Journal of Robotics, Networking and Artificial Life, Vol. 3, No. 3 (December 2016), 174-181

Spatial Grasp Language (SGL) for Distributed Management and Control

Peter Simon Sapaty
Institute of Mathematical Machines and Systems
National Academy of Sciences, Kiev, Ukraine
E-mail: peter.sapaty@gmail.com

Abstract

A full description of the 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 spatial patterns. The latter are covering, grasping and matching distributed systems while creating active infrastructures in them operating in a global-goal-driven manner but without any central resources.

Keywords: spatial intelligence, Spatial Grasp Language, self-evolving scenarios, parallel networked interpretation, 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 situations, with the use of asymmetric and hybrid methods and solutions. A new high-level networking approach is being developed [1-6] which can be useful for solving such world problems. It is based on holistic and gestalt ideas allowing us to grasp the whole first [7-9] rather than traditional approaches based on communicating parts or agents [10].

The resultant Spatial Grasp Technology (SGT) with Spatial Grasp Language (SGL) as its key element has been prototyped and tested on numerous applications. In general, it operates as symbolically shown in Fig. 1. 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 final results retained in the environments or returned as a high knowledge to the starting point(s).

The paper describes full specification of the latest version of SGL currently used in a number of projects related to intelligent management and control of large distributed dynamic systems. SGL is the latest and most advanced version in a row of spatial languages using free but globally controlled movement of program code in networks, with previous variants named as WAVE [1], WAVE-WP [2] and DSL [11].

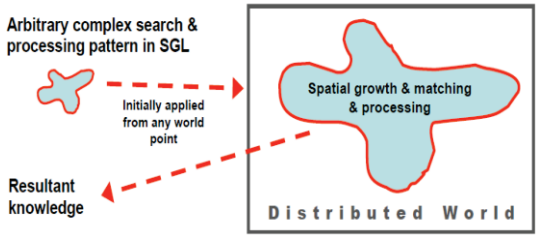


Fig.1. How SGT operates in general.

2. The SGL Worlds

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. SGL directly operates with:

- *Physical World* (PW), continuous and infinite, where each point can be identified and accessed by physical coordinates.
- *Virtual World* (VW), which is discrete and consists of nodes and semantic links between them, both capable of containing arbitrary information.
- *Executive world* (EW), consisting of active doers with communication channels between them, which can represent any devices, humans including, operating on the previous two worlds.

Different combination and integration of these worlds can be possible within the same formalism.

3. Top SGL Syntax

SGL has a recursive structure shown in Fig. 2.

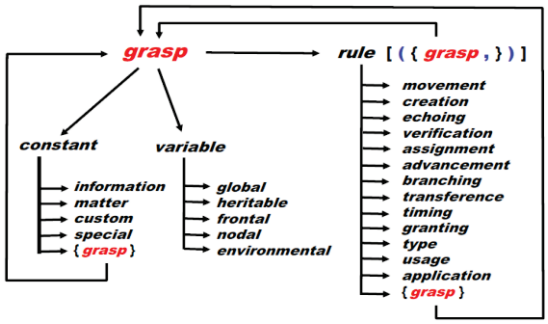


Fig. 2. SGL recursive syntax.

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 problems *in*, *on*, and *over* them.

The SGL topmost definition with scenario named as *grasp* 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.

An SGL scenario, or *grasp* 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. The third variant is 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 (from constants or variables to scenarios of any complexity and space-time coverage).

SGL constants can represent *information*, physical *matter* (or objects), self-identifying *custom* items, *special* words as modifiers for different constructs, or arbitrary structures, as *grasps* again:

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

SGL variables, called "spatial", containing information or matter, can be stationary or mobile an named as:

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 option, *grasp*, allows rules also to be the results of spatial development of any SGL scenarios.

4. SGL Main Features

4.1. *How scenarios evolve*

- SGL scenario is considered developing in *steps*, which can be *parallel*, with new steps produced on the basis of previous steps.
- Any step is always associated with a point or position of the world in which the scenario or its particular part is currently developing.
- Each step provides a resultant *value* (single or multiple) representing information, matter or both, and a resulting control *state* (states ranging by their strength) in the points reached.
- Different scenario parts may evolve from a step in *ordered*, *unordered*, or *parallel* manner, providing new independent or interdependent steps.
- Different parts can *succeed* each other, with new parts evolving from steps got by previous parts.
- This (potentially parallel and distributed) scenario evolution may proceed in *synchronous*, *asynchronous* or *combined* mode.
- SGL operations and decisions in evolving scenario parts can use states and values *returned* from other parts whatever complex and remote they might be, thus combining *forward* and *backward* operations.
- Different steps from the same or different scenario parts can associate with the same world points, sharing information in them.
- Staying with world points, it is possible to *change* local parameters in them, whether physical or virtual, *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. They can also subsequently (or even during

their creation) be navigated, updated and processed by the same or other scenarios.
- Overall organization can be provided by a variety of SGL rules which may be arbitrarily *nested*.

Any sequential or parallel, centralized or distributed, stationary or mobile algorithms operating with both information and physical matter can be written in SGL at any levels and their combinations.

4.2. *Sense and nature of rules*

A rule representing any SGL 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 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.
- A rule can also be a compound one integrating other rules, due to recursion.

All rules, regardless of nature or complexity, pursue 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 which may be stepwise, parallel and arbitrarily complex.
- Produce concluding results by final steps, expressed by control states and values there.
- The final steps may associate with the same (where the rule started) or new world positions.

This uniformity allows us to compose integral spatial algorithms of any complexity and world coverage, operating altogether under unified control.

4.3. *Spatial variables*

The spatial variables are created upon declaration by special rules or by first assignment to them, being as:
- *Global variables* – most expensive, can serve any SGL scenarios and be shared by their different branches. Their existence can depend on features of distributed environments and implementation.
- *Heritable variables* – stationary, serving only the current and all subsequent steps, generally multiple and parallel, which can share them.
- *Frontal variables* – mobile, associating with the current step and accompanying scenario evolution; being transferred between subsequent steps and replicated if a number of new steps emerge.
- *Environmental variables* – stationary or mobile, allowing us to access, analyze, and change different features of worlds navigated.
- *Nodal variables* – a temporary property of world positions, being shared by all activities reaching these nodes under the same scenario identity.

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.

4.4. *Control states and their hierarchical merge*

The following control states appear after performing different scenario steps. Indicating local progress, they are used for distributed control of multiple processes.
- `thru` – reflects *full success* of the current branch of the scenario with capability of further development.
- `done` – indicates success of the current scenario step as *planned termination*, without further development of this branch from the current step.
- `fail` – indicates failure of the current branch, with no possibility of further development from it.
- `fatal` – reports *terminal failure* with nonlocal effect, triggering abortion of all evolving scenario processes together with associated data.

These control states appearing in different branches of a parallel and distributed scenario are used to obtain generalized states at higher levels for proper decisions. The generalization is based on their power, from maximum to minimum: `fatal`, `thru`, `done`, `fail`.

5. Description of Main SGL Constructs

5.1. *Constants*

S*tring* can be represented as any sequence of characters embraced in opening-closing single quotation marks. Instead of single quotes ‘’, a sequence of characters can also be placed into opening-closing braces {}. The latter will indicate the text as a potential *scenario* code to be optimized. *Number* can be represented traditionally like: [*sign*]{*digit*}[.{*digit*}[E[*sign*]{*digit*}]]. Physical *matter* (objects) can be expressed by a sequence of characters embraced in opening-closing double quotes “”. The above self-identified constants can also be set

via explicit naming of their types by corresponding rules.

Special verbal constants can be used as standard parameters (modifiers) in different rules. The basic list of these: `thru`, `done`, `fail`, `fatal`, `infinite`, `nil`, `any`, `all`, `other`, `passed`, `existing`, `neighbors`, `direct`, `noback`, `firstcome`, `forward`, `backward`, `global`, `local`, `synchronous`, `asynchronous`, `virtual`, `physical`, `executive`, `engaged`, `vacant`, `existing`, `passed`.

Constants can also be arbitrarily complex, as aggregates (hierarchical including) – i.e. *grasps* again.

5.2. *Variables*

Different types of variables can be self-identifiable, by the way their names written. Variables of different types can also have any identifiers if explicitly declared by special rules. In case of self-identification they should start with capitals `G`, `H`, `F` or `N`, respectively, for *global*, *heritable*, *frontal*, and *nodal* variables,

Environmental variables have names in all capitals:

- `TYPE` – indicates the type of a node the current step associates with (i.e. `virtual`, `physical`, `executive`, or combination).
- `CONTENT` – returns content/name of the current node (with virtual/executive dimension, or both).
- `ADDRESS` – returns address of the virtual node.
- `QUALITIES` – identifies parameters associated with the current physical position.
- `WHERE` – keeps physical coordinates of the current node. Assigning causes movement to new position.
- `BACK` – keeps internal system link to the preceding world node.
- `PREVIOUS` – refers to a unique address of the previous virtual node.
- `PREDECESSOR` – refers to the content/name of the preceding world node.
- `DOER` – keeps name of the device which interprets current SGL code.
- `RESOURCES` – keeps a list of available or recommended resources for scenario execution.
- `LINK` – keeps a name (same as content) of the virtual link which has just been passed.
- `DIRECTION` – keeps direction (along, against, or neutral) of the passed virtual link.
- `WHEN` – assigning value to this variable sets up an absolute starting time for the following branch.
- `TIME` – returns current absolute time, being read-only global variable.
- `SPEED` – reflects/changes current speed of physical movement of the current node.
- `STATE` – can be used for explicit setting the state of the current step.
- `VALUE` – when accessed, returns resultant value of the latest operation.
- `COLOR` – keeps identity of the current SGL scenario or its branch propagating with it.
- `IN`, `OUT` – special variables for data exchanges reading with the outside world.
- `STATUS` – retrieving or setting the status of a doer node in which the scenario is currently staying.

Other environmental variables for extended applications can be introduced by unique words in all capitals too, or set up by special rules.

5.3. *Rules*

5.3.1. *Movement*

Rules of this class result in virtual hopping to the existing nodes or real movement to new physical locations with current frontal variables and control.

- `hop` – sets virtual propagation to node(s) in virtual, execution, or combined worlds, directly or via links connecting them.
- `move`, `shift` – sets real movement in physical world given by coordinates or their deviations.
- `follow` – allows us to propagate in both virtual and physical spaces by following the set up routes.

5.3.2. *Creation*

This class of rules creates or removes nodes and/or links leading to them during distributed world navigation.

- `create`, `linkup` -- creates new link-node pairs, or only links to existing nodes.
- `delete`, `unlink` -- removes links together with nodes they should lead to, or links only.

5.3.3. *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, the latter being its *terminal* point.

These include: `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`.

5.3.4. *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.

They include: `equal`, `notequal`, `less`, `lessorequal`, `more`, `moreorequal`, `bigger`, `smaller`, `heavier`, `lighter`, `longer`, `shorter`, `empty`, `nonempty`, `belongs`, `notbelongs`, `intersects`, `notintersects`.

5.3.5. *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, being: `assign`, `assignpeers`.

5.3.6. *Advancement*

Rules of this class organize forward or *in depth* advancement in space and time. They can work in synchronous or asynchronous mode.

- `advance` – organizes stepwise advancement in physical, virtual, executive or combined spaces, also in a pure computational space.
- `slide` – works similar to the previous rule unless current scenario fails to succeed, the next scenario will be applied then, and so on.
- `repeat` – invokes the embraced scenario as many times as possible, with each new iteration taking place from final positions of previous invocation.

5.3.7. *Branching*

These rules allow the embraced set of scenario operands develop *in breadth*, each from the same starting position, with the resultant set of positions and order of their appearance depending on a concrete rule

- `branch` – most general variant with independence of scenario operands from each other.
- `sequential` – organizing strictly sequential invocation of all scenario operands.
- `parallel` – fully parallel development of all scenario operands from the same starting position.
- `if` – usually with three scenario operands. If *first* succeeds, the *second* activated, otherwise *third*.
- `or` – allows *only one* operand with the resulting state `thru` or `done` to be registered as successful.
- `orsequential` – launches scenario operands in a sequential manner, as they are written.
- `orparallel` – activates all scenario operands in parallel, choosing first successful in time.
- `and` – activates all scenario operands, demanding all to be successful, regardless of invocation order.
- `andsequential` – activates operands in written order, terminating when first resulting with `fail`.
- `andparallel` – activates all operands in parallel, terminating when first in time fails.
- `choose` – chooses a branch in their sequence *before* its execution, using certain parameters.
- `firstrespond` – selects first branch in time replying termination, regardless of success
- `cycle` – repeatedly invokes embraced scenario from same position until it results with success, *all* positions reached constitute the result.
- `loop` – differs from the previous in that the resultant set of positions include only the set produced by the *last* successful invocation.
- `sling` – invokes repeatedly the embraced scenario until it succeeds, always resulting in the *same* starting position.
- `whirl` – endlessly repeating the embraced scenario from the starting position regardless of its success or failure.
- `split` – performs, if needed, additional static or dynamic partitioning of the embraced scenario. `fringe` – providing the most general variant of splitting for any scenario *after* its execution.

5.3.8. *Transference*

This class of rules organizes different control or data transference activity, accessing local or external procedures and worlds. Includes the following: `run`, `call`, `input`, `output`, `transmit`, `send`, `receive`.

5.3.9. *Timing*

`sleep`, `allowed` – establishes time delay defined by the embraced operand, or sets time limit by the first operand for activity on the second operand.

5.3.10. *Granting*

This class of rules provides different services for advanced management of embraced scenarios, like protecting from fatal events, blocking shared resources or freeing the evolving branches from superior control.

It includes the following: `contain`, `release`, `free` , `blind`, `lift`, `none`, `stay` , `seize`.

5.3.11. *Type and usage*

These rules explicitly assign types to different constructs generally represented as strings, or clarify their usage in different rules, including the following: `global`, `heritable`, `frontal`, `nodal`, `environmental`, `matter`, `number`, `string`, `scenario`, `address`, `coordinate`, `content`, `index`, `time`, `speed`, `name`, `place`, `center`, `range`, `doer`, `human`, `robot`, `node`[`s`], `link`[`s`], `unit`.

5.3.12. *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.

6. Full SGL Summary

The following is full SGL formal description summarizing the listed above language constructs.

grasp → *constant* | *variable* | *rule* [({ *grasp* , })]
constant → *information* | *matter* | *custom* | *special* | *grasp*
variable → *global* | *heritable* | *frontal* | *nodal* | *environmental*
rule → *movement* | *creation* | *echoing* | *verification* | *assignment* | *advancement* | *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` | `more`[`or`]`equal` | `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`

7. Elementary Examples in SGL

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

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

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

- Independent moves in physical space to coordinates (x1, y3) and (x5, y8): `move(x1, y3), move(x5,y8)`
- Creation of a virtual node Peter: `create(Peter)`
- Extending the previous virtual network (so far containing node Peter only) with a new link-node pair for Peter as father of Alex:

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

- Giving direct order to robot Shooter to fire at certain coordinates (x, y): `hop(Shooter); fire(x, y)`
- Ordering John to engage robot Shooter to fire at (x, y), with John confirming the action: `hop(John);`

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

8. SGL Networked Interpretation

The developed technology if used in distributed environments operates as follows. A network of SGL

interpreters (as universal control modules U, Fig. 3) embedded in key system points (humans, robots, sensors, mobile phones, etc.) collectively interprets 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.

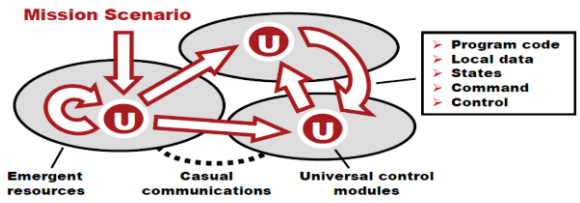
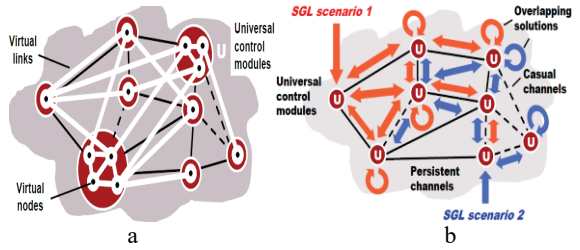


Fig. 3. Self-spreading spatial scenarios in SGL.

The spreading scenarios can create knowledge infrastructures arbitrarily distributed between system components, as in Fig. 4*a*.



a b

Fig. 4. Scenario activity results: a) creation of spatial infrastructures; b) distributed scenario interaction.

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. Many SGL scenarios can operate within the same environments, spatially cooperating or competing in the networked space as overlapping fields of solutions, see Fig. 4*b*.

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

9. Some SGT Application Areas

The following are some researched, discussed, and reported applications of SGT and SGL summarizing their advantages.

- *Graph and network theory* [1, 2]. Highly parallel and fully distributed solutions of main graph and network problems have been programmed in earlier versions of SGL and demonstrated in distributed networks.
- *Distributed interactive simulation* [12, 13]. The technology had been researched for both live control of large dynamic systems like battlefields and distributed interactive simulation of them, any combination too.
- *Solving social problems* [14, 15]. SGL appeared to be useful in social systems area, like support of elderly and handicapped people in modern societies, from intelligent homes improving their everyday life to finding and tracing (if lost, for example).
- *Human terrain* [16]. SGT allows this new topic, originally coined in military, to be used in a broader sense and scale than initially planned, allowing us to solve national and international problems peacefully.
- *Sensor networks* [17]. Multiple sensors scattered over large territories can behave altogether under SGT as a spatial supercomputer operating efficiently even under restricted communications.
- *Military robotics* [18]. SGT paves the way for unified transition up to fully unmanned systems with massive use of advanced robotics. One of practical benefits may be effective management of robotic collectives, regardless of their size by only a single human operator.
- *Intelligence, Surveillance and Reconnaissance (ISR)* [19]. SGT can integrate distributed ISR facilities into flexible goal-driven systems operating under unified command and control, which can be automatic**.**
- *Air and missile defense* [20]. Providing flexible and self-recovering distributed C2 infrastructures, SGT can, for example, effectively use distributed networks of cheap ground or low-altitude sensors to discover, trace, and destroy low flying objects with complex routes by mobile intelligence.
- *Crisis management* [21, 22]. SGT can support advanced distributed systems for crisis management, where complex relief missions, national and international, can be organized and programmed quickly, on the fly.

10. Conclusions

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.

Acknowledgment

The author appreciates the invaluable support for years and personal involvement in this project of Prof. Masanori Sugisaka, especially in the fields of advanced cooperative robotics, IT-based solutions of social problems, distributed sensor networks, and crisis and emergency management.

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.
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. Sapaty, "Crisis Management with Distributed Processing Technology", *International Transactions on Systems Science and Applications*, vol. 1, no. 1, 2006, pp. 81-92.
12. P. Sapaty, M. J. Corbin, S. Seidensticker, "Mobile Intelligence in Distributed Simulations", *Proc. 14th Workshop on Standards for the Interoperability of Distributed Simulations*, IST UCF, Orlando, FL, 1995.
13. 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.
14. P. S. Sapaty, "Solving Social Problems by Distributed Human Terrain Operations", *Journal of Mathematical Machines and Systems* (MMC), №3, 2015.
15. P. P. Sapaty, M. Sugisaka, "Advanced Networking and Robotics for Societal Engagement and Support of Elders", *Proc. 16th International Symposium on Artificial Life and Robotics* (AROB 16th '11), B-Con Plaza, Beppu, Oita, Japan, January 27-29, 2011.
16. P. Sapaty, "Distributed Human Terrain Operations for Solving National and International Problems", *International Relations and Diplomacy*, Vol. 2, No. 9, September 2014.
17. 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.
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. Sapaty, "Providing Over-operability of Advanced ISR Systems by a High-Level Networking Technology", *SMI's Airborne* ISR, 26-27 October 2015, Holiday Inn Kensington Forum, London, United Kingdom.
20. P. Sapaty, "Distributed Missile Defence with Spatial Grasp Technology", *SMi's Military Space*, Holiday Inn Regents Park London, 4-5 March 2015.
21. P. Sapaty, M. Sugisaka, R. Finkelstein, J. Delgado-Frias, N. Mirenkov, "Advanced IT Support of Crisis Relief Missions", *Journal of Emergency Managemen*t, Vol.4, No.4, July/August 2006, pp.29-36.
22. P. Sapaty, M. Sugisaka, R. Finkelstein, J. Delgado-Frias, N. Mirenkov, "Emergent Societies: An Advanced IT Support of Crisis Relief Missions", *Artificial Life and Robotics*, Vol. 11, No. 1, 2007.