possibility of further development in all these directions. And `fatal` will always dominate when merging with any other states unless its destructive influence is contained within a certain higher level rule, as already mentioned (the latter will itself terminate with *fail* in such a case). So ordering these four states by their powers from maximum to minimum will be as follows: `fatal`, `thru`, `done`, `fail`.

These four states, their merge, and use in control rules are standard, language-embedded ones. SGL, as a universal spatial language, also allows us to artificially set up any possible control states, with any numbers and any merge or generalization procedures, which may include the mentioned standard ones or be completely different.

7 Description of Main SGL Constructs

7.1 Constants

7.1.1 Information

String can be represented as any sequence of characters embraced by opening-closing single quotation marks. This sequence should not contain the single quotes itself or they should appear in opening-closing pairs only, with any nesting allowed.

Examples: ‘John’, ‘Peter and Paul’.

Instead of single quotes, a sequence of characters can also be placed into opening-closing curly brackets (or braces {}), which can be used inside the string in pairs too. Braces will indicate the text as a potential *scenario* code which can be immediately optimized (like removing unnecessary spaces and/or adjusting to the standard SGL syntax, say, after using constructs typical to other programming languages for convenience, as explained later). If single quotes are used to embrace texts as a potential SGL code, such code optimization will have to be done during its interpretation, not before, and each time it is involved, with the original text remaining intact.

Number can be represented in a standard way, similar to traditional programming languages, generally in the form: [*sign*]{*digit*}[.{*digit*}[E[*sign*]{*digit*}]].

Examples: `105`, `88.56`, `-15`, `3.3E-5`.

Numbers can also use words instead of digits and accompanying characters (using underscore as separator if more then a single word needed), as follows:

`zero`, `one`, `two`, `three`, `four`, `five`, `six`, `seven`, `eight`, `nine`, `ten`, `eleven`, `twelve`, `thirteen`, `fourteen`, `fifteen`, `sixteen`, `seventeen`, `eighteen`, `nineteen`, `twenty`, `thirty`, `forty`, `fifty`, `sixty`, `seventy`, `eighty`, `ninety`, `hundred`, `thousand`, `million`, `billion`, `trillion`, `dot`, `minus`, `plus`.

The four examples above may look like follows.

a) with mixed representation:

`hundred_five, eighty_eight.56, minus_fifteen, three.3E-five`

b) up to the full and detailed wording:

`one_zero_five, eight_eight_dot_fifty_six, minus_one_five, three_dot_three_E_minus_five`

7.1.2 Physical Matter

Physical *matter* (incl. physical objects) can be represented by a sequence of characters embraced by opening-closing double quotation marks.

Examples: “truck”, “white sand”, “brick”, “water”.

The above mentioned self-identified constants (i.e. strings, scenarios, numbers, and matter) may also be set up by explicit naming their types with the use of corresponding rules.

7.1.3 Custom Constants

For extended applications, other self-identified constants can be introduced too, if not conflict with the language syntax, to be directly interpreted by an extended SGL interpreter. For example, these may be coordinates in physical spaces similar to `x17.5`, `y44.2`, `z-77`, as well as their combination: `x17.5_y44.2_z-77`, or internet addresses like `http://www.amazon.com/`. Special type-defining rules can be used for more complex cases.

7.1.4 Special Constants

Special verbal constants can be used as standard parameters (or modifiers) in different language rules, as will be shown later. The basic list of such words (consisting of lower case letters only) with comments on their possible use is as follows:

- `thru` – indicates (or sets) control state as a success with possibility of further evolution.
- `done` – indicates (or sets) control state as a successful termination, with blocking further development.
- `fail` – indicates (or sets) control state as failure, without further development.
- `fatal` – indicates (or sets) control state as absolute failure, with abortion of active distributed processes.
- `infinite` – indicates infinitely large value.
- `nil` – indicates no value at all.
- `any`, `all`, `other` – stating that any, all, or other (i.e. except the current one) elements under consideration can be used.
- `passed` – hinting that the world nodes to be considered have already been passed by the current scenario branch.
- `existing` – hinting that world nodes with the given names are already existing and should not be created again (i.e. duplicated).
- `neighbors` – stating that the nodes to be accessed are among direct neighbors of the current node, i.e. within a single hop from it by existing links.

- `direct` – stating that the mentioned nodes should be accessed or created (if not exist) from the current node directly, regardless of possible (non)existence of direct links to them.
- `noback` – not allowing to return to the previously occupied node.
- `firstcome` – allowing to access the next-hop nodes only first time with the given scenario ID.
- `forward`, `backward` – allowing to move from the current node via existing links along or against their orientations (ignored when dealing with non-oriented links, which can be traversed in both directions).
- `global`, `local` – may indicate the scope of operations or the world access in different rules.
- `sync[hronous]`, `async[hronous]` – a modifier setting synchronous or asynchronous mode of operations induced by different rules.
- `virtual`, `physical`, `executive` – indicating or setting the type of a node the scenario is currently dealing with (the node can also be of a combined type).
- `engaged`, `vacant` – indicating or setting the state of a resource the current scenario is dealing with (like, say, human or robot, or any physical, virtual or combined world node).
- `existing` – indicating that the node (or nodes) of interest are already existing.
- `passed` – indicating that the nodes under consideration have already been passed by the current scenario branch.

7.1.5 Compound Constants, Grasps

Constants can also be arbitrarily complex, as aggregates (possibly hierarchical) from elementary types (not necessarily the same) described above, being supported by the full SGL syntax (i.e. generally as *grasps* again). They can be composed by using either standard rules described later or, if not sufficient, any additional, custom ones oriented on specific application areas.

7.2 Variables

Different types of variables can be self-identifiable, i.e. by the way their names are written. Variables of different types can also have any identifiers if explicitly declared by special rules, explained later.

7.2.1 Global, Heritable, Frontal, and Nodal Variables

The sense and use of these variables have been explained before, in Section 4.3. In the case of self-identification, they should start with capital letters G, H, F or N, respectively, followed by a sequence of alphanumeric characters (letters and digits only).

Examples: Globe, H214b, Frontal5, Nina37.

7.2.2 Environmental Variables

All these variables have specific names written in all capital letters, with brief explanation of their sense and usage following.

TYPE – indicates the type of a node the current step associates with. This variable returns the node's type (i.e. `virtual`, `physical`, `executive`, or their combination as a list with more than one value). It can also change the existing type by assigning to it another value (simple or combined too) if needed.

CONTENT – returns content of the current node (only if having virtual or executive dimension, or both), which can be any string of characters (in the simplest case the latter just serving as its name). Assigning to CONTENT allows us to change the existing node's content when staying in it. In a purely physical node CONTENT returns `nil` (as physical nodes can be identified by their addresses only). If a node is of both virtual and executive nature, this variable deals with the virtual one.

ADDRESS – returns address of the current virtual node. This is read-only variable as node addresses are set up automatically by the underlying distributed interpretation system during the creation of virtual nodes, or by a system it has been put on top of (for example, it can be an internet address of the node).

QUALITIES – identifies a list of available physical parameters associated with the current physical position, or node, depending on the chosen implementation and application (for example, these may be temperature, humidity, air pressure, visibility, radiation, noise or pollution level, density, salinity, etc.). These parameters (generally as a list of values) can be obtained by reading the variable. They may also be changed (depending on their nature and implementation system capabilities) by assigning new values to QUALITIES, thus locally influencing the world from its particular point (or at least attempting to).

WHERE – keeps physical coordinates of the current physical node in the chosen coordinate system (the node can be combined one, additionally having virtual and/or executive features). These coordinates can be obtained by reading the variable. Assigning a new value to this variable causes physical movement of the current node into the new position (while preserving its identity, all information surrounding, and control and data links with other nodes).

BACK – keeps internal system link to the preceding world node (virtual, executive or combined one with virtual or executive dimension), allowing the scenario to most efficiently return to the previously occupied node, if needed. Referring to internal interpretation mechanisms only, the content of BACK cannot be lifted, recorded, or changed from the scenario level.

PREVIOUS – refers to an absolute and unique address of the previous virtual node (or combined with execution and/or physical dimensions), allowing us to return to the node directly. This may be more expensive than using BACK, but the content of PREVIOUS, unlike BACK, can be lifted, recorded, and used elsewhere in the scenario.

PREDECESSOR – refers to the content/name of the preceding world node (the one with virtual or executive dimension). Its content can be lifted, recorded, and used subsequently, including for organization of direct hops to this node. Such hops, however, can also lead to other nodes with the same content/name, as node contents/names are generally not unique throughout the world operated in SGT. Assigning to PREDECESSOR can change content/name of the previous node.

DOER – keeps a name of the device (say, laptop, robot, smart sensor, or even a human) which interprets the current SGL code. This device can be chosen for the scenario automatically, say, from the list of offered ones, or just picked up from those known or guessed to be available. It can also be appointed explicitly by assigning its name to DOER, causing the current SGL code move into this device and execute there unless it terminates or another device is assigned to DOER, say, when the current one becomes inefficient or fails.

RESOURCES – keeps a list of available or recommended resources (human, robotic, electronic, mechanical, etc., by their types or names) which can be used for execution of the current and subsequent parts of the SGL scenario. This list can contain potential doers too, which after being

selected by different scenario branches appear (by their names) in variables DOER associated with the branches. RESOURCES can be accessed and changed by assignment, and in case of distributed SGL interpretation it can be replicated with its content, the latter, possibly, partitioned between different branches by the internal interpretation planning and optimization procedures.

LINK – keeps a name (same as content) of the virtual link which has just been passed. By assigning new value to it you can change the link's content/name. Assigning `nil` or empty to LINK removes the link passed.

DIRECTION – keeps direction (along, against, or neutral) of the passed virtual link. Assigning to this variable values like `plus`, `minus`, or `nil` (same as `+`, `-`, or empty) can change its orientation or make non-oriented.

WHEN – assigning value to this variable sets up an absolute starting time for the following scenario branch, thus allowing us to suspend and schedule certain operations and their groups in time.

TIME – returns current absolute time, being read-only global variable.

SPEED – reflects speed of physical movement of the node (physical, executive or combined, the latter may include virtual dimension too) in which control (represented by the current step) is staying. By assigning to this variable, you can change the speed of the current node. In case of a pure virtual node, the notion of speed is irrelevant and will return `nil` when accessed, also causing no effect when assigned to.

STATE – can be used for explicit setting of control state of the current step by assigning to it one of the following: `thru`, `done`, `fail`, or `fatal`. (These states, as mentioned before, are also generated implicitly, automatically on the results of success or failure of different operations, belonging to the overall internal control of scenarios.) Reading STATE will always return `thru` as this could only be possible if the previous operation terminated with `thru` too, thus letting this operation to proceed. A certain state explicitly set up in this variable can be used subsequently at higher levels (possibly, together with termination states of other branches) within distributed control provided by SGL rules, whereas assigning `fatal` to STATE causes already mentioned abortion of distributed processes with associated data.

VALUE – when accessed, returns the resultant value of the latest operation (say, an assignment to a variable or just naming a variable or constant). Assignment to VALUE leaves its content available to the next operation. This variable allows us to organize balanced processing combining sequences of operations with their representation as nested expressions in SGL. (As follows from syntax of Fig. 1, the resultant values of operations can also be accessed implicitly if these operations or their sequences are themselves standing as operands of higher level rules.)

COLOR – keeps identity of the current SGL scenario or its branch, which propagates together with the scenario and influences grouping of different nodal variables under this identity at world nodes. This means that different scenarios or their branches with different identities are protected from influencing each other via the use of identically named nodal variables. However, scenarios with different colors can penetrate into each other information areas if they know the other's colors, by temporarily assigning the needed new identity to COLOR (to perform cooperative or stealth operations) while restoring the previous color afterwards. Any numerical or string value can be explicitly assigned to COLOR. By default, different scenarios are implicitly assigning the same value in

COLOR at the start, thus being capable of sharing all information at navigated nodes, unless change their personal color themselves.

IN – special variable reading from which asks for data from the outside world in the current point of it; this input data becoming its resultant value.

OUT – special variable allowing us to send information from the scenario to the outside world in its current point, by assigning the output value to this variable.

STATUS – retrieving or setting the status of a doer node in which the scenario is currently staying (engaged or vacant, possibly, with a numerical estimate of the level of engagement or vacancy). This feedback from the implementation layer could be useful for a higher-level supervision, planning, and guidance of the use and distribution of resources executing the scenario, rather than doing this fully automatically by standard procedures which may not always be optimal, especially under resource shortages.

Other environmental variables for extended applications can be introduced and identified by unique words in all capitals too, or they may use any names if explicitly set up by a special rule, as mentioned later.

As can be seen, most environmental variables are serving as stationary ones, except RESOURCES and COLOR, which are mobile. The global variable TIME may symbolically be considered as stationary too but in reality may depend on implementation details.

7.3 Rules

The concept of *rule* is not only dominant in SGL for setting most diverse activities ranging from elementary data & knowledge & physical matter processing to overall management and control, but also *the only one*. This provides a universal, integral and unified approach to expressing any operations in distributed dynamic worlds, and if needed, in parallel and fully distributed mode. This section describes the main repertoire of introduced and researched SGL rules with summary of their sense and possible applications.

7.3.1 Movement

Rules of this class result in virtual hopping to the existing nodes (the ones having virtual or executive dimensions) or real movement to new physical locations, associating the remaining scenario (with current frontal variables and control) with the nodes reached. The resultant values of the movements are represented by the reached node names (in case of virtual, executive or combined nodes) or nil in case of pure physical nodes, with control state thru in them if the movement was successful. If no destinations reached, the movement results with state fail and value nil.

hop – sets virtual propagation to node(s) in virtual, execution, or combined worlds (the latter may have physical dimension too), directly or via links connecting them. In case of a direct hop, except node name or address, special modifier direct should be included into parameters of the rule. If a hop to take place from a node to a node via an existing link, both destination node name/address and link name (with orientation if needed) should be among parameters of the rule. This hop rule can also cause independent and parallel propagation to a number of nodes if there are more than one node connected to the current one by the named link, and only link name mentioned (or given by indicator all, for all links involved). In a more general case, parallel hops can be organized from the current node if the destination attributes are given by a list of names/addresses of nodes and names of links (or direct or all indicators) which should lead to them.

`move` – sets real movement in physical world to a particular location given by coordinates in a chosen coordinate system. The destination location becomes a new temporary node with no name (or `nil`) which disappears when all current scenario activities leave it for other nodes. If, however, the destination node is to have virtual dimension too (indicated by `virtual` in the parameters of the rule, possibly, accompanied by a certain name otherwise default name used), it will remain intact and can be accessed by other scenarios or different branches of the current one unless removed explicitly.

`shift` – differs from the `move` only in that movement in physical world is set by deviations of physical coordinates from the current position rather than by absolute physical coordinates.

`follow` – allows us to propagate in both virtual and physical spaces by following arbitrary routes set up by sequences of links, nodes, physical coordinates, etc., or via obtained internal interpretation tracks using recorded entries to them (as explained later).

7.3.2 Creation

This class of rules creates or removes nodes and/or links leading to them during distributed world navigation. After the creation, the resultant values will be their names (there may be more than one destination node created) with termination state `thru`, and the next steps will be associated with the nodes reached, starting in them. If the operation fails, its resultant value will be `nil` and control state `fail` in the node it started. After the node(s) successful removal operation, the resultant value in the starting node will be the same as before and control state `thru`.

`create` – starting in the current world position, creates either new virtual link-node pairs or new isolated nodes. For the first case, the rule is supplied with names and orientations of new links and names of new nodes these links should lead to, which may be multiple. For the second case, the rule has to use modifier `direct` indicating direct nodes creation, i.e. without links to them. If to use modifiers `existing` or `passed` for the link-node creation hinting that such nodes already exist (also if nodes are given by addresses, thus indicating their existence) only links will be created to them by `create`.

`linkup` – just simplifies the latest rule, creating only links with proper names from the current node to the already existing nodes, without the need to use modifiers `existing` or `passed`. However, still using modifier `passed` may help us narrow direct search of the already existing nodes.

`delete` – removes links together with nodes they should lead to, starting from the current node. Links and nodes to be removed should be either explicitly named or represented by modifiers `any` or `all`. Using modifier `direct` instead of link name together with node name will allow us to remove such node (or nodes) from the current node directly. In all cases, when a node is deleted, all its links with other nodes will be removed too.

`unlink` – removes only links leading to neighboring nodes where, similar to the previous case, they should be explicitly named or modifiers `any` or `all` used instead. The resultant values on the rule will be represented by these node names, with states `thru` in them, similar to `hop` and `linkup` operations. The next scenario step will start in these neighboring nodes.

The above creation rules, depending on the implementation, can also be used in a broader sense and scale, as *contexts* embracing arbitrary scenarios and influencing hop operations within their scope (the same scenarios will be capable of operating in creation or deletion mode with them).

7.3.3 Echoing

The rules of this class use terminal control states and terminal values from the embraced scenario (which may be remote) to obtain the resultant state and value in the world point it started, also being it's *terminal* point (from which the rest of the scenario, if any, will develop). The usual resultant control state for these rules is `thru` (`fail` occurs only when certain terminal values happen to be unavailable or result unachievable, say, as division by zero). Depending on the rule's semantics, the resultant value can be compound, like a list of values, which may be nested.

`state` – returns the resultant generalized state of the embraced SGL scenario upon its completion, whatever its complexity and space coverage. This state being the result of the ascending fringe-to-root generalization of terminal states of the scenario embraced, where states with higher power (from max to min as: `fatal`, `thru`, `done`, `fail`) dominate in this potentially distributed and parallel process, as already mentioned. The resultant state returned is treated as the *resultant value* on the rule, the latter always terminating with own control state `thru`, even in the case of resultant `fatal`, thus restricting its spreading by echo rules. (Another restriction of influence of `fatal` by a special rule will be explained later.)

`order` – returns an ordered list of final values of the scenario embraced corresponding to the order of launching related branches rather than the order of their completion. For parallel branches these orders may, for example, relate to how they were activated, possibly, with the use of time stamping upon invocation.

`rake` – returns a list of final values of the scenario embraced in an arbitrary order. This order may, for example, depend on the order of completion of branches; it can also be influenced by peculiarities of the echoing collection procedure of the results.

`sum` – returns the sum of all final values of the scenario embraced.

`count` – returns the number of all resultant values associated with the scenario embraced, rather than values themselves as by the previous rules.

`first`, `last`, `min`, `max`, `random`, `average` – return, correspondingly, the first, the last, minimum, maximum, random, or average value from all terminal values returned by the scenario embraced, where `first` and `last` will depend on ordering of the results with details similar to the rule `order` above.

`element` – returns the value of an element of the list on its left operand by index or content (see corresponding usage rules later) given by the right operand. If the right operand is a list of indices/contents, the result will be the list of corresponding values from the left operand. If `element` is used within the left operand of assignment (explained later), instead of returning values it will be providing an access to them.

`sortup`, `sortdown` return an ordered list of values produced by the operand embraced, starting from maximum or minimum value and terminating, correspondingly, with minimum or maximum one.

`reverse` – changes to the opposite the order of values from the embraced operand.

`add`, `subtract`, `multiply`, `divide`, `degree` – perform the corresponding operations on two or more operands of the scenario embraced. If the operands represent multiple values as lists, these operations are performed between the peer elements, with the resulting value being multiple too.

`separate` – separates the left operand string value by the string at the right operand used as a delimiter in a repeated manner for the left string, with the result being the list of separated values. If the right operand is a list of delimiters, its elements will used sequentially and cyclically unless the string at the left is fully partitioned. If the left operand represents a list of strings, each one is separated by the right operand as above, with the resulting lists of separated values merged into a common list in the order they were received.

`unite` – integrates the list of values at the left (as strings, or to be converted into strings automatically if not) by a repeated delimiter as a string (or a cyclic list of them) at the right into a united string.

`attach` – makes the resultant string by connecting the right string operand to the end of the left one. If operands are lists with more than one element, the attachment is made between their peer elements, receiving the resultant list of united strings. This rule can also operate with more than two operands.

`append` – forms the resultant list from left and right operands, appending the latter to the end of the former, where both operands may be lists themselves. More than two operands can be used too.

`common` – returns intersection of two or more lists as operands, with the result including same elements of all lists, if any, otherwise `nil`.

`withdraw` – its result will be the first element of the list provided by the embraced operand, with this element also simultaneously withdrawn from the list (the latter makes sense only for a variable containing a list of values as the operand). This rule can work with more than one element by adding another operand providing the number of elements to be withdrawn and represented as the result.

`access` – returns an internal access (which can be recorded, say, in a variable) to all terminal positions of the embraced scenario, which can be used to reenter them most efficiently afterwards (on internal system level). This reentry may be performed by the rule `follow` described before.

7.3.4 Verification

These rules provide control state `thru` or `fail` reflecting the result of certain verification procedures, also `nil` as own resultant value, while remaining in the same world positions after completion.

`equal`, `notequal`, `less`, `less[or]equal`, `more`, `more[or]equal`, `bigger`, `smaller`, `heavier`, `lighter`, `longer`, `shorter` – make comparison between left and right operands, which can represent information or physical matter, or both. In case of vector operands, state `thru` appears only if all peer values satisfy the condition set up by the rule (except `notequal`, for which even a single non-correspondence between peers will result in `thru`). The list of such rules can be easily extended for more specific applications, if supported properly on implementation level.

`empty`, `nonempty` – checks for emptiness (i.e. non-existence, same as `nil`) or non-emptiness (existence) of the resultant value obtained from the embraced scenario.

`belongs`, `notbelongs` – verifies whether the left operand value (single or a list) belongs as a whole to the right operand, potentially a list too.

`intersects`, `notintersects` – verifies whether there are common elements (values) between left and right operands, being generally lists. More than two operands can be used for this rule too, with at least a same single element to be present in all of them to result in `thru`.

7.3.5 Assignment

This class of rules assigns the result of the right scenario operand (which may be arbitrarily remote, also as a list of values) to the variable or set of variables directly named or reached by the left scenario operand, which may be remote too. The left operand can also provide pointers to certain elements of the reached variables which should be changed by the assignment rather than the whole variables (see rule `element` above). These rules will leave control in the same world position they've started, its resultant state `thru` if assignment was successful otherwise `fail`, and the same value as assigned to the left operand. There are two options of the assignment.

`assign` – assigns the same value of the right operand (which may be a list) to all variables accessed (or their elements pointed) by the left operand. If the right operand is represented by `nil` or empty, the left operand variables as a whole (or only their elements pointed) will be removed.

`assignpeers` – assigns values of different elements of the list on the right operand to different variables (or their pointed elements) associated with the destinations reached on the left operand, in a peer-to-peer mode.

7.3.6 Advancement

Rules of this class organize forward or "in depth" advancement in space and time. They can work in synchronous or asynchronous mode using modifiers `sync`[`hronous`] or `async`[`hronous`] (the second one optional as asynchronous is default mode).

`advance` – organizes stepwise advancement in physical, virtual, executive or combined spaces, also in a pure computational space while staying in the same world nodes (thus moving in time only). For this, the embraced SGL scenarios are used in a sequence, as written, where each new scenario applies from all terminal world nodes reached by the previous scenario (these nodes may happen to be the same as before if only computations took place). The resultant world positions and resultant values on the rule are associated with the final steps of the final scenarios on the rule. And the rule's resultant state is a generalization of control states associated with its final steps. The frontal variables, if any, are being inherited at new steps from the preceding steps (with their copies removed from the previous positions), thus moving from one step to another, and between scenario operands, being also replicated if multiple steps emerge from a previous step.

If no final step occurs with states `thru` or `done`, the whole advancement on this rule is considered as failed (with generalized state `fail`), resulting in no possibility to continue the scenario evolution in this direction. On default or with modifier `asynchronous`, the sequence of scenarios develops in space and time independently in different directions, and different operand scenarios in the sequence may happen to be active at the same time. With the use of `synchronous` modifier, all invocations of every new scenario in their sequence can start only after full completion of all invocations of the previous scenario.

`slide` – works similar to the previous rule unless the next scenario fails to produce resultant state `thru` or `done` from some world node; in this case the next scenario from their sequence will be applied from the same starting position, and so on. The resultant world nodes and values in them will be from the last successfully applied scenario (not necessarily the same in their sequence when independently developing in different directions). The results on the whole rule, in their extreme, may even happen to correspond to the existing results in the node the rule started (including node's position) before the rule's application, with state `thru` always being the resultant state in any cases. Both synchronous and asynchronous modes of parallel interpretation of this rule, similar to the previous rule `advance`, can

be possible, where in the synchronous case different scenarios can start only after full completion of the previous ones.

`repeat` – invokes the embraced scenario as many times as possible, with each new iteration taking place from all final positions with state `thru` reached by the previous invocation. If no final steps of the scenario invocation completed with state `thru`, the starting position from which this iteration failed together with its value will be included into the set of final positions and values on the whole rule (and this set may have positions from different iterations).

Similar to the previous rule `slide`, in the extreme case the final set of positions on the whole rule may happen to contain only the position from which the rule started, with state `thru` and value it had at the beginning. By supplying additional numeric modifier to this rule, it is possible to explicitly limit the number of allowed repetitions of the embraced scenario (of course, the operand scenario may be organized to properly control the needed number of iterations itself, but with additional modifier this may be more convention is come cases).

Both synchronous and asynchronous modes of parallel interpretation of this rule, similar to the previous rules `advance` and `slide` are possible. In the synchronous mode, at any moment of time only the same scenario iteration can develop in a potentially distributed space-time continuum, whereas in the asynchronous case these may happen to be different iterations working in parallel.

7.3.7 Branching

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 branching rule. Branching may be static and explicit if we have a clear set of individual operand scenarios separated by comma. It can also be implicit and dynamic, as explained later. For all branching rules that follow, the frontal variables associated with the starting position will be replicated together with contents, with the copies obtained developing independently within different branches. The original variable will be removed from the starting position then. Details of this replication if variable holds physical matter rather than information can depend on the application and implementation details.

`branch` – most general variant with logical independence of scenario operands from each other, and any possible order of their invocation and development from the starting position (from strictly sequential to fully parallel, and from chaotic to absolutely ordered). The resultant set of positions and associated values will unite all terminal positions & values on all scenario operands involved, and the resultant control state on the whole rule is the generalization of generalized states on all scenario branches.

`sequential` – organizing strictly sequential invocation of all scenario operands, regardless of their resultant generalized control states, and launching the next scenario only after full completion of the previous one. The resultant set of positions, values, and rule's control state will be same as for `branch`.

`parallel` – organizing fully parallel development of all scenario operands from the same starting position (at least as much as this can be achieved within existing environment, resources, and implementation). The resultant set of positions, values, and rule's control state will be same as for the previous two rules.

`if` – usually has three scenario operands. If the *first* one results with generalized termination state `thru` or `done`, the *second* scenario is activated, otherwise the *third* one will be launched. The resultant

set of positions & associated values will be exactly the same as achieved by the second or third scenarios after their completion. If the third scenario is absent and the first one results with `fail`, the resultant position will be the one the rule started from, with state `thru` and value it had at the start. If only a *single* operand (i.e. the first one) is under the rule, it will also result with its starting position, initial value in it, and state `thru`, regardless of the generalized termination state of this single operand, its positions reached and values in them (all these will be ignored for the further scenario development, if any).

`or` – allows *only one* operand scenario in their sequence (not specifying which, may be any) with the resulting state `thru` or `done` to be registered as successful and resultant, with the resulting positions & associated values on it to be the resulting ones on the whole rule. The activities of all other scenario operands and all results produced by them will be cancelled. If no branch results with `thru` or `done`, the rule will terminate with `fail` and `nil` value. Used in combination with the previous rules `sequential` and `parallel`, it may have the following more clarified and detailed options.

`orsequential` – launches the scenario operands in a strictly sequential manner, one after the other as they are written, waiting for their full completion before launching the next one, unless the first one replying with generalized state `thru` or `done`, providing the result on the rule as a whole. Invocation of the remaining scenarios in the sequence will be aborthed, and all results of the previous scenarios will be removed.

`orparallel` – activates all scenario operands in parallel from the same current position, with the first one in time replying with generalized `thru` or `done` being registered as the resultant branch for the rule. All other branches will be forcefully terminated without waiting for their completion (or just ignored, depending on implementation, which in general may not be the same as the termination for global results)

The resultant scenario in all three cases above provides its final set of positions with values and states in them as the result on the whole rule. If no scenario operand returns states `thru` or `done`, the whole rule will result with state `fail` in its starting position and `nil` as resultant value.

`and` – activates each scenario operand from the same position, demanding all of them to return generalized states `thru` or `done`. If at least a single operand returns generalized `fail`, the whole rule results with state `fail` and `nil` value in the starting position while forcefully terminating the development of all other branches, which may still be in progress. If all operand scenarios succeed, the resulting set of positions unites all resultant positions on all operands with their associated values. Combining the rule with rules `sequential` and `parallel` (as we did for `or`) clarifies their activation and termination order, as follows. (These two options can, in principle, produce differing general results if different scenario operands work in intersecting domains and share intermediate results.)

`andsequential` – activates each scenario operand from the same position in the written order, terminating the rule when first one resulting with fail, while ignoring other operands and removing all results produced by this and all previous operands.

`andparallel` – activates each scenario operand from the same position, terminating the rule when the first one in time results with `fail`, while aborting all other operands activity and removing all results produced by the current one.

`choose` – chooses a scenario branch in their sequence *before* its execution, using certain parameters among which, for example, may be its numerical order in the sequence (or a list of such orders to select more than one branch). This rule can also be aggregated with other rules like `first`, `last`, `random`,

or `any` clarifying the branch to be chosen (used here as modifiers among parameters rather than rules). The resultant set of positions, their values and states will be taken from the branch(es) chosen.

`firstrespond` – selects the first branch in time replying its complete termination, regardless of its generalized termination state, which may happen to be `fail` too, even though the other branches (to be forcefully terminated afterwards) could respond later with `thru` or `done`. The set of positions on this selected branch and their associated values (if any) will be taken as those for the whole rule. This rule assumes that different branches are launched independently and in parallel. But it differs fundamentally from the rule `orparallel` as the latter selects the first in time branch replying with success (i.e. `thru` or `done`) for which, in the worst case, all branches may need to be executed in full to find the branch needed. A modification of this rule my have an additional parameter establishing, for example time limit within which replies are expected or allowed from branches (where there may be more than one branch as the result), otherwise failure if no branch responded in time.

`cycle` – repeatedly invokes the embraced scenario from the same starting position until its resultant generalized state remains `thru` or `done`, where on different invocations same or different sets of resultant positions with different values may emerge. The resultant set of positions on the rule will be an integration of all positions on all successful scenario invocations with their values. If no invocation of the embraced scenario succeeds, the resultant state `fail` in the starting position and `nil` value will emerge.

`loop` – differs from the previous rule in that the resultant set of positions on it being only the set produced by the *last* successful invocation of the embraced scenario (it will terminate, as before, with `fail` and `nil` in the starting position if no invocation succeeds).

`sling` – invokes repeatedly the embraced scenario until it provides state `thru` or `done`, resulting in the same starting position with state `thru` and its associated value when the last iteration results with `fail`.

`whirl` – endlessly repeating the embraced scenario from the starting position regardless of its success or failure with no resultant positions or values produced. External forceful termination of this construct may be needed, like using first in time termination of a competitive branch (say, under higher-level rule `orparallel`).

It could also be possible to set a limit on the number of repetitions (or duration time) in these cycling-looping-slinging-whirling rules – by supplying them with an additional parameter restricting the repeated scenario invocations.

`split` – performs, if needed, additional static or dynamic partitioning of the embraced scenario to different branches, especially in complex and not clear at first sight cases, all starting from the same current position. It may be used alone or in combination with the above mentioned branching rules, preparing separate branches for the latter. Some examples follow.

- If `split` embraces explicit branches separated by commas, it does nothing as the branches are already declared.
- It the embraced single operand represents broadcasting move or hop (creative or destructive including) in multiple directions, the branches are formed from all possible variants of elementary moves or hops, *before* their execution.

- If the rule's operand is an arbitrary scenario (not belonging to the two cases above), the branches are formed *after* their completion, where each position reached (with associated values) starts a new branch.
- If an arbitrary scenario terminates with a single or multiple positions which have multiple values associated with them (i.e. lists), each constituent value in these lists starts an individual branch, becoming its sole value.

`fringe` – being the most general variant of splitting for any scenario after its execution, is considering all final positions reached by the scenario as individual branches. It may also have additional parameters helping us to select or reject the received branches as candidates for a further scenario evolution (possibly, with involvement of both forward and echo operations over the control hierarchy produced by the scenario, for making proper decisions).

7.3.8 Transference

This class of rules organizes different control or data transference activity.

`run` – transfers control to the SGL code (treated as a procedure) resulting from invocation of the embraced scenario (which can be of arbitrary complexity and space coverage). The procedure (or procedures, if a list of them) obtained in such a way and activated will produce the resultant set of positions with associated values and control states as the result on the rule, similar to other rules.

`call` – transfers control to a code produced by the embraced scenario which may represent activation of external systems (including those working in other formalisms), with resultant position being the same where the rule started, value in it corresponding to what has been returned from the external call, and state `thru` if the call was successful, otherwise `fail`.

`input` – provides input of external information or physical matter (objects) on the initiative of SGL scenario, resulting in the same position but with value received from the outside. The rule may have an additional argument clarifying a particular external source from which the input should take place. The rule extends possibilities provided by reading from environmental variable `IN` explained before.

`output` – outputs the resultant value obtained by the embraced scenario, which can be multiple, with the same resultant position as before but associated value just sent outside (for virtual data only). The rule may have an additional pointer to a particular external sink. The rule extends possibilities provided by assignment to the previously explained environmental variable `OUT`.

`transmit` – represents a variant of `output` for specific applications, say, involving long distance radio communications and broadcasting features, with potentially multiple addresses. It may have additional parameters clarifying the action needed.

`send` – staying in the current position associated with physical, virtual, executive (or combined) node, transfers information or matter obtained by the scenario on the first operand to other similar node given by name, address or coordinates provided by the second operand, assuming that a companion rule `receive` is engaged there. The rule may have an additional parameter setting acceptable time delay for a consumption of this data at the receiving end. If the transaction is successful, the resultant position will be the same where the rule started with state `thru` and value sent (virtual only) otherwise `nil` and state `fail`.

`receive` – a companion to rule `send`, naming the source of data to be received from (defined similarly to the destination node in `send`). Additional timing (as a second operand) may be set up too, after expiration of which the rule will be considered as failed. In case of successful receipt of data, the

rule will result in the same position with the value obtained from `send` and state `thru`, otherwise with `nil` and state `fail`.

7.3.9 Timing

`sleep` – establishes time delay defined by the embraced scenario operand, with no activities in the meantime by this particular scenario branch. The starting position and its existing value will be the result on the rule after the time passed, with state `thru`. Such time delay of the related branch can also be achieved by assigning the current absolute time (received from environmental variable `TIME`), incremented by the delay value returned from the scenario embraced by `sleep`, to environmental variable `WHEN` described before.

`allowed` – sets time limit by the first operand for activity of the scenario on second operand. If the scenario terminates before time limit expires, its resultant positions with values and states will define the result on this rule. Otherwise the scenario will be forcefully aborted with state `fail` in the starting position as the rule's result.

7.3.10 Granting

`contain` – restricts the spread of destructive consequences caused by control state `fatal` within the ruled scenario. This state may appear automatically or can be assigned explicitly to environmental variable `STATE`, triggering emergent completion of all scenario processes and removal of data associated with the scenario. The resultant position will the one the rule started, its value `nil`, and state `fail`. Without occurrence of `fatal`, the resultant positions, their values and states on the rule will be exactly the same as of the scenario embraced.

`release` – allows the embraced scenario develop free from the main scenario, abandoning bilateral control links with it, starting from the current position (the main scenario after the rule's activation "will not see" this construct any more). The released, now independent, scenario will develop using standard subordination and command and control mechanisms, as usual. For the main scenario, this rule will result in its starting position with state `thru` and original value there.

`free` – differs from the previous case in that despite its independence and control freedom from the main scenario, as before, it is nevertheless obliged to return data obtained in its terminal positions if such a request has been issued by rules at higher levels.

`blind` – blocks the embraced scenario from engagement in further development after its completion, but retains the possibility to reply to higher levels with values associated with final positions reached. This being equivalent to setting control state `done` in each terminal position.

`lift` – removes the blocking of further development caused by states `done` in terminal positions of the embraced scenarios (including the effect caused by rule `blind`), substituting them with `thru`, thus allowing further development from these positions by a subsequent scenario.

`none` – sets `nil` (or empty) as a returned value of the whole scenario embraced, with the rule resulting in the same starting position with sate `thru`.

`stay` – whatever the scenario embraced and its evolution in space, the resultant position will always be the same this rule started from, with the latest value in it and state `thru`. As can be seen, this rule differs from the previous one only by its resultant value.

`seize` – establishes, seizes, an absolute control over the resources associated with the current virtual, physical, executive or combined node, blocking these from any other accesses and allowing only the

embraced scenario to work with them, thus preventing possible competition for the node's assets which may lead to unexpected results. This resource blockage is automatically lifted after the embraced scenario terminates. The resultant set of positions on the rule with their values and states will be the ones from the scenario embraced (the latter may potentially be of any complexity and space-time coverage). If the node has already been blocked by another scenario exercising its own rule `seize`, the current scenario will be waiting for the release of the node. If more than two scenarios are competing for the node's resources, they will be organized in a FIFO manner at the node.

7.3.11 Type

These rules explicitly assign types to different constructs generally represented as strings (given explicitly or being the result of an arbitrary operand scenario with single or multiple elements). These rules result in the same positions the rule started, `nil` value and state `thru` (`fail` appears only if a string element does not satisfy certain constrains mentioned below).

`global`, `heritable`, `frontal`, `nodal`, `environmental` – allow different types of variables to have any identifiers (letter and/or digits only) rather than those restricted for self-identification, as explained before. These new names will continue represent the variables with their types in the subsequent scenario development to its full depth unless redefined by these rules. As regards environmental variables, their names differing from the standard ones and new kinds of such variables may need special adjustment with the implementation layer which is directly accessing corresponding physical or virtual resources.

`matter`, `number`, `string`, `scenario` – allow arbitrary strings (with letters, digits and some other characters but not violating the SGL syntax) obtained by the scenario embraced to represent corresponding values rather than using self-identifiable representations mentioned before (with automatic internal types conversion, if needed).

7.3.12 Usage

`address`, `coordinate`, `content`, `index`, `time`, `speed`, `name`, `place`, `center`, `range`, `doer`, `human`, `robot`, `node[s]`, `link[s]` – explicitly clarify the purpose or usage of different values in other rules, adding flexibility to composition of SGL scenarios for which strict order of operands and presence all of them may be optional. The rules result in the same positions they've started with the values clarified by them.

`unit` – identifies the set of values produced by the embraced scenario as an integral unit (like list) for further processing. This may also be useful for hierarchical structuring of data, where elements within declared units may be other units themselves, and so on. The rule results in the same position it started with the value being the unit formed.

7.3.13 Application

Additional application, or custom, rules can allow SGL to be extended unlimitedly while effectively embracing and embedding specifics of different application areas. They can be used similarly to other language rules while obeying established internal interpretation principles and unified command and control. These rules will, however, need extension of and adjustment to the standard language interpretation system.

7.3.14 Aggregated, Grasp

This brings another level of recursion into the language structure where rules can themselves be defined by arbitrary scenarios, or *grasps* (and not only by the explicit names described above), possibly,

aggregated with each other and their modifiers, to operate jointly on the scenarios embraced. Such aggregation can increase and sharpen the power and flexibility of the language and reduce redundancy in complex operations over distributed environments.

8 Full SGL Summary

The following is full SGL formal description summarizing the listed above language constructs, where, as already mentioned, syntactic categories are shown in italics, vertical bar separates alternatives, parts in braces indicate zero or more repetitions with a delimiter at the right if more than one, and constructs in brackets are optional. The remaining characters and words are the language symbols (including boldfaced braces).

grasp	→ constant | variable | rule [({ grasp, })]
constant	→ information | matter | custom | special | grasp
variable	→ global | heritable | frontal | nodal | environmental
rule	→ movement | creation | echoing | verification | assignment | branching | transference | timing | granting | type | usage | application | grasp
information	→ string | scenario | number
string	→ ‘{character}’
scenario	→ **{**{character}**}**
number	→ [sign]{digit}[.{digit}[e[sign]{digit}]]
matter	→ “{character}”
special	→ `thru` | `done` | `fail` | `fatal` | `infinite` | `nil` | `any` | `all` | `other` | `passed` | `existing` | `neighbors` | `direct` | `noback` | `firstcome` | `forward` |`backward` | `global` | `local` | `sync[hronous]` | `async[hronous]` |`virtual` | `physical` | `executive` | `engaged` | `vacant` | `existing` | `passed`
global	→ G{alphameric}
heritable	→ H{alphameric}
frontal	→ F{alphameric}
nodal	→ N{alphameric}
environmental	→ `TYPE` | `CONTENT` | `ADDRESS` | `QUALITIES` | `WHERE` | `BACK` | `PREVIOUS` | `PREDECESSOR` | `DOER` | `RESOURCES` | `LINK` | `DIRECTION` | `WHEN` | `TIME` | `SPEED` | `STATE` | `VALUE` | `COLOR` | `IN` | `OUT` | `STATUS`
movement	→ `hop` | `move` | `shift` | `follow`
creation	→ `create` | `linkup` | `delete` | `unlink`
echoing	→ `state` | `order` | `rake` | `sum` | `count` | `first` | `last` | `min` | `max` | `random` | `average` | `element` | `sortup` | `sortdown` | `reverse` | `add` | `subtract` | `multiply` | `divide` | `degree` | `separate` | `unite` | `attach` | `append` | `common` | `withdraw` | `access`
verification	→ `equal` | `notequal` | `less` | `less[or]equal` | `more` | `bigger` | `smaller` | `heavier` | `lighter` | `longer` | `shorter` | `empty` | `nonempty` | `belongs` | `notbelongs` | `intersects` | `notintersects`
assignment	→ `assign` | `assignpeers`
advancement	→ `advance` | `slide` | `repeat`
branching	→ `branch` | `sequential` | `parallel` | `if` | `or` | `orsequential` | `orparallel` | `and` | `andsequential` | `andparallel` | `choose` | `firstrespond` | `cycle` | `loop` | `sling` | `whirl` | `split` | `fringe`

transference → run | call | input | output | transmit | send | receive
timing → sleep | allowed
granting → contain | release | free | blind | lift | none | stay | seize
type → global | heritable | frontal | nodal | environmental | matter | number | string | scenario
usage → address | coordinate | content | index | time | speed | name | place | center | range | doer | human | soldier | robot | node[s] | link[s] | unit

9 Elementary Examples in SGL

Let us consider some elementary scenarios in SGL from the mentioned three worlds (PW, VW, and EW).

(a) Assignment of the sum of three constants 27, 33, and 55.6 to a variable named Result:

```
assign(Result, add(27, 33, 55.6))
```

(b) Independent moves in physical space to coordinates (x1, y3) and (x5, y8):

```
branch(move(place(x1, y3)), move(place(x5, y8)))
```

(c) Creation of a virtual node Peter:

```
create(direct, node('Peter'))
```

(d) Extending the previous virtual network (so far containing node Peter only) with a new link-node pair father of Alex:

```
advance(hop(direct, node('Peter')),
        create(link(+'fatherof'), node('Alex')))
```

(e) Giving direct order to robot Shooter to fire at certain coordinates (x, y):

```
advance(hop(direct, robot('Shooter')), fire(place(x, y)))
```

(f) Ordering soldier John to engage robot Shooter to fire at coordinates (x, y), with John confirming completion of the robot's action:

```
advance(hop(direct, soldier('John')),
        if(advance(hop(direct, robot('Shooter')),
           fire(place(x, y))), output(OK)))
```

10 Simplifications and Use of Conventional Notations

To simplify SGL programs, traditional to existing programming languages abbreviations of operations, also conventional delimiters can be used too. These can include semicolons for separation of actions following one another in space (i.e. without the rule advance, but not related to its modification slide), just using commas for separating of independent branches (omitting the most general rule branch for such cases), omitting single quotes for strings used as names which do not intersect with the language variables, the use of traditional characters for arithmetic operations and infix notations, skipping identification rules in cases where contents are clear without them, or reduction of the number of parentheses with the help of other characters, like semicolon.

These and similar simplifications should, however, be used with a good deal of caution, especially for complexly structured and nested scenarios, otherwise may distort the scenario structures, also leading to their wrong interpretation. With the presence of such deviations, the scenario text can be readily updated to SGL standards by a preprocessing converter, with subsequent distributed execution by the networked interpreter oriented and optimized on the universal syntax of Figure 2. For the examples of the previous section these simplifications may look like follows.

(a) Assignment of the sum of constants to a variable:

```
Result = 27 + 33 + 55.6
```

(b) Independent moves in physical space to given coordinates:

```
move(x1, y3), move(x5, y8)   or
move((x1, y3), (x5, y8))   or
move(x1_y3, x5_y8)
```

(c) Creation of a virtual node:

```
create(Peter)
```

(d) Extending the virtual network with a new link-node pair:

```
hop(Peter); create(+fatherof, Alex)
```

(e) Giving direct command to a robot to fire:

```
hop(Shooter); fire(x, y)
```

(f) Ordering soldier to engage robot to fire by given coordinates, confirming the action's completion:

```
hop(John); if((hop(Shooter); fire(x, y)), output(OK))
```

11 SGL Networked Interpretation

The developed technology if used in distributed environments operates as follows. A network of SGL interpreters (as universal control modules U, Figure 3) embedded into key system points (humans, robots, sensors, mobile phones, etc.) collectively interprets high-level mission scenarios written in SGL. Capable of representing any parallel and distributed algorithms, these scenarios can start *from any node*, covering at runtime the whole world or its parts needed with operations and control.

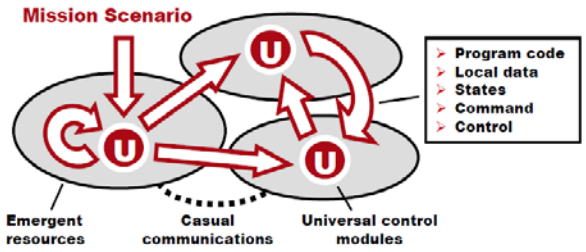

Figure 3: Self-Spreading Spatial Scenarios in SGL

The spreading scenarios can create knowledge infrastructures arbitrarily distributed between system components, as in Figure 4. Navigated by same or other scenarios, these can effectively support distributed databases, command and control (C2), situation awareness and autonomous decisions, also simulate any other existing or hypothetic computational and/or control models.

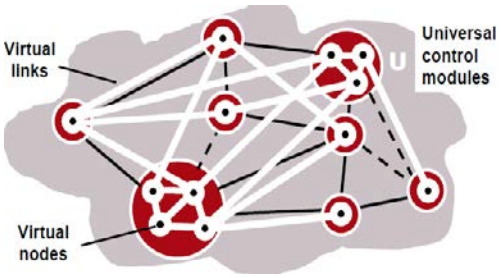

Figure 4: Creation of Spatial Infrastructures

Many SGL scenarios can operate within the same environments, spatially cooperating or competing in the networked space as overlapping fields of solutions, see Figure 5.

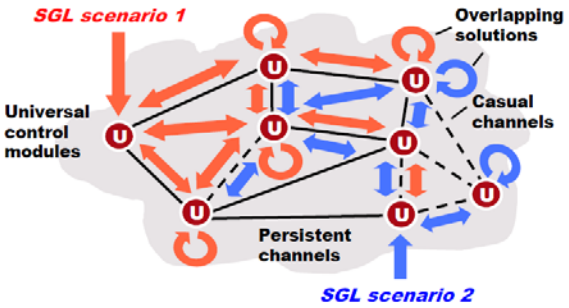

Figure 5: Spatial Interaction of Different Scenarios

The dynamic network of SGL interpreters covering any distributed spaces, the whole world including, can be considered as a new type of parallel supercomputer, which can have any (including runtime changing) networking topology and operate without any central facilities or control. A backbone of the networked interpreter is its *spatial track system* providing global awareness and automatic C2 over multiple distributed processes, also creating, supporting, and managing (including removing when becoming useless) different distributed information and control resources.

12 Some SGT Application Areas

The following are only some researched, discussed, and reported applications of SGT and SGL summarizing their advantages, with other application areas and possible solutions in them described in detail in the existing publications.

Intelligence, Surveillance and Reconnaissance (ISR) [16, 17]. SGT can integrate distributed ISR facilities into flexible goal-driven systems operating under unified command and control, which can be automatic**.** These integrated systems can analyze and properly impact critical infrastructures, both native and adversary's, as well as create new infrastructures for a variety of purposes.

Military robotics [18-21]. SGT paves the way for unified transition to automated up to fully unmanned systems with massive use of advanced robotics. One of practical benefits may be effective management of advanced robotic collectives, regardless of their size and spatial distribution, by only a single human operator, due to high level of their internal self-organization and integral external responsiveness.

Human terrain [22, 23]. SGT allows this new topic, originally coined in military, to be considered and used in a much broader sense and scale than initially planned, allowing us to solve complex national and international conflicts and problems by intelligent and peaceful, predominantly nonmilitary means, while fully obeying existing ethical standards.

Air and missile defense [24, 25]. Providing flexible and self-recovering distributed C2 infrastructures it can, for example, effectively use distributed networks of cheap ground or low-altitude sensors to discover, trace and destroy multiple cruise missiles with complex routes, versus existing expensive high-altitude planes, drones, and aerostats (with an example already shown above). Other examples, also related to ballistic missiles, show the applicability of SGT for the defence against.

Command and Control [26]. Description in SGL of semantic-level military missions is much clearer and more compact (up to 10 times) than if written in traditional Battle Management Languages (BML). This simplicity may allow us redefine the whole scenario or its parts at runtime when goals and environment change rapidly, especially in asymmetric situations and operations, also naturally engage robotic units.

Distributed interactive simulation [27]. The technology can be used for both live control of large dynamic systems and distributed interactive simulation of them (the latter serving as a look-ahead to the former), also any combination thereof, with watershed between the two changing at runtime.

More on investigated and reported SGL applications can be found in other existing publications, [28-34] including.

13 Conclusion

We have described ideology, syntax, basics of semantics, and main constructs of a completely different language, oriented on programming and processing of distributed spaces directly. With the use of it, the whole distributed world, equipped with communicating SGL interpreters, can be considered as an integral and universal spatial machine capable of solving arbitrary complex problems in this world (*machine* rather than *computer* as it can directly operate with physical matter and objects too).

Multiple communicating "processors" or "doers" of this machine, being stationary or mobile, can include humans, computers, robots, smart sensors, any mechanical and electronic equipment capable of cooperatively solving problems formulated in SGL. Being understandable and suitable for both manned and unmanned components, the language offers a real way to unified transition to massively robotized systems, including fully unmanned ones, as within the SGL operational scenarios any component can easily change its manned to unmanned status and vice versa, and at any moment of time.

REFERENCES

[1]. P. S. Sapaty, Mobile Processing in Distributed and Open Environments, John Wiley & Sons, New York, 1999.

[2]. P. S. Sapaty, Ruling Distributed Dynamic Worlds. John Wiley & Sons, New York, 2005.

[3]. P. Sapaty "The World as an Integral Distributed Brain under Spatial Grasp Paradigm", Book chapter in Intelligent Systems for Science and Information, Springer, Feb. 4, 2014.

[4]. P. S. Sapaty, "Meeting the world challenges with advanced system organizations", Book chapter in: Informatics in Control Automation and Robotics, Lecture Notes in Electrical Engineering, Vol. 85, 1st Edition, Springer, 2011.

[5]. P. Sapaty, "Logic flow in active data", Book chapter in: VLSI for Artificial Intelligence and Neural Networks. Springer; Softcover reprint of the original 1st ed. 1991 edition, 2012.

[6]. P. Sapaty, "Distributed technology for global dominance", In R. Suresh (Ed.), Proceedings of SPIE Volume 6981, defense transformation and net-centric systems, 2008.

[7]. M. Wertheimer, Gestalt theory, Erlangen, Berlin, 1924, 380 p.

[8]. P. Sapaty, "Gestalt-Based Ideology and Technology for Spatial Control of Distributed Dynamic Systems", International Gestalt Theory Congress, 16th Scientific Convention of the GTA, University of Osnabrück, Germany, March 26 - 29, 2009.

[9]. P. Sapaty, "Gestalt-based integrity of distributed networked systems", SPIE Europe Security + Defence, bcc Berliner Congress Centre, Berlin Germany, 2009.

[10]. M. Minsky, The Society of Mind, Simon & Schuster, New York, 1988, 336 p.

[11]. P. S. Sapaty, "Over-Operability in Distributed Simulation and Control", The MSIAC's M&S Journal Online. Vol. 4, No 2, 2002, 8 p.

[12]. K. Wilber, “Waves, streams, states and self: Further considerations for an integral theory of consciousness”, Journal of Consciousness Studies 7 (11-12), 2000.

[13]. P. S. Sapaty, "The WAVE Model for advanced knowledge processing", in CAD Accelerators (A.P. Ambler, P. Agrawal & W.R. Moore, Eds.), Elsevier Science Publ. B.V., 1990.

[14]. P. Sapaty, A Distributed Processing System. European Patent No. 0389655, Publ. 10.11.93, European Patent Office.

[15]. P. Sapaty, “Crisis Management with Distributed Processing Technology”, International Transactions on Systems Science and Applications, vol. 1, no. 1, 2006, pp. 81-92.

[16]. P. Sapaty, “Providing Over-operability of Advanced ISR Systems by a High-Level Networking Technology”, SMI’s Airborne ISR, 26th to 27th October 2015, Holiday Inn Kensington Forum, London, United Kingdom.

[17]. P .S. Sapaty, “Integration of ISR with Advanced Command and Control for Critical Mission Applications”, SMi’s ISR conference, Holiday Inn Regents Park, London, 7-8 April 2014.

[18]. P. Sapaty, “Military Robotics: Latest Trends and Spatial Grasp Solutions”, International Journal of Advanced Research in Artificial Intelligence, Vol. 4, No.4, 2015.

[19]. P. S. Sapaty, “Unified Transition to Cooperative Unmanned Systems under Spatial Grasp Paradigm”, International journal Transactions on Networks and Communications (TNC), Vol.2, Issue 2, Apr. 2014.

[20]. P. Sapaty, “High-Level Technology to Manage Distributed Robotized Systems”, Proc. Military Robotics 2010, May 25-27, Jolly St Ermins, London UK.

[21]. P. Sapaty, “From Manned to Smart Unmanned Systems: A Unified Transition”, SMi’s Military Robotics, Holiday Inn Regents Park London, 21-22 May 2014.

[22]. P. Sapaty, “Distributed Human Terrain Operations for Solving National and International Problems”, International Relations and Diplomacy, Vol. 2, No. 9, September 2014.

[23]. P. S. Sapaty, “Solving Social Problems by Distributed Human Terrain Operations”, Journal of Mathematical Machines and Systems (MMC), №3, 2015.

[24]. P. S. Sapaty, “Distributed air & missile defense with spatial grasp technology”, Intelligent Control and Automation, Scientific Research, 3(2), 2012, pp. 117-131.

[25]. P. Sapaty, “Distributed Missile Defence with Spatial Grasp Technology”, SMi’s Military Space, Holiday Inn Regents Park London, 4th-5th March 2015.

[26]. P. Sapaty, “Unified Transition to Cooperative Unmanned Systems under Spatial Grasp Paradigm”, 19th International Command and Control Research and Technology Symposium, June 16-19, 2014, Alexandria, Virginia.

[27]. P. S. Sapaty, M. J. Corbin, and P. M. Borst, "Towards the development of large-scale distributed simulations", Proc. 12th Workshop on Standards for the Interoperability of Distributed Simulations", IST UCF, Orlando, FL, March 1995, pp. 199-212.

[28]. P. S. Sapaty, “Advanced Naval Operations Under Spatial Grasp Technology”, International

Conference Naval Combat Systems, 28 - 29 July, 2015 - Park Plaza Victoria, London, United Kingdom.

[29]. P. Sapaty, "Night Vision under Advanced Spatial Intelligence: A key to Battlefield Dominance", SMi's Night Vision 2013 Conference, London, United Kingdom, 5-6 June 2013.

[30]. P. S. Sapaty, "Providing spatial integrity for distributed unmanned systems", Proc. 6th International Conference in Control, Automation and Robotics ICINCO 2009, Milan, Italy, 2009.

[31]. P. Sapaty, M. Sugisaka, M. J. Delgado-Frias, J. Filipe, N. Mirenkov, "Intelligent management of distributed dynamic sensor networks. Artificial Life and Robotics, 12(1-2), Springer Japan, March 2008, pp. 51-59.

[32]. P. Sapaty, M. Sugisaka, R. Finkelstein, J. Delgado-Frias, N. Mirenkov, "Emergent Societies: An Advanced IT Support of Crisis Relief Missions", Proc. Eleventh International Symposium on Artificial Life and Robotics (AROB 11th'06), Beppu, Japan, Jan 23-26.

[33]. P. Sapaty, "Grasping the Whole by Spatial Intelligence: A Higher Level for Distributed Avionics", Proc. international conference Military Avionics 2008, Jan. 30 - Feb.1, Café Royal, London, UK, 2008.

[34]. P. Sapaty, "Unified transition to cooperative unmanned systems under Spatial Grasp paradigm", International Symposium on Artificial Life and Robotics AROB 19th 2014, January 22-24, 2014. B-Con PLAZA, Beppu, JAPAN.