APPENDIX: SPATIAL GRASP LANGUAGE DETAILS

This text describes the following.

- A1: *Full SGL Syntax*.
- A2: *SGL Constants*. Including: Information, Physical matter, Special Constants, Custom Constants, Compound Constants
- A3: *SGL Variables*. Including: Global Variables, Heritable Variables, Frontal Variables, Nodal Variables, Environmental Variables
- A4: *SGL Rules*. Including: Type, Usage, Movement, Creation, Echoing, Verification, Assignment, Advancement, Branching, Transference, Exchange, Timing, Qualification, Grasping

A1. FULL SGL SYNTAX

Syntactic categories are in italics, vertical bar separates alternatives, parts in braces indicate zero or more repetitions with a delimiter at the right, 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* | *special* | *custom* | *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 | allother | current | passed | existing | neighbors | direct | forward | backward | synchronous | asynchronous | virtual | physical | executive | engaged | vacant | firstcome | unique
variable → *global* | *heritable* | *frontal* | *nodal* | *environmental*
global → G{*alphameric*}

heritable → H{*alphameric*}
frontal → F{*alphameric*}
nodal → N{*alphameric*}
environmental → TYPE | NAME | CONTENT | ADDRESS | QUALITIES | WHERE | BACK | PREVIOUS | PREDECESSOR | DOER | RESOURCES | LINK | DIRECTION | WHEN | TIME | STATE | VALUE | IDENTITY | IN | OUT | STATUS
rule → *type* | *usage* | *movement* | *creation* | *echoing* | *verification* | *assignment* | *advancement* | *branching* | *transference* | *exchange* | *timing* | *qualifying* | *grasp*
type → global | heritable | frontal | nodal | environmental | matter | number | string | scenario | constant | custom
usage → address | coordinate | content | index | time | speed | name | place | center | range | doer | node | link | unit
movement → hop | hopfirst | hopforth | move | shift | follow
creation → create | linkup | delete | unlink
echoing → state | rake | order | unit | unique | sum | count | first | last | min | max | random | average | sortup | sortdown | reverse | element | position | fromto | add | subtract | multiply | divide | degree | separate | unite | attach | append | common | withdraw | increment | decrement | access | invert | apply | location | distance
verification → equal | nonequal | less | lessorequal | more | moreorequal | bigger | smaller | heavier | lighter | longer | shorter | empty | nonempty | belong | notbelong | intersect | notintersect | yes | no
assignment → assign | assignpeers
advancement → advance | slide | repeat | align | fringe
branching → branch | sequence | parallel | if | or | and | or_sequence | or_parallel | and_sequence | and_parallel | choose | quickest | cycle | loop | sling | whirl | split | replicate
transference → run | call
exchange → input | output | send | receive | emit | get
timing → sleep | allowed
qualification → contain | release | trackless | free | blind | quit | abort | stay | lift | seize | exit

A2. CONSTANTS

Constants can be self-identifiable by the way they are written, as follows in this section, or defined by special rules embracing them for arbitrary textual representations, as shown later.

A2.1 Information

String
A string can be represented in most general way as a sequence of characters embraced by opening–closing single quotation marks:

'{*character*}'

This sequence should not contain other single quotes inside unless they appear in opening–closing pairs, with such nesting allowed to any depth. If single words representing information are not intersecting with other language constructs, the quotes around them can be omitted.

Scenario
Another string representation may be in the form of explicit SGL scenario body:

{{*character*}}

For this case a sequence of characters should be placed into opening–closing curly brackets, or braces {}, shown here in bold (to distinguish from braces used for textual repetition in the language syntax), which can be used inside the string and nested in pairs too. Braces will indicate the text as a potential *scenario* code which should be optimized *before* its any usage. This may involve removing unnecessary spaces, substituting names of rules by their allowed shortcuts, or adjusting to the standard SGL syntax after using constructs typical to other programming languages for convenience (like semicolons as separators instead of sequencing rules). If single quotes embrace SGL texts to be used as an executable scenario, such code optimization will have to be done each time *during* its interpretation, not before, with the original text remaining intact.

Number
Number can be represented in a standard way, similar to traditional programming languages, generally in the following form (with brackets identifying optional parts, and braces the repeating characters).

[*sign*]{*digit*}[.{*digit*}[E[*sign*]{*digit*}]]

Numbers can also use words instead of digits and accompanying characters like sign and dot (with underscore as separator if more then one word needed to represent them).

A2.2 Physical Matter

Physical *matter* (including physical objects) in the most general way can be reflected in SGL by a sequence of characters embraced by opening–closing double quotation marks:

"{*character*}"

A2.3 Special Constants

thru – indicates (or artificially sets up) control state of the scenario in the current world point as an *absolute success* with possibility of further scenario evolution from this particular point. This does not influence scenario developments in other world points.

done – indicates (or sets artificially up) control state as a *successful termination* in the current world point, with blocking further scenario development from this particular point. Not influencing scenario developments in other world points.

fail – indicates (or artificially sets up) scenario control state as *failure* in the current world point, without possibility of further scenario development from this particular point. Not influencing scenario developments in other world points.

fatal – indicates (or sets up artificially) control state as nonlocal *fatal failure* starting in the current world point and causing massive termination and removal of all active distributed processes with related local data in this and other world points reached by the same scenario (which may have parallel branches). The destructive influence of this state may be contained at higher levels by special rules explained later.

infinite – indicates infinitely large value.

nil – indicates no value at all.

any, all, other, allother – stating that *any one*, *all* (the current one including), *any other*, or *all other* (the current one excluding in the last two cases) elements can be considered and used by some rule.

current – refers to the current element (like node) only, for its further consideration or reentering (possibly, with proper conditions).

passed – informing that the mentioned elements (like world nodes) have already been passed by the current scenario branch on its way to the current point. These elements may in some cases be accessed easier by backwarding via the SGL history-based distributed control than by a global search just by name.

existing – hinting that world nodes with given names which are currently under consideration already exist 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., located within a single hop from it via the existing links.

direct – stating that the mentioned nodes should be accessed or created from the current node directly, without consideration of possible semantic links to them (even if such links already exist, as in the case of accessing neighboring nodes).

forward, backward, neutral – allowing us to move from the current node via existing links along, against, or regardless their orientations (ignored when dealing with nonoriented links, which can always be traversed in both directions).

synchronous, asynchronous – 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, having more than one such indicator, with maximum three).

engaged, vacant – indicating or setting the state of a resource the current scenario is dealing with (like, human, robot or any other physical, virtual, or combined world node).

firstcome – allows the current scenario with its unique identity to enter the world nodes only first time (the capability based on internal language node marking mechanisms, which can be used for different purposes, including blocking of unplanned or unwanted cycling).

unique – allows the return of only unique elements received from the embraced scenario final positions while omitting duplicates (the returned results should form a list without repeating elements).

Any other parameters or modifiers can be used in SGL scenarios too, say, as general constants (strings or numbers) if their repertoire and meanings are not covered by the aforementioned examples.

A2.4 Custom Constants

Other self-identifiable, or *custom*, constants can be incorporated to be directly processed by updated SGL, if do not conflict with the language syntax, otherwise should be declared by special rules. They can represent standing without quotes or braces information and physical matter substances or objects, as well as their combinations.

A2.5 Compound Constants

Constants can also be compound ones, using the recursive *grasp* definition in SGL syntax, which allows us to represent nested hierarchical structures consisting of multiple (elementary or compound again) objects. This, in particular for constants, can be expressed as:

rule({*constant*,})

Different SGL rules explained later may be used for such structuring, with more to be added for particular applications.

A3. VARIABLES

A3.1 Global Variables

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

G{*alphameric*}

These variables can exist only in single copies with particular names, being common for both read and write operations to all processes of the same scenario, regardless of their physical or virtual distribution and world points they may cover. Global variables can be created by first assignment to them within any scenario branch and used afterward by the entire scenario, including all its branches. They cease to exist only when removed explicitly or the whole scenario that created them terminates.

A3.2 Heritable Variables

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

H{*alphameric*}

Heritable variables, being created by first assignment to them at some scenario development stage, are becoming common for read–write operations *for all subsequent* scenario operations (generally multiple, parallel, and distributed) evolving from this particular point and wherever is space they may happen to be. This means that such variables are unique only within concrete hereditary scenario developments, to all their depth. The lifetime of these variables depends on the continuing activity of processes that can potentially inherit them, with their removal made explicitly or after all such processes

terminate. Heritable variables can also model global variables if declared at the very beginning of the scenario starting from a single point, as all scenario developments can be using and sharing them afterward.

A3.3 Frontal Variables

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

F{*alphameric*}

Each of these variables is serving only the current scenario branch operating in the current world point. They cannot be shared with other branches evolving in the same or other world points, while always accompanying the scenario control. If the scenario splits into individual branches in the same world point or when moving to other points, these variables are replicated with the same names and contents and serve these branches independently. There may be different variants of working with frontal variables holding physical matter or objects rather than information, especially when serving physical movement and possibility of replication or reproduction of their contents in distributed environments.

A3.4 Nodal Variables

Variables of this type, their identifiers starting with capital N, are a temporary and exclusive property of the world points visited by SGL scenarios, which can create, change, or remove them.

N{*alphameric*}

Capable of being shared by all scenario branches visiting these nodes, they are created by first assignment to them and stay in the node until removed explicitly or the whole scenario remains active. These variables also cease to exist when nodes they associate with are removed by any scenario reaching them.

A3.5 Environmental Variables

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

TYPE – indicates the type of a node the current scenario step associates with and returns a verbal expression of the node's type (i.e., virtual, physical, executive, or their combination). It can also change the existing node's type by assigning to it another value (simple or combined).

NAME – returns name of the current node as a string of characters (only if the node has virtual or executive dimension or both). Assigning to this variable when staying in the node can change the node's name.

CONTENT – returns content of the current node (if it has virtual or executive dimension or both) as arbitrary constant (say, any text in quotes, vector or nested structure of multiple texts, etc.) if this content had been assigned to this node previously, when staying in it. Assigning to this variable when staying in the node can change the node's content. In the case of executive nodes (like human, robot, server, etc.), CONTENT may return and change, if allowed, some existing specific data like dossier on a human or technical characteristics of a robot.

ADDRESS – returns a unique address of the current virtual node (or the one having virtual dimension). This is read-only variable as node addresses are set up automatically by the underlying distributed SGL interpretation system during node's creation, or by an external system (for example, like internet address of the node). The returned address can be remembered and used afterward for direct hops to this node form any positions of the distributed virtual world, if such hops allowed by implementation.

QUALITIES – identifies a list of selected formalized 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 attempted to 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.

WHERE – keeps world coordinates of the current physical node (or the one having physical dimension too) in the chosen coordinate system. These coordinates can be obtained by reading this variable. Assigning a new value to this variable (with possible speed added) can cause physical movement into the new position (with preserving node's identity, virtual and/or executive features if any, all its 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) allowing the scenario to most efficiently return to the previously occupied node, if needed. This variable refers to internal interpretation mechanisms only (its content cannot be lifted, recorded, or

changed from the scenario level), and can be used in direct hop operations only.

PREVIOUS – refers to the absolute and unique address of the previous virtual node (or combined one with executive and/or physical dimensions), allowing us to return to the node directly. This return may be on a higher level and therefore more expensive than using BACK, but the content of PREVIOUS, unlike BACK, can be lifted, recorded, and used elsewhere in the scenario (but not changed, similar to ADDRESS).

PREDECESSOR – refers to the name of preceding world node (the one with virtual or executive dimension, visited just before the current one). Its content can be lifted, recorded, and subsequently used, for organization of direct hops to this node too (on highest and most expensive level, however). Assigning to PREDECESSOR in the current node can change the name of the previous node.

DOER – keeps the name of the device (say, laptop, robot, satellite, smart sensor, or a specially equipped human) which interprets the current SGL code in the current world position. This device can be initially chosen for the scenario automatically from the list of recommended devices or just picked up from those expected available. It can also be appointed explicitly by assigning its name to DOER, causing the remaining SGL code (along with its current information surrounding) to move immediately into this device and execute there. (The change of the device can also be done automatically by the distributed SGL interpreter, say, depending on unpredictable circumstances or by dynamic space-conquering optimization.)

RESOURCES – may keep a list of available or recommended resources (human, robotic, electronic, mechanical, etc., by their types or names) which can be used for planning and execution of the current and subsequent parts of the SGL scenario. This list can also contain potential doers which may appear (by their names) in variables DOER. The contents of RESOURCES can be changed by assignment to it, and in case of automatic distributed SGL interpretation and spatial branching may be replicated or partitioned (or both).

LINK – keeps the name (same as content) of the virtual link which has just been passed. Assigning a new value to it 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 the link nonoriented.

WHEN – assigning value to this variable sets up an absolute starting time for the following scenario branch (i.e., starting with the next operation), thus allowing us to suspend and schedule operations and their groups in time.

TIME – returns the current absolute system time as the read-only global variable.

STATE – can be used for explicit setting resultant control state of the current scenario step by assigning to it one of the following constants: thru, done, fail, or fatal, which will influence further scenario development from the current world point (and in a broader scale in the case of fatal). These control states are also generated implicitly and automatically on the results of success or failure of different operations (belonging to the internal interpretation mechanisms of SGL scenarios). Reading STATE will always return thru as this could be possible only if the previous operation terminated with thru too, thus letting this operation to proceed. A certain state explicitly set up in this variable can also be used at higher levels (possibly, together with termination states of other branches) within distributed control provided by nested SGL rules, whereas assigning fatal to STATE may cause abortion of multiple distributed processes with associated data.

VALUE – when accessed, returns the resultant value of the latest, i.e., preceding, operation (say, an assignment to it or any other variable, unassigned result of arithmetic or string operation, or just naming a variable or constant). Such explicit or implicit assignment to VALUE always leaves its content available to the next operation, which may happen to be convenient by combining different operations traditionally grouped in expressions, within their sequences.

IDENTITY – keeps identity, or color, 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 reached. This allows different scenarios or their branches with personal identities to be protected from influencing each other, even if they are using same named nodal variables in the same world nodes. However, scenarios with different identities can penetrate into each other information fields if they know the other's colors, by temporarily assigning the needed new color to IDENTITY at proper stages and world points (say, to perform cooperative or stealth operations) while restoring the previous color afterward, if needed. Any numerical or string value can be explicitly assigned to IDENTITY. By default, different scenarios may be keeping the same value in IDENTITY assigned automatically at the start (which may be any, including empty), thus being capable of sharing all information at navigated nodes, unless they change their personal color themselves.

IN – special variable requesting and reading data from the outside world in its current point. The received data are becoming the resultant value of the reading operation.

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

STATUS – retrieving or setting the status of (especially doer) node in which the scenario is currently staying (like engaged or vacant, possibly, with a numerical estimate of the level of engagement or vacancy). This feedback from implementation layer on the SGL scenario layer can be useful for a higher-level supervision, planning and distribution of resources executing the scenario rather than doing this implicitly and automatically.

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 defined by using special rule, as shown later. As can be seen, most environmental variables are behaving as stationary ones, except RESOURCES and IDENTITY, which are mobile in nature. The global variable TIME may be considered as stationary too, but can also be implemented in the form of individual TIME clocks regularly updating their system time copies and propagating with scenarios as frontal variables.

A4. RULES

A4.1 Type

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

global, heritable, frontal, nodal, environmental – allow different types of variables to have any alphanumeric names rather than those oriented on self-identification, as explained before. These names will represent variables with needed types in the subsequent scenario developments unless redefined by these rules too.

matter, number, string, scenario, constant – allow arbitrary results obtained by the embraced scenario to properly represent the needed values rather than using self-identifiable representations mentioned before.

A4.2 Usage

These rules explain how to use the information units they embrace, with main variants as follows:

They are adding certain flexibility to representation of SGL scenarios where strict order of operands in different rules and also presence of them all may not be absolute.

address – identifies the embraced value (which may also be an arbitrary scenario producing this value or values if multiple) as an address of a virtual node.

coordinate – identifies the embraced value as physical coordinates (say, one, two, or three dimensional), which may also be a list of coordinates of a number of physical locations.

content – identifies the embraced operand as a content (or contents) which may, for example, relate to certain values in a list for its search by contents.

index – identifies the embraced operand as an index (or indices) which may represent orders of elements in a list for its search by the index operation.

time – informs that the embraced operand represents time value.

speed – informs that the embraced operand represents a value of speed.

name – identifies the embraced operand as a name (say, of a virtual or executive node or nodes).

center – depending on applications, indicates that virtual address or physical coordinates embraced may relate to the center of some region.

range – identifies virtual or physical distance that can, for example, be used as a threshold for certain operations in distributed spaces, especially those evolving from a chosen or expected center.

doer – identifies the embraced name or any other value as belonging to executive node (like human, robot, server, satellite, smart phone, etc.).

node – (or nodes, if more appropriate) identifies the embraced value or values as keeping names of nodes having virtual or/and executive dimensions.

link – (or links, if more appropriate) informs that the embraced value or values represent names of links connecting nodes with virtual or/and executive dimensions.

A4.3 Movement

They may result in virtual hopping to the existing nodes (the ones having virtual or/and executive dimensions) or in real movement to new physical locations, subsequently starting the remaining scenario if any (with current frontal variables and control) in the nodes reached. The resultant values of such movements are represented by names of reached nodes (in case of virtual, executive, or combined nodes) or nil in case of purely physical nodes, with control state thru in them if the movement was successful. If no destinations have been reached, the movement results with state fail and value nil in the rule's starting node. These rules have the following options.

hop – sets electronic propagation to a node (or nodes) in virtual, execution, or combined spaces (the latter may have physical dimension too), directly or via semantic links connecting them with the starting node. In case of a direct

hop, except destination node name or address, special modifier direct may be included into parameters of the rule. If the hop is to take place from a node to a particular node via existing link, both destination node name/address and link name (with orientation if appropriate) should be parameters of the rule. This 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 same named links, 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 rule's parameters are given by a list of possible names/addresses of destination nodes and a list of names of links which may lead to them (direct and/or all indicators can be used here too). The hop rule may have additional modifiers setting certain conditions for this operation, like first-come, which is based on internal language interpretation mechanism properly marking the visited nodes (for example, to be used for blocking unexpected cycles in network propagations). Another modifiers may link the virtual propagation with some physical parameters of possible combined destination nodes, say, by giving threshold distances to them from the current node (if with physical parameters too).

hopfirst – modification of the hop rule allowing it to come to a node only first time (for the scenario with certain identity), which is based on internal interpretation mechanism marking the nodes visited. The use of this rule can be similar to the previous rule hop with modifier firstcome (which can also be used in other cases, like new linking to the existing nodes, as mentioned later).

hopforth – modification of the previous rule allowing it to hop to a node which is not the one just visited before, i.e., excluding the return to the previous node. It may be considered as a restricted variant of hopfirst rule. Both rules can be useful for effective blocking of looping in networked structures for certain scenarios.

move – sets real movement in physical world from the current node with physical dimension (which may be combined with virtual and executive ones) to a particular location given by coordinates in a chosen coordinate system. The destination location becomes a new temporary node with no (nil) name, which disappears when the current scenario activities leave it for other nodes. The location reached may, however, become a persistent or even permanent node if virtual dimension also assigned to it (possibly, virtual name too), after which such combined node can become visible from outside, may keep individual nodal variables, and can be entered and shared by other scenario branches. Speed value for the physical propagation by move may be given as an additional parameter.

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 their absolute values.

follow – allows us to move in virtual, physical, and combined spaces using already (internally, by distributed SGL interpretation) recorded and saved paths from a starting node to the destinations reached, to enter the latter again from the starting node in a simplified way, as will be explained later.

A4.4 Creation

They create or remove nodes and/or links leading to them during distributed world navigation. After termination of the creation rules, their resultant values will correspond to the names of reached nodes with termination states thru in them, and the next scenario steps if any will start from all these nodes. After removal of the destination nodes and/or links leading to them, the resultant world position will be the rule's starting node with the same value as before and control state thru. If the creation or removal operation fails, its resultant value will be nil and control state fail in the node the rule started, thus blocking any further scenario development from this node.

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. If to use modifiers existing or passed for the link–node creation hinting that such nodes already exist or, moreover, have already been passed by this scenario, only links will be created to them by create. Same will take place if nodes are given by their addresses, the latter always indicating their existence. The already mentioned modifier firstcome, if used, will not allow entering the nodes more than once by the same colored scenario.

linkup – restricts the previous rule by creating only links with proper names from the current node to the already existing nodes given by their names or addresses. Using modifier passed, if appropriate, may help us to narrow the search of already existing nodes. Also, the modifier firstcome, if used, will not allow entering the nodes more than once by same colored scenario or its branch, thus blocking linkup operation for this case.

delete – starting from the current node, removes links together with nodes they should lead to. 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 the node name will allow us to remove such node (or nodes) from the current node directly. In all cases, when a node is deleted, its all 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 aforementioned creation rules, depending on implementations, can also be used in a broader sense and scale, as *contexts* embracing arbitrary scenarios and influencing hop operations within their scope. This means that the same scenarios will be capable of operating in the creation and deletion modes too, and not only for navigating the existing networks. These contexts can influence both links and nodes when dealing with the existing networks (or just with empty spaces in which such networks should be created from scratch).

A4.5 Echoing

This class of rules, oriented on various aspects of data and knowledge processing, contains the following rules which may use local and remote values for different operations:

The listed rules use terminal world positions reached by the embraced scenario with their control states and associated final values (which may be local or arbitrarily remote) to obtain the resultant state and value *in the location where the rule started*. This location will represent the rule's single terminal point from which the rest of the scenario, if any, can develop further. The usual resultant control state for these rules is thru (state fail occurs only if certain terminal values happen to be unavailable or the result is unachievable, say, like division by zero). Depending on the rule's semantics, the resultant value may happen to be compound like a list of values, which may also be hierarchically nested.

The semantics of different echoing rules is as follows.

state – returns the resultant generalized state of the embraced SGL scenario upon its completion, whatever its complexity and space coverage may be. This state being the result of ascending fringe-to-root generalization of terminal states of the scenario embraced, where states with higher power (their sequence from maximum to minimum values as: fatal, thru, done, fail) dominate in this potentially distributed and parallel process. The resultant state returned is treated as the *resultant value* on the rule, the latter always terminating with own final control state thru, even in the case of resultant fatal (thus blocking the spreading destructive influence of fatal at the rule's starting point).

rake – returns a list of final values of the scenario embraced in an arbitrary order, which may, for example, be influenced by the order of completion of branches and times of reaching their final destinations. Additionally using

unique as modifier, described before, the rule will result in collecting only unique values, i.e., with possible duplicates omitted/removed.

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 potentially parallel branches, these orders may, for example, relate to how they were activated, possibly, with the use of time stamps upon invocation. Similar to the previous rule, modifier unique can be used too for avoiding duplicate values.

unit – returns a list of values while arranging it as an integral parenthesized unit which should not be mixed with elements returned from other branches which may represent integral units too, to form (potentially hierarchical and nested) lists of lists of the obtained values at higher levels. This rule can be combined with rules rake or order to explicitly set up the expected order of returned values in the unit formed. Without unit, at any scenario level, the returned values from different subordinate branches will represent same level mixture of all obtained results.

sum – returns the sum of all final values of the scenario embraced (modifier unique can be used here for summing only unique final values).

count – returns the number of all resultant values associated with the scenario embraced, rather than the values themselves as by the previous rules (modifier unique can be used too for counting only unique values).

first, last, min, max, random, average – return, correspondingly, the first, last, minimum, maximum, randomly chosen, or average value from all terminal values returned by the scenario embraced. The rules first and last may also need initial ordering of the returned results by previously using or integrating with rule order discussed before, also sortup and sortdown explained below. The modifier unique can be used for all mentioned rules too.

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

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

element – returns the value of an element of the list on its left operand requested by its index (on default) or content (clarifying this by rule content) given by the right operand. If the right operand is itself a list of indices or contents, the result will be a list of corresponding values from the left operand. If element is used within the left operand of assignment, instead of returning values it will be providing an access to them, in order to be updated, as explained later. Each given index representing unique order can return from the left operand one or none value (the latter if the index exceeds total number of elements), whereas each content in the right operand can return from the

left operand none (or nil), one, or more elements as a list, as there can be repeating values at the left.

position – returns the index (or indices) of the list on its left operand requested by the content given by the right operand. There may be more than a single index returned if same content repeats in the list, or if the right operand is itself a list of contents, then each one will participate in the search. The total absence of searched contents in the left operand will result innilvalue on this rule.

fromto – returns an ordered list of digital values by naming its first (operand 1) and last (operand 2) elements as well as step value (operand 3) allowing the next element to be obtained from the previous one. Another modification (depending on implementation) may take into account the starting element, step value, and the number of needed elements in the list.

add, subtract, multiply, divide, degree – perform corresponding operations on two or more operands embraced, each potentially represented by arbitrary scenario with local or remote results. If the operands themselves provide multiple values, as lists, these operations are performed between peer elements of these lists, with the resultant value being multiple, as a list too.

separate – separates the left operand string value by the string at the right operand used as a delimiter (in case to be present at the left) in a repeated manner for the left string, with the result being the list of separated substring values. If the right operand is a list of delimiters, its elements will be used sequentially, one after the other, and cyclically unless the string at the left is fully processed/partitioned. If the left operand represents a list of strings, each one is processed by the right operand as above, with the resultant lists of separated values merging into a common list in the order they were produced.

unite – integrates the list of values at the left (as strings, or to be converted into strings automatically) by a repeated delimiter as a string too (or a cyclically used list of them) at the right into a united string.

attach – produces the resultant string by connecting the right string operand directly 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 by appending the latter to the end of the former as individual elements, where both operands may be lists themselves. More than two operands can be used too for this operation.

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

withdraw – its returned result will be the first element of the list provided by the embraced operand, which can usually be a variable, along with withdrawing this element from the head of the list (thus simultaneously changing the content of the variable). This rule can have another operand providing the number of elements to be withdrawn in one step and represented as the result. When the embraced list is empty or has fewer elements than needed to be withdrawn, the rule returns nil value and terminates with fail state.

increment – adds 1 (one) to the value of the embraced operand which will be the result on this rule, thus simultaneously changing the content of the operand itself (this making sense only if it is a variable, which will be having now the increased value). If another value, not 1, to be added, the second operand can be employed for keeping this value.

decrement – behaves similar to the previous rule increment but subtracts rather than adds 1 from the value of the embraced operand, with the content of the latter simultaneously changed too. Second operand can be used too if the value to be subtracted not equals 1. In all cases if the decrementing result appears to be less than zero, the rule will terminate with fail and value nil.

access – by embracing a scenario or its branch, returns a reference to the internal history-based optimized and recorded structure (which may be spatially distributed) leading from the rule-activation node to the reached terminal nodes on the considered scenario. This reference can be remembered (say, in a variable) and subsequently used from the same starting node to reach exactly the same terminal nodes again in an economic and speedy manner. The terminal nodes reentry can be performed by the rule follow described before, with its operand reflecting the remembered access reference acquired by access.

invert – changes the sign of a value or orientation of a link to the opposite, while producing no effect on zero values or nonoriented links.

apply – organizes application of the first operand as one or a set of rules described above operating jointly from the same starting point (names of which can also be obtained by arbitrary scenario standing for this operand and not only given explicitly) to the same second scenario operand, which may be arbitrary too. If multiple application rules engaged on the first operand, the obtained results on the second operand can happen to be multiple too.

location – returns world locations of the final nodes reached by the embraced scenario, which mean for virtual nodes their network addresses, and for physical nodes physical coordinates. This may be equivalent to using in the final world positions environmental variables ADDRESS or WHERE for providing respected open values, with their subsequent collection by other echo rules (directly using location may, however, happen to be more convenient in certain cases).

distance – returns distance between two physical points defined by absolute physical coordinates expressed by its parameters, where each one can be represented by an arbitrary scenario.

A4.6 Verification

These rules provide control state thru or fail reflecting the result of concrete verification procedure, also nil as own resultant value, while remaining after completion in the same world positions where they started.

equal, nonequal, less, lessorequal, more, moreorequal, bigger, smaller, heavier, lighter, longer, shorter – make corresponding comparison between left and right operands, which can represent (or result in, if being arbitrary scenarios) information or physical matter/objects, or both. In case of vector operands, state thru appears only if all peer values satisfy the condition set up by the rule (except nonequal, for which even a single noncorrespondence between any peers will result in overall thru). The list of such rules can be easily extended for more specific applications, if supported properly on the implementation level.

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

belong, notbelong – verifies whether the left operand value (single or a list with all its elements) belongs as a whole to the right operand generally represented as a list (which may have a single element too).

intersect, notintersect – verifies whether there are common elements (values) in the left and right operands, considered generally as lists. More than two operands can be used for these rules too, with at least a single same element to be present in all of them to result thru for intersect, or no such elements for notintersect.

yes – verifies generalized state of the embraced scenario providing own control state thru in case of thru or done from the entire scenario, and control state fail in case of resultant fail or fatal (thus allowing to continue from the node where the rule started only in case of success of the embraced scenario, otherwise terminating).

no – verifies generalized state of the embraced scenario resulting with own control state thru in case of fail or fatal from the scenario, and control state fail in case of thru or done (i.e., allowing to continue from the rule's starting node only in case of failure of the embraced scenario, otherwise terminating).

A4.7 Assignment

These rules assign the result of the right scenario operand (which may be arbitrarily remote, also represent a list of values which can be nested) 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 contents of variables (see also rule element mentioned before). 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 (which may be a list) as assigned to the left operand. There are two options of the assignment, as follows.

assign – assigns the same value of the right operand (which may be a list of values) to all values (like, say, node names) or variables accessed by the left operand (or their particular elements pointed, which may themselves become lists after assignment, thus extending the lists of contents of these variables). If the right operand is represented by nil or empty, the left operand nodes or variables as a whole (or only their certain elements pointed) will be removed.

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

A4.8 Advancement

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

advance – organizes stepwise scenarios advancement in physical, virtual, executive, or combined spaces, also in a pure computational space (the latter when staying in the same world nodes with certain data processing, thus moving in time only). For this, the embraced SGL scenario operands are used in a sequence, as written, where each new scenario shifts to and applies from all terminal world points reached by the previous scenario. The resultant world positions and values on the whole rule are associated with the final steps of the last scenario on the rule (more correctly: of the invocation of all this scenario copies which may operate in parallel by starting from possible multiple points reached by the previous scenario). And the rule's resultant state is a generalization of control states associated with these final steps. If no final steps occur with states thru or done, the whole advancement on this rule is

considered as failed (i.e., with generalized state fail), thus resulting without possibility to continue scenario evolution in this direction. On default or with modifier asynchronous, the sequence of scenarios on advance develops in space and time independently in different directions, with the next scenario from their sequence replicating and starting immediately in all points reached by the previous scenario. This means that different operand scenarios in their sequence may happen to be active simultaneously at the same time, as being developed independently and in parallel, with different times of their completion. With the use of synchronous modifier, all invocations of every new scenario (in general: all its multiple copies) in their sequence can start only *after full completion* of all invocations of the previous scenario.

slide – works similar to the previous rule unless a scenario in their sequence fails to produce resultant state thru or done from some world node. In this case the next scenario from the sequence will be applied from the same starting position of the previous, failed, scenario and so on. The resultant world nodes and values in them will be from the last successfully applied scenarios (not necessarily the same from their sequence, as independently developing in different directions). The results on the whole rule, in their extreme, may even happen to correspond only to the existing value of the node in which the whole rule started (including the node's world position), 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, are possible, where in the synchronous option, different scenarios (not necessarily their same copies) can simultaneously start only after full completion of the previous parallel steps (also potentially involving different scenarios).

repeat – invokes the embraced scenario as many times as possible, with each new iterations taking place in parallel from all final positions with state thru reached by the previous invocations. If some scenario iteration fails, its current starting position with its value will be included into the set of final positions and values on the whole rule (this set may have starting positions from different failed iterations which developed independently in a distributed space). 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 scenario repetitions. Of course, the operand scenario can be easily internally organized to properly control the allowed number of iterations itself, but using this additional modifier may be useful in some 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 (possibly its many copies from different nodes) can develop (whereas

some previous ones may have already stopped in other directions). In the asynchronous case, there may be different iterations working in parallel.

align – is based on confirmation of full termination of all activities of the embraced operand scenario in all its final nodes. Only after this, the remaining scenario part, if any, will be allowed to continue from all the nodes reached.

fringe – allows us to establish certain constraints (say, by additional parameters) on the terminal world nodes reached by the embraced scenario with final values in them, to be considered as starting positions for the following scenario parts. For example, by comparing values in all terminal nodes and allowing the scenario to continue from a node with maximum or minimum value, integrating this rule with previously mentioned rules max or min like max_fringe or min_fringe can also be possible. Without additional conditions or constraints, this rule is equivalent to the previous one align.

For the advancement rules, frontal variables propagate on the forefronts together with advancement of control and operations in distributed spaces, with next scenarios or their iterations picking up frontal variables brought to their starting points by the previous scenarios (or their previous iterations), being also replicated if this control automatically splits into different branches. And the capability and variants of explicit naming and splitting into separate branches will be considered in detail in the next section.

A4.9 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. The rest of the SGL scenario will be developing from all or some of the positions and nodes reached on the rule. The 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 rule's starting position will be replicated together with their contents and used independently within different branches, to be inherited by the following scenario, if any, beyond the branching rules. Details of this replication for frontal variables with physical matter rather than information can depend on application and implementation details.

A brief explanation of how these rules work is as follows.

branch – the most general and neutral variant of branching, with logical independence of the scenario operands from each other and any possible order of their invocation and development from the starting position (say, ranging from arbitrary to strictly sequential to fully parallel, also any mixture thereof).

The resultant set of positions reached with their associated values will unite all terminal positions and values on all scenario operands involved under branch. The resultant control state on the whole rule will be based on generalization of the generalized control states on all scenario branches (based on max to min powers of control states: fatal, thru, done, and fail, as mentioned before).

sequence – organizing strictly sequential invocation of all scenario operands regardless of their success or failure, and launching the next scenario only after full completion of the previous one. The resultant set of positions, values, and rule's global control state will be similar to branch. However, the final results may vary due to different invocation order of the scenario operands and possible common information used.

parallel – organizing fully parallel development of all scenario operands from the same starting position (at least as much as this can be achieved within the existing environment, resources, and implementation). The resultant set of positions, values, and rule's control state will be similar to the previous two rules, but may not be the same, as explained before.

if – may have three scenario operands. If the *first* scenario 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 and associated values will be the same as achieved by the second or third scenarios after their completion. If the third operand scenario is absent and the first one results with fail, or only the first operand is present regardless of its success or failure, the resultant position will be the one the rule started from, with state thru and value it had at the start.

or – allows *only one* operand scenario with the resulting state thru or done, without any predetermined order of their invocation, to be registered as resultant, with the final positions and 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 terminated and canceled. If no branch results with thru or done, the rule will terminate with fail and nil value. If used in combination with the previous rules sequence and parallel, it may have the following features.

or_sequence – will launch the scenario operands in strictly sequential manner, one after the other as they are written, waiting for their full completion before activating the next operand, unless the first one in the sequence replies 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 skipped.

or_parallel – 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 due to side effects when working with common resources).

The resultant scenario chosen in all three cases described above with or 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 the resultant value.

and – activates all scenario operands from the same position, without any predetermined order, 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 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 scenario operands with their associated values. Combining rule and with rules sequence and parallel (as we did for or) will clarify their activation and termination order, as follows. (These two options can, in principle, produce dissimilar general results if different scenario operands work with intersecting domains and share information there.)

and_sequence – activates scenario operands from the same position in the written order, launching next scenario only after the previous one completes with thru or done, and terminating the whole rule when the current scenario results with fail. The remaining scenario operands will be ignored, and all results produced by this and all previous operands will be removed (as far as this can be achievable in a distributed environment).

and_parallel – activates in parallel all scenario operands from the same world position, terminating the rule when the first one in time results with fail, while aborting activity of all other operands and removing all results produced by the rule. (The completeness of such cleaning may also depend on its complexity and implementation reality in large distributed spaces, as for the previous case.)

choose – chooses a scenario branch in their sequence *before* its execution, using additional 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 or random, by forming combined ones: choose_first, choose_last, choose_random. The resultant set of positions on the rule, their values, and states will be taken from the branch (or branches) chosen.

quickest – 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 other branches (to be forcefully terminated now) could respond later with thru or done. The state, 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.) It differs fundamentally from the rule or_parallel 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 quickest may have an additional parameter establishing, for example, time limit within which replies are expected or allowed from the branches (and there may be more than one replying branch, which all with thru or done will be giving integrated result on the rule, otherwise it will terminate with failure). A similar time limit could also be established for the rule branch discussed earlier, by considering for the result only branches that reply in proper time. The special timing rules will also be considered later.

cycle – repeatedly invokes the embraced scenario from the same starting position until its resultant generalized state remains thru or done, where on different invocations the same or different sets of resultant positions (with same or different values) may emerge. The resultant set of positions on the whole rule will be an integration of all positions on all successful scenario invocations with their associated values. The following scenario will be developing from all these world positions reached (some or all may be repeating as same starting points) except the ones resulting with state done. If no invocation of the embraced scenario succeeds, the resultant state fail in the starting position with 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. (The rule will terminate, as before, with fail and nil in the starting position if no scenario invocation succeeds.)

sling – invokes repeatedly the embraced scenario until it provides state thru or done, always resulting in the same starting position with state thru and its previously associated value when the last iteration results with fail (or no invocation was successful at all).

whirl – endlessly repeating the embraced scenario from the starting position regardless of its success or failure and ignoring any resultant positions or values produced. External forceful termination of this construct may be needed, like using first in time termination of another, competitive, branch (under the higher-level rule or_parallel). It could also be possible to set an explicit limit on the number of possible repetitions or duration time in the aforementioned cycling-looping-slinging-whirling rules – by supplying them with an additional parameter restricting the repeated scenario invocations, also using the timing rules explained later.

split – performs, if needed, additional (and deeper than usual) static or dynamic partitioning of the embraced scenario into 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 aforementioned branching rules while preparing separate branches for these rules, ahead of their invocation. Some examples follow.

- If split embraces explicit branches separated by commas, it influences nothing as the branches are already declared.
- If the embraced single operand represents broadcasting move or hop (creative or removal including) in multiple directions, the branches are formed from possible variants of its elementary moves or hops, *before* their execution.
- If the rule's operand is an arbitrary scenario (not belonging to the two previous cases), the branches are formed *after* its completion, where each final position reached by the scenario (with its associated values) represents a new branch.
- If the embraced scenario terminates with a single world position but having associated list of values, each value in this list will be treated as an independent position and branch. The rest of SGL scenario will be developing in parallel from each such new branch, with its individual value available by environmental variable VALUE described before.

In a more complex modification, the rule split may be applied to the scenario represented as an explicit advancement of different scenarios (one after the other, covered by rule advance). In this case, the first such scenario will be split as above, and the remaining ones attached as the following advancement to each obtained branch, thus forming extended branches with the same replicated "tail" (which can be governed altogether by branching rules described above). By the experience with different applications of SGL and its previous variants, such advanced nonlocal splitting mechanism may be useful and effective in different circumstances.

replicate – replicates the scenario given by its second operand, providing the number of its copies given by the first operand, with each copy behaving as independent branch starting from the same current world position.

A4.10 Transference

They organize transference of control in distributed scenarios.

run – transfers control to the SGL code treated as a procedure and being a result of invocation of the embraced scenario (which can be of arbitrary complexity and space coverage, or can just be an explicit constant or variable). The procedure (or a list of them) obtained and activated in such a way can produce a set of world positions with associated values and control states as the result on the rule, similar to other rules. In case of failure to treat and

activate results of the embraced operand as an SGL scenario, this rule will terminate with value nil and state fail in the node it started.

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

A4.11 Exchange

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 and associated value sent outside (in case of physical matter, the resultant value may depend on the applications). 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.

send – staying in the current position associated with physical, virtual, executive (or combined) node sends 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 already engaged there. The rule may have an additional parameter setting acceptable time delay for the consumption of these 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 (in case of physical matter, this may depend on application and implementation capabilities), 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 too, after expiration of which the rule will be considered as failed. In case of successful receipt of the data, the rule will result in the same world position and the value obtained (information or matter) from send and state thru, otherwise will terminate with value nil and state fail.

emit – depending on implementation and technical capabilities, can trigger nonlocal to global continuous broadcasting of the data obtained by the

embraced scenario, possibly, with tagging of this source (like setting the emission frequency). Another operand providing time allowed for this broadcasting may be present too. No feedback from possible consumers of the sent data is expected. Will terminate in the application node with the broadcast value and state thru in case of success, otherwise with nil and fail.

get – tries to receive data which can be broadcast from some source (say, identified by its tag or frequency), with resultant value as the received data and state thru in the application node, otherwise with nil and state fail. Similar to the previous rule, additional operand can be introduced for limiting the activity time of this rule. No synchronization with the data emitting node is expected.

A4.12 Timing

These rules are dealing with conditions related to a time interval for the scenarios they embrace.

sleep – establishes time delay defined by the embraced scenario operand, with suspending activities of this particular scenario branch in the current node. The rule's starting position and its existing value, also state thru, will be the result on the rule after the time passed. Similar time delay of the related branch can also be achieved by assigning the current absolute time (say, received from environmental variable TIME) incremented by the needed delay value to environmental variable WHEN described before.

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

A4.13 Qualification

These rules are providing certain qualities or abilities, also setting constraints or restrictions to the scenarios they embrace, as follows.

contain – restricts the spread of abortive consequences caused by control state fatal within the ruled scenario. This state may appear automatically and accidently in different scenario development points or can be assigned explicitly to environmental variable STATE, triggering emergent completion of multiple scenario processes and removal of temporary data associated with them. The resultant position on the rule contain having state fatal inside its scenario will be the one it started from, with 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 from the scenario embraced and normally terminated. The destructive influence from state fatal is also automatically stopped if the scenario in which it may appear is covered by rule state (converting any embraced control state into a value), also rules yes and no (first changing the embraced state into fail and second into thru in case of fatal), as described before. But after these three rules the resultant world positions always correspond to the single rule's starting node regardless of what the embraced scenario produces, whereas contain without fatal results in exactly the same final positions and values (which may be many) as the scenario embraced.

release – allows the embraced scenario develop free from the main scenario, abandoning bilateral control links with it (the main scenario after the rule's activation "will not see" this construct any more). The released, now independent, scenario will develop in a usual way using its standard subordination and control mechanisms. For the main scenario, this rule will be immediately considered as terminated in the point it started, with state thru and original value there.

trackless – allows the whole embraced scenario to develop absolutely free from any previous stages (i.e., without saving any control and information links with them) like a real virus, being in some sense extreme and global variant of release. Under this rule, heritable variables, also environmental ones like BACK, PREVIOUS, PREDECESSOR, LINK, DIRECTION, and VALUE, will always result with nil, but frontal variables carrying information between different stages will work as usual.

free – differs from the previous case in that despite its independence and control freedom from the main scenario, as before, the embraced scenario will nevertheless be obliged to return the final data obtained in its terminal positions to the main scenario (if such a request issued by certain rules covering the part under free).

blind, quit, abort – after full completion of the embraced scenario, these rules result in the same position the rule started with respective states done, fail, or fatal, thus preventing further scenario development from this point (also triggering nonlocal termination and cancellation processes in case of fatal). These rules may represent more economic solutions than explicit termination of all final branches of the embraced scenario with states done or fail. If the ruled scenario is omitted (i.e., rule names standing alone), these rules will be equivalent to assigning the related states to environmental variable STATE in the position they've started.

stay – whatever the scenario embraced and its evolution in space, the resultant position will always be the same this rule started from (and not termination positions of the ruled scenario), with value nil and state thru in it. If the ruled scenario is omitted, this rule standing alone just represents an

empty operation in the current point or assignment state thru to variable STATE in it. Such empty operation with stay can also be used for declaring a new scenario branch which can be independently followed by the rest of the scenario starting from this point.

lift – lifts blocking of the further scenario developments set up by states done in the embraced scenario wherever it happened to emerge (including equivalent effect caused by rules blind), substituting them with thru and allowing further developments from all such positions by the rest of the scenario, which may be massive and space-distributed.

seize – establishes, or 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 same as from the scenario embraced. If the node has already been blocked by some other 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.

exit – terminated the innermost loop in which it is included.

A4.14 Grasping

The rule's identifier can be expressed not only as a directly given name but also by the result produced by a scenario of any complexity and treated as rule's name. It can also be a compound one, integrated from multiple names provided by different scenarios, so in general we may have the following:

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

Under this extended definition, resulting from recursive SGL syntax, additional parameters can also associate with the rule's names, before embracing the main scenario operands. Such aggregation can simplify the structure of SGL scenarios, also making them more flexible and adjustable to changing goals and environments in which they operate.