OUT OF THE BOX: SELF-ORGANIZING AWARENESS

MICHAEL MANTHEY

tauquernions.org
manthey@acm.org

Abstract: Space-like computation - the complement to Turing's time-like computation - is inherently distributed, self-organizing, and capable of multi-level awareness [aka. consciousness]. The section headings are: Introduction, Definition and Description of Space-like Computation, The Fundamental Reasoning Behind Space-like Computation, Primitive Functionalities, Use of Event Windows, How Space-like Computations Satisfy Goals, Security, and Miscellaneous; includes an Appendix with the *essential source code*. The paradigm has the signature $U(1) \times SU(2) \times SU(3) \times SO(4)$ using geometric (Clifford) algebra over $\mathbb{Z}_3 = \{0, 1, -1\}$. This document is extracted from the USPTO application of 2015 [incl. international], with minor edits and annotations; the patent issued on April 28, 2020.

1 INTRODUCTION

Computation, whether understood as explicit executions on hardware or more abstractly as *literal sequences* of otherwise unspecified events, is inherently *time-like*, in that at its very core, each step is fundamentally *irreversible*. Sequential computation by its very nature *consumes* information, and this includes so-called *parallelism* - the organized execution of *multiple* copies of one or more sequential processes. These are facts supported by key mathematical theorems by Alan Turing (1920-40's) and Claude Shannon (1948++). This kind of computation is therefore called *time-like*, and is characteristic of virtually all computation today.

The present invention [1] presents a *new* kind of computation - *Space-like Computation* - that is the conceptual opposite of time-like computation. Space-like computation is *reversible* (ie. wave-like) and *creates* information (see the Coin Demonstration, below). This invention rests on novel mathematics, which show that space-like computations are in principle different from traditional sequential (and parallel) computations.

As a result there is no inbuilt sense of "time" in a space-like computation, although there is plenty of *change*. This change can either be viewed as the evolution of a complex waveform (representing the activity spectrum of the computation) or as the dynamics of a population of discrete concurrent bit-flips - *the two views are EXACTLY equivalent*, courtesy of another theorem, Parseval's Identity (1799).

Parseval's Identity states that the projection of a function $\mathcal{F}$ onto an n-dimensional orthogonal space is the Fourier decomposition of $\mathcal{F}$. Parseval's Identity is a generalization of the Pythagorean theorem to n dimensions. In the n-dimensional coordinate system, $\mathcal{F}$'s current value corresponds to a hyper-hypotenuse in an n-dimensional hyper-cube, and the projection breaks that hyper-hypotenuse down into the various pieces along each of the dimensions that go into its construction.

To construct an n-dimensional cube, begin with an ordinary plane right triangle with unit sides a and b. Reflect this triangle on its hypotenuse, forming a square with sides a and b, area ab, and diagonal $d = \sqrt{a^2 + b^2}$. Next, lift this square one unit vertically (c) to make a unit cube with volume abc. Its diagonal is $d = \sqrt{a^2 + b^2 + c^2}$ and this sum-of-squares symmetry continues as we make a 4d cube, then 5d, etc.

At the same time, going back to the starting right triangle, we can also express the sides a and b as $a = \cos\theta$ and $b = \sin\theta$, where θ is the angle between a and the hypotenuse. And now all becomes clear: substituting these sine and cosine equivalents for $a, b, c, d, ...$ up through the dimensions will yield, for the n-dimensional hypotenuse (= the current value of the function $\mathcal{F}$, whose projection we began with), a big sum of ... sines and cosines, ie. Fourier's world.

So the world of waves and the world of orthogonal coordinate systems are the same world. It is in the latter that we will connect to computation. The connection is this: let each dimension correspond to the state of some process, where all these processes $a, b, c, \ldots, ab, ac, \ldots, abc, \ldots$ are notionally independent (think *orthogonal*), though interacting otherwise freely and concurrently. [1] Looking at the ongoing Heracletian flurry of process-state evolution in such a system, the high frequency Fourier bands correspond to short-term, fine-grained details, and low frequency bands to long-term symmetries and global developments. These cross-summed Fourier bands constitute the world of *qualia* — the *feeling* of

1 In the geometric (Clifford) algebra over $\mathbb{Z}_3 = \{0, 1, -1\}$ used here, 1-vectors are processes with one bit of state, ± 1, whence an m-vector has m bits of state. For concurrent processes a, b write $a + b$; when a, b interact write ab; ab too is a process with external appearance ± 1. And so on. NB: $ab = -ba \cong \sqrt{-1}$.

(eg.) redness vs. the optical frequencies detected by individual retinal cells.

We posit that awareness, and consciousness as awareness of awareness, are both *on-going self-preserving* resonances in this hierarchical process structure ... is the computational version of the universal wave function, having both discrete and wave-like elements and properties. Although we don't show it here, the overall mathematical structure of the paradigm is $U(1) \times SU(2) \times SU(3) \times SO(4)$; see [8].

A not-too-misleading analogy is the relationship between a computer's operating system and the myriad processes it attends to. The operating system corresponds to a space-like computation, which expresses the causal potential (what events *could* happen next), whereas the sequential processes it is manifesting correspond to what goes on in Classical 3+1d.

We note that in Bostrom's terminology [5], space-like computation is a "slow build" technology (because it works bottom-up), which is a very desirable trait considering that space-like computations are self-organizing.

In the following, §2 and §3 provide background information, whereas the new technology is presented beginning in §4.

2 DEFINITION AND DESCRIPTION OF SPACE-LIKE COMPUTATION

A *space-like computation* S is characterized by the following properties:

2.1. A *space-like computation* is *distributed*, which means that the entire system, consisting of a potentially very large number of concurrent, independent but interacting entities, exhibits *coherent global behavior* with little or no centralized control. The exact technological criteria for how to design and build distributed systems in general have proven elusive, which lacuna the present invention fills.

2.2. A *space-like computation* is *self-organizing*, meaning that given some *method* and the necessary surrounding environmental inputs, that method will - over time - assemble these inputs into a coherent entity of discrete units that interacts with that environment in a stable fashion. How to design self-organizing computations is a current topic of high-profile research, which the present invention advances.

2.3. A *space-like computation* is *hierarchical*, meaning that the self-organization includes the creation of new discrete units - representing combinations of (combinations of ...) the initial units - that themselves become fodder for the self-organizational method. *Hierarchy* is a universal and well-proven technological tool to control conceptual complexity, a tool whose use is found across the industry, from programming languages to data bases to communication protocols. *Uniquely*, this hierarchy is that of the calculus (ie. integration and differentiation), but in the geometric idiom of boundaries and co-boundaries.

2.4. A *space-like computation* is *not Turing-limited*, meaning that a true space-like computation *cannot* be simulated by a universal Turing machine, which is by definition limited to sequential computations (including parallelism). This is the topic of the Coin Demonstration below, which gives an easily understood counter-example. It is widely thought that Turing's theorems prove that *all* computation is sequential *in principle*, so the present invention is strongly innovative.

2.5. A *space-like computation* is *meaningless* unless connected to, and interacting with, a surrounding environment because it then cannot grow. As opposed, say, to a sequential computation that computes the value of π, which computation presumably would find genuine meaning in its solitary endeavor, its world being complete.

2.6. A *space-like computation* uses a *broadcast/listen communications discipline*. ie. broadcast one's own state and listen (ie. react) accordingly to others'. The reason for this is that the alternative, a request/reply discipline, is inherently functional in character, since it implements $y = f(x)$: "Request that f do its thing on x and reply with the result". But one man's y is another man's x, so $z = g(y)$ is also a possibility. So then $g(y) = g(f(x))$, and the sequence *first-do-*f*-then-do-*g is namely *sequential*, $f; g$, ie. time-like. This is namely how contemporary computer systems are organized. In contrast, a space-like computation assembles the steps in its sequential processes *on-the-fly*, as described in §6. Contemporary technology largely ignores broadcast/listen protocols namely because they don't fit the dominant $y = f(x)$ organizational paradigm.

As described in the provisional filing [1], it is possible that some space-like computations might be[come] self-aware, but this is not a constituting property.

3 THE FUNDAMENTAL REASONING BEHIND SPACE-LIKE COMPUTATION

The fundamental reasoning underlying this (and earlier) patents [2,3] is best explained via the following Coin Demonstration.

The following ***Coin Demonstration*** clarifies.

ACT I *A man stands in front of you with both hands behind his back. He shows you one hand containing a coin, and then returns the hand and the coin behind his back. After a brief pause, he again shows you the same hand with what appears to be an identical coin. He again hides it, and then asks, "How many coins do I have?"*

Understand first that this is not a trick question, nor some clever play on words - we are simply describing a particular and straightforward situation. The best answer at this point then is that the man has "at least one coin", which implicitly seeks *one bit* of information: two possible but mutually exclusive states: *state1* = "one coin", and *state2* = "more than one coin".

One is now at a decision point - *if* one coin *then* doX *else* doY - and exactly one bit of information can resolve the situation. Said differently, when one is able to make this decision, one has *ipso facto* received one bit of information.

ACT II *The man now extends his hand and it contains two identical coins.*

Stipulating that the two coins are in every relevant respect identical to the coins we saw earlier, we now know that there are *two* coins, that is, *we have received one bit of information*, in that the ambiguity is resolved. We have now arrived at the demonstration's dramatic peak:

ACT III *The man asks, "Where did that bit of information come from?"*

Indeed, where *did* it come from?! [2]

The bit originates in the *simultaneous presence* of the two coins - their ***co-occurrence*** - and encodes the now-observed *fact* that the two *processes,*

2 [Think about it! Where *did* that bit come from? Thin air?]

whose states are the two coins, respectively, do not exclude each other's existence when in said states. [3]

Thus, there is information in (and about) the environment that *cannot* be acquired sequentially, and true concurrency therefore *cannot* be simulated by a Turing machine. Can a given state of process a exist simultaneously with a given state of process b, or do they exclude each other's existence? *This* is the fundamental distinction.

More formally, we can by definition write $a + \tilde{a} = 0$ and $b + \tilde{b} = 0$ [~ = *not* = *minus*] meaning that (process state) a excludes (process state) $\tilde{a}$, and similarly (process state) b excludes (process state) $\tilde{b}$. [4] Their *concurrent* existence can be captured by adding these two equations, and associativity gives two ways to view the result. The first is

$$(a + \tilde{b}) + (\tilde{a} + b) = 0$$

which is the usual excluded middle: if it's not the one (eg. that's +) then it's the other. This arrangement is convenient to our usual way of thinking, and easily encodes the traditional *one/zero* (or $1/\tilde{1}$) distinction. [5] The second view is

$$(a + b) + (\tilde{a} + \tilde{b}) = 0$$

which are the two superposition states: either both or neither.

The Coin Demonstration shows that *by its very existence*, a 2-co-occurrence like $a + b$ contains one bit of information. Co-occurrence relationships are *structural*, ie. *space-like*, by their very nature. This *space*-like information (vs. Shannon's *time*-like information) ultimately forms the structure and content of the Fourier bands, eg. {all 2-vectors}.

Sets of m-vectors - $\{xy\}, \{xyz\}, \{wxyz\}, \ldots$ - are successively lower *under*tones of the concurrent flux at the system boundary $x + y + z + \ldots$, and constitute a simultaneous structural *and* functional decomposition of that flux into a hierarchy of stable and meta-stable processes. The lower the frequency, the longer-term its influence.

But where do these m-vectors come from?

---

3 Cf. Leibniz's indistinguishables, and their being the germ of the concept of space: simultaneous states, like the presence of the two coins, are namely indistinguishable in time.

4 This is the logical bottom, and so there are no superpositions of $a/\tilde{a}$ and $b/\tilde{b}$: they are 1d exclusionary distinctions . Superposition first emerges at level 2 with ab via the distinction *exclude* vs. *co-occur*.

5 Since $\tilde{x}$ is not the same as $0x$, an occurrence $\tilde{x}$ is meaningful; in terms of sensors, $x/\tilde{x}$ is a *sensing* of an externality of x, not x itself.

ACT IV *The man holds both hands out in front of him. One hand is empty, but there is a coin in the other. He closes his hands and puts them behind his back. Then he holds them out again, and we see that the coin has changed hands. He asks, "Did anything happen?"*

This is a rather harder question to answer. [6] To the above two *co*ncurrent *exclusion*ary processes we now apply the *co-exclusion inference*, whose opening syllogism is: *if* a excludes $\tilde{a}$, *and* b excludes $\tilde{b}$, *then* $a + \tilde{b}$ excludes $\tilde{a} + b$ (or, conjugately, $a + b$ excludes $\tilde{a} + \tilde{b}$)... . This we have just derived.

The inference's conclusion is: *and therefore,* ab *exists*. The reasoning is that we can logically replace the two *one*-bit-of-state processes a, b with one *two*-bits-of-state process ab, since what counts in processes is sequentiality, not state size, and exclusion births sequence (here, in the form of *alternation* between the two complementary states). That is, the existence of the two co-exclusions $(a + \tilde{b}) \mid (\tilde{a} + b)$ and $(a + b) \mid (\tilde{a} + \tilde{b})$ contains sufficient information for ab to be able to encode them, and therefore, logically and computationally speaking, ab can rightfully be instantiated.

We write $\delta(a + \tilde{b}) = ab = -\delta(\tilde{a} + b)$ and $\delta(a + b) = ab = -\delta(\tilde{a} + \tilde{b})$, where δ is a co-boundary operator (analogous to integration in calculus); differentiation is the opposite, $ab \xrightarrow{\partial} a + b$. A fully realized ab is, we see, comprised of two *conjugate* co-exclusions, a sine/cosine-type relationship. Higher grade operators abc, $abcd$, ... are constructed similarly: $\delta(ab + c) = abc$, $\delta(ab + cd) = abcd$, etc.

We can now answer the man's question, *Did anything happen*? We can answer, "Yes, when the coin changed hands, the state of the system rotated 180°: $ab(a + \tilde{b})ba = \tilde{a} + b$." We see that one bit of information ("something happened") results from the alternation of the two mutually exclusive states. [The transition $a + b \xrightarrow{\delta} ab$ is in fact the basic act of perception, called the *first perception*, subsequent meta-perceptions being derivative.]

Summarizing, *every Action in a space-like computation carries the above-described semantics, because thay are all based on co-exclusion.*

6 What makes it tricky is that if at the same time as the man hides the coin he has shown you, you walk around to his back side (be careful how you do it), then it would look to you like nothing happened at all, *vis a vis* the coin, when he shows it again: it's still in the same place relative to you.

4 PRIMITIVE FUNCTIONALITIES

Our claims derive from the innovations described in this and the following sections. The attached *TLinda* code in the Appendix specifies only the minimal functionality, ie. a minimal "suggested implementation" to achieve a space-like computation.

4.1 NOVEL CONCURRENCY-CONTROL OPERATIONS

A space-like computation is easiest to specify using a *coordination language*, in our case *TLinda* [1], which for good reason is the most popular programming language in Tlön. A coordination language is wholly concerned with coordinating the interaction of concurrent processes, here called *threads*.

Consequently, *TLinda* has *no* facilities for arithemetic calculation, although it retains the usual control-flow facilities: if-then-else and the loop constructions while-do, repeat-until, and (novel) forever-loop; interestingly, only forever-loop is actually used.

TLinda derives from the *Linda* language, arguably the originator of the idea (1985, [6]), which postulates a global tuple space *TS* with four operations on a tuple *T = [field1, field2, ...]: Out(T), Rd(T), In(T), Eval(T)*.

We now describe these standard *Linda* operations.

Out(T) makes *T present* in *TS*.

Rd(T), ***if*** *T*'s form *matches* that of a *present* tuple in *TS*, will ***then*** accordingly bind *T*'s variables to the corresponding fields of the match. Otherwise the *Rd blocks* the issuing thread until a tuple matching *T* shows up in *TS*.

In(T) is the same as *Rd(T)*, except that it also *removes T* from *TS under mutual exclusion*. The latter assures that one can create a synchronization token when necessary. Otherwise, each thread manages its *own* tuples, which once allocated remain so - only a tuple's *presence counter* (never < 0) indicates its availability.

Finally, *Eval(T)* treats tuple *T* as the code-descriptor of a thread-body to be executed, and a new independent thread is spawned. There is *no* sense of *Eval(T)* as a function that will return a value to the thread that issued the *Eval* (or any other thread, for that matter).

To these classic *Linda* operations *TLinda* adds *Co U,...,V* and *NotCo U,...,V* and variants *AntiRd*, *AntiCo* and *AntiNotCo*. These test for and *block* on (*wait* for) the co-occurrence or lack thereof, respectively, of the tuples *U,...,V* in *TS*. *AntiRd* blocks on the presence of *U* alone.

Each of these has a "one-shot" predicate version - *Rdp*, *Cop*, *NotCop*, *AntiCop*, and *AntiNotCop* - that performs the usual operation (if possible, and no blocking) and returns a True/False indication thereof.

Finally, *TLinda* has a special construction - *Event Windows (EW)* [3] - for efficiently discovering co-exclusions among tuples, which are turned into Actions (think m-vectors, $m \geqslant 2$). Recognizing that m-vectors can themselves be the subject of an EW's focus provides the self-organizing component of a space-like computation. Claimed in [2,3].

Thus the overall style of the computation derives from the utterly concurrent *associative match* of tuples (expressing current process states, cf. *broadcast*) in a global space, *combined* with the inbuilt synchronization properties of the tuple operations themselves.

4.2 NOT ALL SENSORS NEED HAVE EFFECTORS

A *sensor* is a *TLinda* object that converts some phenomenon in the surround into a binary signal, where +1 indicates that whatever the sensor senses is currently *present* in *TS*, and -1 means that whatever it senses is currently *absent* from TS. [7] The $\mathbb{Z}_3 = \{0, 1, 2\} = \{0, 1, -1\}$ number system can clearly be generalized to $\mathbb{Z}_n$ or even the real numbers $\mathbb{R}$, but this comes at the expense of mathematical tractability, that is, the ability to prove that a given space-like computation does what is claimed it does.

A computation invokes an *Effector* (E) to influence its environment, and is defined in terms of a sensor s that detects the effector's effect on the surround: $s \xrightarrow{E} -s = \tilde{s}$. More complicated effectors can easily be defined using this template.

It follows naturally that in space-like computations, sensors will vastly outnumber effectors, as it is the number of sensors n that determines the amount of information that can be learned, which is of order $\mathcal{O}(2^n)$, whereas (in principle) effectors *consume* information.

7 A zero value from a sensor indicates an error or exception, not least because *Void* is not a 'value'. Also, *sensing* is to be distinguished from *measuring*: $1 \pm x$ vs. $-1 \pm x$.

4.3 SPACE-LIKE COMPUTATION IS FUNDAMENTALLY NON-NUMERICAL

Unlike virtually all time-like computations, a space-like computation does little or no arithmetic. Nevertheless, although not part of the main thrust, it is inevitable that some arithmetic must be performed. *TLinda* can simply be expanded to include it, and/or it can be simulated by a space-like computation, for example one that maps the binary number 10011 into the "binary" hierarchy $+abcde - abcd - abc + ab + a$ via the mappings $1 \mapsto +$ and $0 \mapsto -$. Another possibility is to define special-purpose Actions that respond to requests for arithmetic calculations of various kinds.

4.4 TO SAVE/RESTORE A SPACE-LIKE COMPUTATION

It is possible to record co-exclusions and Actions as they occur, which record can serve as a means to restore or clone a given space-like computation. However, keeping in mind the No-Copy theorem of quantum mechanics, the restored or cloned computation must necessarily differ from the original because the particular internal state at the time the latest item was recorded must also be preserved, which computation ("dump") must therefore take place *outside of* the space-like computation that is being copied. Thus the procedure consists of the two steps: record continually, and when desired, dump. The restore operation then uses this dump to re-establish the structure and state of the interrupted computation.

4.5 EQUIVALENCE CLASSES AS "UNITS", AS SENSORS, AS GOALS.

Every Action is labelled with its *level* and its *grade* therein, and perhaps other categorical information, that can identify the Action as the member of a given equivalence class of Actions. An Action's *level* is based on the number of co-exclusions beneath it in the hierarchy. An Action's *grade* [8] is one of $\{1, 2, 3\}$ and higher grades do not exist (see §5, below) in the suggested implementation.

Examples of such categories are "all Actions with grade $2 \bmod 4$", or "all Actions with [level,grade] = $[12, 3]$". These categories correspond

8 A 1-vector a has grade 1, a 2-vector ab has grade 2, etc. A scalar number like ± 1 has grade 0.

to various frequency bands as established by Parseval's Identity. Every such category can be represented by a single Action whose external state (spin) is determined by some property of its constituents, for example the sum of their spins in $\mathbb{Z}_3$ arithmetic. These Actions can then be treated like any other.

5 USE OF EVENT WINDOWS

Earlier patents [2,3] specify that in principle, *any* tuples in Tuple Space may be co-excluded to form *Actions* - which is what Event Windows effectively do (namely δ) - but give no hint of *which* tuples it is best to focus on. With efficient hardware and software implementation of large systems in mind, the following innovations [9] optimize this generality:

5.1 *TLinda*'s novel *Opposite* operation specifies that the two opposite spin values of a newly formed Action are not themselves to be co-excluded (as this would be redundant, though perhaps thinkable in some research context). This constitutes a simple but crucial optimization.

5.2 When instantiating an Action (ie. implementing δ), ensure that its two constituent boundaries B_1, B_2 ("parent nodes") do not share any sensor s_x. That is, require that $B_1 \cap B_2 = 0$. This is efficiently accomplished by recording each constituent sensor s_i in a list associated with the given Action at the time of instantiation. Thus, at any level, the Action's list contains the names of all of the sensors $s_i \in \{s\}$ that actually constitute that Action. Sensors, being the bottom *level*, have no parents. B_1, B_2 , usually themselves being Actions, will possess such lists and their intersection is thus easily computed. If $B_1 \cap B_2 \neq 0$, then no Action is instantiated. (This is not an error, just a semantically disallowed combination, because it mis-states the effective level and grade of the Action.)

5.3. When instantiating an Action, emit a tuple that indicates that it is a *top-node*, ie. has no children. Complementarily, remove any tuples that mark the Action's parent nodes as being on top.[10]

5.4 An Action of grade n can express 2^{n-1} distinct co-exclusions. This means that there can, in principle, come to exist 2^{n-1} allocated and executing instances of the given Action. These instances must then further engage in a mutual-exclusion protocol to ensure that only one of

9 The underlying mathematics is novel, and so there is no prior art.
10 Cf. [4] : Corm thread, p.3.

them is in play in a given context. The first of these outcomes is wasteful and the second complex. To avoid both, the Action's code should specify/control all 2^{n-1} alternatives in *one* thread, which thread's very sequentiality has the pleasant side-effect of automatically yielding the desired mutual exclusion.[11] It is a design decision whether to instantiate all 2^{n-1} instances on the first Event Window hit, or only when an Event Window hit specifically prompts.

5.5 To ensure both stable operation and tractable mathematical semantics (thus allowing formal proof-of-function), Event Windows *shall* be employed as follows:

Notation: $EW(p, q)$, where $p, q > 0$, specify two vectorial *grades* whose corresponding Actions' names and spins are to be noted in the Event Window, and which thus can come to be co-excluded to form a new Action.

5.5.1. $EW(1, 1)$, which co-excludes entities with grade $1 \bmod 4$, creating an Action with grade $1 + 1 = 2 \bmod 4$. For example, $\delta(a, b) \mapsto ab$, = grade 2.

5.5.2. $EW(1, 2)$, which co-excludes entities with grades $1 \bmod 4$ and $2 \bmod 4$, for example $\delta(a + bc) \mapsto abc$, = grade 3.

5.5.3. $EW(2, 2)$, which co-excludes entities with grade $2 \bmod 4$, for example $\delta(ab + cd) \mapsto abcd$, = grade 4. The sign of $abcd$ is then mapped to a (conceptually new) 1-vector, ie. grade 1. This 1-vector represents $abcd$.

5.5.4. $EW(2, 3)$, which co-excludes entities with grades $2 \bmod 4$ and $3 \bmod 4$, for example $\delta(ab + cde) \mapsto abcde$, = grade 5. The sign of $abcde$ is then mapped to a (conceptually new) 1-vector, ie. grade 1. This 1-vector represents $abcde$.

5.5.5. $EW(3, 3)$, which co-excludes entities with grade $3 \bmod 4$, whence, similarly, $\delta(abc + def) \mapsto abcdef$ = grade $6 \overset{\bmod 4}{\longmapsto}$ grade 2, and $abcdef$ is similarly replaced by a representative 2-vector on level Lvl+1 with roots in its two 3-vector parents on level Lvl.

5.6. $EW(1, 3)$ will be disallowed in most applications because it deals in conditional facts - things that might or might not occur or be 'true', since the co-exclusion $a + bcd \,|\, -a - bcd \xrightarrow{\delta} abcd$ implies that when

11 The TLinda code shown in the accompanying document does *not* do this, for clarity and brevity.

$\mathtt{a}$ indicates 'not present in the surround', the reality-status of $\mathtt{abcd}$ becomes dubious. It characterizes speculative thought of the kind found in scientific endeavor, speculations, day-dreams, fantasies, conspiracy theorizing, and the like. In the lowest tier, $1+3$ is identified with dark matter, and is (roughly) the square root of $2+2$-built spacetime [because rotated 90°] ... most products of such $3+1 \longrightarrow 4$ elements are zero, but a minority map to $2+2$, eg. $(\mathtt{a}+\mathtt{bcd})(\mathtt{b}+\mathtt{acd}) = -\mathtt{ab}-\mathtt{cd}$ (and curiously $(\mathtt{b}+\mathtt{acd})(\mathtt{a}+\mathtt{bcd}) = \mathtt{ab}-\mathtt{cd}$. We call this relationship, *which is a literal information-transformation mechanism*, the **Keyhole Identity** for how it connects time and space.

Thus our hierarchical construction rises in *tiers* of four levels, corresponding to grades $1\text{mod}4$ through to $4\text{mod}4$. These tiers connect the quantum world to our every-day world. The "classical" sequential world and the quantum mechanical world are present in every tier, as follows.

Note that all possible co-exclusions up to grade 5 are considered, and chief among these are the 3d *quaternion* triples $\{\mathtt{ab}, \mathtt{bc}, \mathtt{ca}\}$ of $SU(2)$, which are the $1+1$ co-exclusions; *and* the $2+2$ 4-dimensional *tauquernion* triples $\{\mathtt{ab}-\mathtt{cd}, \mathtt{ac}+\mathtt{bd}, \mathtt{ad}-\mathtt{bc}\}$, underlying $SO(4)$, giving 3+1d spacetime; *and* the $2+3$ *tauquinion* triples $\{\{\mathtt{ab}+\mathtt{cde}\}\}$ whose variants make $SU(3)$, the seat of electro-magnetism. That is, the entire bubble-up and trickle down chain passes through (ie. is successively rotated by) three distinct representations of the quaternions, and only these ... *everything is made out of space*. This, along with co-occurrence's resonance with Leibniz' construction of space[12], is the reason I call this kind of computation *space-like*.

These functionalities will all likely be implemented in the same module (the *Corm*), and activated when that module's Event Window discovers a new co-exclusion. This table summarizes:

pairs	δ(pair)	new level		
$3\bmod 4,\ 3\bmod 4$	$\rightsquigarrow$	6	$= 2\bmod 4$	$\searrow$
$2\bmod 4,\ 3\bmod 4$	$\rightsquigarrow$	5	$= 1\bmod 4$	$\searrow$
$2\bmod 4,\ 2\bmod 4$	$\rightsquigarrow$	4	$= 0\bmod 4$	$\searrow$
$1\bmod 4,\ 2\bmod 4$	$\rightsquigarrow$	3	charge	$.\ \downarrow$
$1\bmod 4,\ 1\bmod 4$	$\rightsquigarrow$	2	spin	$\overset{6}{\leftarrow}\downarrow$
$0\bmod 4,\ 1\bmod 4$	$\rightsquigarrow$	1	existence	$\overset{4,5}{\leftarrow}$

12 Even though two things are indistinguishable [eg. co-occurring events are indistinguishable in the time domain] one still knows there are two because they are *distinct*. Therefore there is a distance separating them. Place a metric on this distance relationship and you have "space".

The top-most "level" of the Action hierarchy is a dynamically changing set of Actions, namely those that (so far) have no "child" nodes. Once these Actions are co-excluded with each other, the resulting Action, a new child, replaces its constituent parent Actions in the Top-Most club.

Devote one or more Event Windows to the set(s) of top-most Actions. These Windows provide a high-level view of the computation's growth and evolution, and as well an opportunity to steer these.

To delete an Action, delete also all upward (child) Actions that depend on it. [Actions are only *added* via Event Windows.]

To merge two space-like computations P, Q, add P's sensor set to Q's, and vice versa, whether literally or effectively.

6 HOW SPACE-LIKE COMPUTATIONS SATISFY GOALS

A space-like computation is first and foremost *hierarchical*. This hierarchy is formed by Actions that are discovered and instantiated via the Event Window mechanism (§5). The bottom-most level is the sensors of the surrounding environment, expressed mathematically as 1-vectors. All succeeding levels are m-vectors, $m > 1$, insomuch as their constituent Actions are combinations of 1-vectors. [The optimization that telescopes $m \bmod 4$ as described in §5 is ignored in this section, as it does not impinge.]

Actions exist to carry out *goals*, which when successfully carried out, end with the successful operation of one or more effectors, which propagate the goal's intent into the surrounding environment.

Goals arise from *impulses*, which originate (for present purposes) in the environment.

The space-like computation that ensues upon the receipt of an impulse "change x to $\tilde{x}$" assembles a set of changes to the environment, these changes to be carried out in an order that achieves the desired effect. An example is re-setting the state of a satellite, which has many sensors and many effectors. Finding this set of changes and carrying them out in the proper order is not so easily done!

Suppose for example that there are five different actions that can be carried out. Then *each step* in the final plan must be, or have been, assembled from these five different *possibilities*. This means that the number of

possible plans of n steps is 5^n, an exponential growth. Worse yet, in any realistic system - like a satellite - "five" is far too small, ... 50 effectors and $n = 500$ steps are more likely, yielding 50^{500} possibilities! Obviously searching for a workable plan in a space this size is a formidable task.

The following sections describe our novel and very efficient solution to this problem.

6.1 OPTIMIZATIONS TO EFFECT ON-THE-FLY SEQUENCING

The first optimization is to assemble plans on-the-fly rather than trying to work it all out ahead of time: given that the environment will constantly surprise, this is unworkable (cf. Stanford AI's *Shakey*). Instead, therefore, make only very simple plans for very simple situations, which are deployed automatically. Anything more complicated is to be handled "upstairs". This is the basis for the very successful *subsumption architecture* in robotics [7], which emphasizes simple fail-proof bottom-level effectors (eg. "seek light/darkness", "follow the wall", etc.). Anything beyond these simple effectors is the job of higher-level modules, which are designed and assembled on the same philosophy. One can rightly say that a space-like computation's self-organizing ability *automates* the structuring and assembly of arbitarily complicated subsumption architectures.

The second optimization is to fight the exponential explosion of possibilities with the logarithmic reduction offered by *hierarchy*. A hierarchy of Actions - first Actions sitting directly above the system's effectors, then Actions sitting above and including these, etc - all deduced from the environment's observed behavior, is key. The Event Window mechanism builds exactly such hierarchies, but as revealed in [2,3], that mechanism is unselective and therefore does not scale well to large systems. This is remedied by the present invention, as described in the preceding section on the Use of Event Windows.

The *third optimization* is *a specific way to use the Action hierarchy to create complex many-leveled plans on-the-fly*.

A plan is expressed in terms of goals to be achieved. A *goal* is a state of S that S tries to achieve by carrying out the various actions that it is capable of. [13] Usually the goal state is a set of sensor states, but it could also be some more abstract state that corresponds to some higher-level consideration.

13 Which includes the environment as reflected in its sensors.

There are two phases to this process, the *bubble up* of both sensory information and introduced goals (ie. *impulses*) from S's boundary to the upper levels of hierarchy, and the subsequent *trickle down* of the resulting goals of reaction. Both of these concurrent flows are the responsibility of Actions, and all Actions behave identically in this respect. Thus, the various threads constituting an Action are grouped into three main sets:

- Threads responsible for peering *downward* in the hierarchy for what might have changed, and when a such a change occurs, update the Action's state accordingly, and *bubble* this change further *up*ward to its child nodes. This has two aspects:
  - Bubbling sensory information up
  - Bubbling impulses up (which triggers back-chaining)
- Threads responsible for back-chaining
- Threads responsible for peering *upward* in the hierarchy, looking for *downward trickling* goals with this Action's name. This again has two aspects (if the Action is *grounded* [14]):
  - When such a goal appears, split it into goals on the Action's parent[15] nodes.
  - Instigate "firing" on any forward chain(s).

The *TLinda* code for all of these threads appears in the Appendix, which code only specifies minimal functionality, a "suggested implementation".

6.2 META-EFFECTORS: IMPULSES UP AND GOALS DOWN

The bubble-up of sensor states is accomplished by the *TLinda* code for *Meta-Sensors*, and is very straightforward. [16] This section (§6) therefore treats only the upward propagation of introduced goals, called *impulses*,

14 *Grounded* means that the pre-conditions for the Action to succeed are currently met by the environment; aka. *Relevant*.

15 Since the hierarchy is built bottom-up (cf. co-boundary), *created* nodes [ie. upward] are *child* nodes, and *below* the child nodes are their *parent* nodes. Ie. the opposite of other trees in computing, which are drawn with the root at the *top*, whence parent nodes lie *above* their children.

16 Cf. Appendix, Object *SenseEffect*.

and the consequent downward propagation (*trickle down*) of *goals* of actual intent, ending at the level of individual effectors, which realize that intent in the surround.

An impulse is *introduced* at some level of the Topsy hierarchy, which - due the hierarchy's self-similarity - we can denote as "the bottom" level for present purposes. This "introduction" requests simple inversion in the value of the sensor $X \rightsquigarrow \tilde{X}$, where $\tilde{}$ means "opposite of". We use either the algebraic forms $(\pm 1 + X)$, $|X| = 1$, or the Tlinda tuple encodings $[\uparrow, X, \mathsf{NotX}]$ for an *impulse* and $[!, X, \mathsf{NotX}]$ for a *goal*; in either case, X refers to a sensor or meta-sensor.

6.3 EXAMPLE PROBLEM

To explicate further, consider the following Block World, with sensors:

Hand	*Hand@*	*Hand@*	*Hand@*	*Place A*	*Place B*	*Place C*
Full/Empty	*a/not*	*b/not*	*c/not*	*Full/Empty*	*Full/Empty*	*Full/Empty*
$h/\tilde{h}$	$@_a/\tilde{@}_a$	$@_b/\tilde{@}_b$	$@_c/\tilde{@}_c$	$a/\tilde{a}$	$b/\tilde{b}$	$c/\tilde{c}$

This Block World consists of three *places* A,B,C that can each hold max one block; a *Hand* that can be @ each of the places A,B,C; and the Hand can be either *Full* or *Empty*.

The *primitive* effectors in the Block World are

- *Hand Grasp/Release*:
  - grasp : $\tilde{h} \rightarrow h \quad + \quad$ place $\quad x \rightarrow \tilde{x}$;
  - release : $h \rightarrow \tilde{h} \quad + \quad$ place $\quad \tilde{x} \rightarrow x$.
- *Hand Move1 Left/Right*, eg. x, y change, z doesn't: [17]
  - $@_x \rightarrow \tilde{@}_x$
  - $\tilde{@}_y \rightarrow @_y$
  - $@_z \rightarrow @_z$

Previous to this, the computation has deduced the following Actions ("$|$" means xor, with precedence less than +) :

17 Although this describes things as "to the left" and "to the right" etc, this is local to the Hand, and not relative to Places A, B, C, whose local adjacency geography is unknown.

Hand at x *and not at* y: $@_x + \tilde{@}_y \mid \tilde{@}_x + @_y \stackrel{\delta}{\Longrightarrow} @_x\tilde{@}_y \leftrightarrow \tilde{@}_x @_y$

Grasp/Release ON a, b, c:

$$a + \tilde{h} \mid \tilde{a} + h \stackrel{\delta}{\Longrightarrow} a\tilde{h} \leftrightarrow \tilde{a}h$$

$$b + \tilde{h} \mid \tilde{b} + h \stackrel{\delta}{\Longrightarrow} b\tilde{h} \leftrightarrow \tilde{b}h$$

$$c + \tilde{h} \mid \tilde{c} + h \stackrel{\delta}{\Longrightarrow} c\tilde{h} \leftrightarrow \tilde{c}h$$

Grasp/Release AT a, b, c:

$$@_a: \quad a\tilde{h} + \tilde{@}_a \mid \tilde{a}h + @_a \stackrel{\delta}{\Longrightarrow} a\tilde{h}\tilde{@}_a \leftrightarrow \tilde{a}h@_a$$

$$@_b: \quad b\tilde{h} + \tilde{@}_b \mid \tilde{b}h + @_b \stackrel{\delta}{\Longrightarrow} b\tilde{h}\tilde{@}_b \leftrightarrow \tilde{b}h@_b$$

$$@_c: \quad c\tilde{h} + \tilde{@}_c \mid \tilde{c}h + @_c \stackrel{\delta}{\Longrightarrow} c\tilde{h}\tilde{@}_c \leftrightarrow \tilde{c}h@_c$$

Block at x *and not at* y: $x + \tilde{y} \mid \tilde{x} + y \stackrel{\delta}{\Longrightarrow} x\tilde{y} \leftrightarrow \tilde{x}y$. Eg. $a\tilde{b} \leftrightarrow \tilde{a}b$.

In the bubble-up phase, changes to (say) x are picked up by its co-exclusions xy, $xz\ldots$ and sent further up the hierarchy. This applies to sensors, meta-sensors, and meta-effectors. See the accompanying *TLinda* code in the Appendix.

The problem we pose - to both motivate and illustrate the goal generation algorithm - is a goal to move a *Block* from Place A to Place C. That is, we issue the impulse to fill Place C, $[\uparrow, \tilde{c}, c]$. *Let us assume that besides the above actions, the system S has learned the actions* $a\tilde{b} \leftrightarrow \tilde{a}b$ *and* $b\tilde{c} \leftrightarrow \tilde{b}c$, *but not yet the action* $a\tilde{c} \leftrightarrow \tilde{a}c$, so to accomplish $[!, \tilde{c}, c]$ it must first move the block from a to b, and thence from b to c.

Thus if the corresponding goal simply echoes the impulse, which will be a goal on c, namely $[!, \tilde{c}, c]$, there is *no explicit connection* between c back to a (where the block currently is), and therefore no basis for issuing a subgoal to empty a (nor the latter's natural subsequents). Thus although all the necessary machinery is present, the goal will not be fulfilled. The present algorithm solves this problem, which is of a *very* general nature, and does so very efficiently.

6.4 PROBLEM SOLVED

Propagating the initial impulse and translating it into various goals requires establishing the missing connection(s) between the current state and the goal state. This has both a horizontal and a vertical aspect.

The *vertical* aspect is the propagation of environmental impulses for change, [↑, X, NotX], upwards in the hierarchy, which *bubbling up* is a crucial aspect of space-like computation's efficiency, since this reduces the search space logarithmically. These up-bubbling impulses are ultimately translated into goals that "tree out" as they *trickle down* and *retrace the trails* of the impulses as they bubbled up. In so doing, they intersect the horizontal aspect, as follows.

The *horizontal* aspect of impulse propagation is a distributed version of *back-chaining*. Instead of thinking current-state-to-goal-state, see the problem the other way around, as chaining *from the goal-state* backwards to the current-state, and then use these (causal) paths that connect the two.[18] In either direction, of course, the possibilities tree out exponentially, so it is critical to keep this explosion from impacting efficiency. The trick is to *use the intersection of the the vertical and horizontal trees.*

Every level of the Action-hierarchy corresponds to such a "horizontal" plane. Furthermore, as one ascends upwards in the hierarchy, a level's Actions' effective grade increases (~doubles) at each step, and the computation's reasoning becomes correspondingly and increasingly global in scope and consideration.

When an Action *Me=XY* notices an impulse [↑, X, NotX], where X is one of *Me*'s two constituent parent nodes, and *Me* is not *Grounded* (aka. not *Relevant* ... can't 'fire' in its current state), it issues an impulse [↑, Y, NotY] that asks, Is there any Action (on this level) that flips *(*X*'s partner)* Y as part of its co-exclusion (because then it can flip X by itself)? Every Action has a thread that namely looks for these backlinking impulses, and finding one, adds itself to the chain in the same way: this is the back-chaining.

This continues (in concurrent distributed fashion) until the chain reaches Actions that *are* grounded, which means that the Action in question could (if allowed) initiate the chain of effector activations that would eventually reach the instigating Action [Me, X, NotX], because the current

18 Cf. MetaEffector Thread 2 – Accept Back-chain Triggers *to* Me.

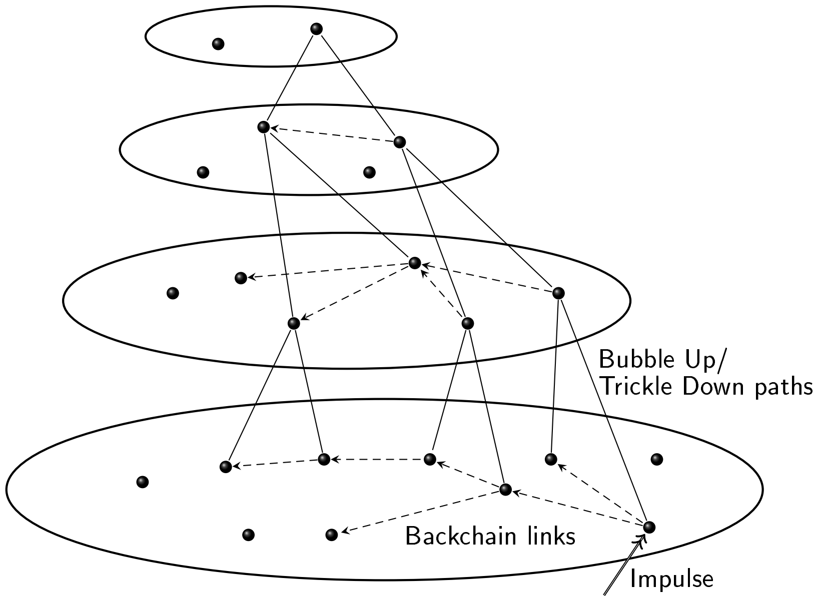
state of the environment matches the state required to (probably) succeed. [19] The algorithm, being a coordination algorithm, avoids creating unnecessary copies of things, eg. in the case of overlapping or out-dated chains, by storing such state information as *thread* state (together with the surround) rather than *memory* state, as is typical of conventional computations.

However, this back-chaining process does namely *not* allow such chains to begin execution yet, because there can be many such chains - all possible routes from B to A - which if all executed at once would create much wasted effort or worse. Rather, the backward chains - in every level on the way up - are kept in abeyance until *grounded* higher- or top-level nodes issue the required goal(s). These goals then trickle down, tree-ing out, and any grounded Actions that they meet are only then allowed to activate their parents' (meta-)effectors. The hierarchical stacking of the chains ensures that sub-goals are emitted in a coherent order (and if not, then learning will eventually remedy this).

An impulse or back-chain that has reached the Action *Me*, and *Me* is not grounded, results in both the further propagation of the back-chain and the bubbling up of the impulse, this latter in the form of a new impulse, this time on *Me* itself: *MeImpulse = ['^', XY, -XY].* This in turn triggers back-chaining on the next level. When a *top node* - one with no child - is reached, the code (as shown) just turns the impulse into the requested goal and sends it down. Clearly this should in many cases be expanded into some more nuanced decision, probably including humans.

It seems an obvious optimization to note the Action-pairs that arise as a back-chain is transitted forwards. We use the tuple ["D;",AtoB] to indicate that the two Child actions A, B are to be executed in strict order: first A then B, written $A; B$. Indeed, increment a counter each time this pair is used, and use this to prioritize alternative plans! Clearly, this will eventually create long chains of "good" plans, and hierarchies thereof. A further optimization! *Unfortunately*, this solves a non-existent problem, as any given pair will *always* be chosen in the same situation anyway, and the choice algorithm is *very* efficient. So nothing is gained. Furthermore, the flexibility provided by the space-like hierarchy - which gathers *all* the relevant aspects of a situation - is usurped by a myopic, less flexible, and possibly out-dated time-like solution. Nevertheless, this functionality is indicated in the attached *TLinda* code (but commented out).

19 Reality requires a more limiting criterion, eg. a chain-length threshold, but in the interests of clarity and brevity, we choose not to clutter the code with such details.



A "figurative" illustration of the backchain, bubble up, and trickle down paths caused by an initiating impulse, The network reconfigures itself automatically as goals are fulfilled and new impulse configurations arise. It is tacitly assumed that there are MANY, perhaps competing, impulses and goals present.

There can, in principle, also be impulses, goals, actions, etc. for *Relevance* and even *goals*. However, such generalizations are semantically very wild, and as well mathematically fraught.

Geographically distant Actions, and their Child and Parent nodes, are connected by a subscription mechanism that automatically propagates given state changes between such nodes, whether these be sensors, impulses, goals, effectors, or control tuples eg. *Relevant*.

Although the concept of "recalling something from memory" *per se* is an implicit part of the operation of a space-like computation, it is nevertheless useful for users to be able to simply request some particular computation or fact. The former is largely the province of *impulse* and *goal* propagation, as described; but the latter - "fact" recall - needs a few more words.

To recall a "fact" means to generate an *impulse* for a particular state, such that the up-bubbling *Relevance* indicators are the focus of a devoted Event Window, wherein the desired information is to be found.

Discovery - finding novel processes: by using the contents of an Event Window that is tied to a *recall* operation, a space-like computation can insert related subgoals to be carried out simultaneously. With trial and error/experimentation, the lacking concepts/nodes will be knitted together. One can also pursue this problem from the Fourier side, by having as a goal the generation of a particular resonance.

7 SECURITY

Because a space-like computation derives from, and is *reactive* with, its environment, an external process can only approach it on the former's own terms. Otherwise the space-like computation will not react at all. This is the basis for securing the integrity of a space-like computation relative to its environment.

Relative to a space-like computation's interior, individual communications (namely individual tuples in TS) with other entities can be encrypted. Encrypted or not, such communications are so microscopic in their scope and atomic in their content - typically just a ± 1 value and an anonymous tag - as to be of little value anyway. It is only in the *dynamic event structure* erected and maintained by the space-like computation that actual *meaning* lies.

7.1. The basic principle is that information about a space-like computation - call it S - may only be obtained via *interaction*. Since S *creates* (internal) information through its interaction with its *surround* (cf. Bubble Up), which surround here is an interrogating process $P_?$, the latter will never get more information from S than S gets from $P_?$. Only if $P_?$ is a space-like computation like S can the information exchange be equal.

7.2. Thus information (from interaction) Bubbles Up inside S before Trickling Down in the form of a *reaction* (in the shared environment) that it is up to $P_?$ to perceive. Such latter perceptions are then necessarily $P_?$'s *only* source of information about S.

7.3. In particular, the *names* of the Actions (whose hierarchical arrangement constitutes S) are in principal unknown to $P_?$, since only *behavior* ("motion") is externally visible. Lacking these names, $P_?$ cannot hope to penetrate S. To ensure, additionally, that Actions remain anonymous relative to each other within S, an Action's name N_A is created from its constituent boundaries B_1 and B_2 as $N_A = f(B_1, B_2)$ where f is a so-called trap-door function that loses information, whence B_1 and B_2

cannot be derived from N_A. [We suggest that $f = xor$, a common and efficient hardware operation. Choosing f to be an encryption (which is by nature reversible) would constitute a security weakness.]

In this way, only the Action itself knows its parent nodes, and no other Action can, because the names B_1 and B_2 are only known within the Action, and the names B_1 and B_2 were themselves earlier created in the same way. And this is true of all Actions. Indeed, with little overhead, the B_i can be assigned new names every time they are co-excluded. Giving the sensors constituting S's boundary with its surround *random* names then ensures, topologically, a basic inscrutable defensive perimeter. Such code-naming strategies combined with the formal power of the underlying (Clifford) algebraic semantics endows space-like computations with uncommonly strong and fundamental architectural support for building stable and secure systems.

This is especially comforting when one recalls that space-like computations are self-organizing, whence all this security occurs automatically, not least because all Actions obey the same interaction protocols. An Action that doesn't obey these will have a very hard time interacting with any other Actions because it doesn't know any of their names.

7.4. As a further measure against rogue *TLinda* code, each tuple is to be equipped with one or more special bits that will deny-match-to-unbound-fields. That is, we extend the standard minimal *Linda* tuple match (namely on typed fields) such that "wild card" matches are not allowed on designated fields and/or tuples. This simple measure short-circuits "fishing expeditions".

7.5. $P_?$ causes S to react by issuing an *impulse* $[\uparrow, s \to s']$ that requests that S change the state of (environmentally shared) s to s'. This impulse Bubbles Up (§6) until it meets an Action (one or more) that can satisfy it, which results in the issuing of a corresponding goal (each), which Trickles Down, ultimately resulting in the effector goal $[!, s \to s']$.

The *transition* $s \to s'$ is the only thing that $P_?$ sees. Exceptionally, S may advertise a higher-level Action (ie. N_A vs. the sensory boundary $\{s\}$) as available to external impulse. However, since Action N_A *is* the goal-fulfiller, the impulse does not propagate further up the hierarchy, thus implicitly limiting $P_?$'s access to S's interior to what S allows, and no more. One can further tag such impulses as *external* and limit or forbid collateral activations accordingly.

8 MISCELLANEOUS

8.1. An *avatar* is a software structure that represents a (usually human) entity to the remainder of a computer system. Most commonly, people employ avatars to represent them in video and on-line games, but the concept is very general, both for the user (who has great flexibility in choosing their representation) and the system designer (who can design a safe, robust, general purpose and generic avatar interface that is used by all).

Such an avatar structure is realized in a space-like computation via a suite of system-supplied Actions whose sensors and effectors *are* the virtual world to which it is interfacing, and whose motivations are rooted in a space-like computation that is being inhabited by the user (who is passing impulses up to its avatar suite, and awaiting and decorating these freely). Avatars can, whole or in part, be driven by software, but *must always be anchored to a human* individual, so there is someone to whom responsibility for its actions can be assigned.

8.2. Goal-driven processes can *deadlock* or *livelock*, both which lock a set of processes in a permanent closed cyclic pact. Via Parseval's Identity, one can view the operation of a space-like computation in the wave domain. "Dead-spots", "lines" or some other characteristic pattern in the spectrum of the computation will then *indicate* the presence of a deadlock or livelock, and as well *identify* the Actions participating in the lock.

The lock can be broken by *temporarily* removing (or otherwise silencing) the goals or impulses involved, and then restablishing same (eg. according to established principles of contemporary operating system design) after a fitting period of time. Since most such locks are "unlucky", they will likely not re-appear after such an interruption. A subtler method is to exploit back-chains that circle back to their originator. This condition can be sensed by a dedicated thread and used to trigger recovery.

8.3 The mental mechanism that focuses on the most current events is called short term memory (STM), and is subsumed by, and therewith generalized by, the event window mechanism.

REFERENCES

1. USPTO #14/732,590, filed 2015: Space-like Computation for Computational Engines

2. Manthey, M.J. 1994 patent, on Co-Exclusion (expired)

3. Manthey, M.J. 1998 patent, on Event Windows (expired)

4. See Appendix.

5. Bostrom, Nick. *Super-intelligence - Paths, Dangers, Strategies*. Oxford University Press, 2014.

6. Gelernter, David & Carriero, Nicholas. wikipedia.org/wiki/Linda_Programming_System

7. en.wikipedia.org/wiki/Subsumption_architecture

8. Manthey, M. "Awareness Lies Outside of Turing's Box", Proc. ANPA 2018.

APPENDIX: TLinda Source Code[20]

NOTATION.

A Tuple Name begins with Uppercase, Xxx, and is defined with square brackets: Xxx = [...].

A tuple has some number of typed fields, for example Xxx = [*integer* n, *string* "...", *tuple* Yyy].

Two tuples MATCH if all their fields match in both type and content.

A ? indicates a wild-card that matches any value (but the type must match). For example: [3, T, "foo"] matches [?n, T, ?who].

A wild card variable ?... becomes bound to the matched field-value; so from above, n=3 and who = "foo".

The tuple operations In, Rd, Co etc. have boolean-predicate versions Rdp U,...,V, Inp U,...,V and Cop U,...,V . These are "one-shot" test operators. Thus Rdp, Inp and Cop return False and do not block if U,...,V aren't present.

An Action is defined by the tuple ['D',SomeState,OppositeState] and is a set of MetaSensor and MetaEffector threads.

Goal tuples look like ['!', OldState,NewState]. Impulse tuples look like ['^',OldState,NewState].

All communication between threads is via the Tuple Space operations Rd, In, and Out.

Threads are never de-allocated. Rather, if a thread's execution has obviated its further existence, its further presence will be used to host system-maintenance activities, eg. updating the graphical interface.

```
thread StartUp()                          Boot system up

begin

. Eval Sensor( ...first... ) ...
. Eval Sensor( ...last.... )            -- Instantiate the sensors
. Eval Effector( ...first... ) ...      – and
. Eval Effector( ...last.... )          -- Effectors.
. Eval Corm(1)                              -- Instantiate the initial Corm
. Eval UserInterface(...)               -- Fire up graphical interface.
```

20 The more pedestrian modules are inherited from the original implementation and are correct, whereas the new pieces (Corm, MetaSensor, MetaEffector), which constitute both the novelty and core of the invention, have never run, and cannot be guaranteed to be bug-free. The "connecting tissue" for these modules is to be inferred from the respective parameter lists plus tuple space. Apologies for various loose ends.

```
. forever
. ...
. loop                                  -- Service graphical interface
end StartUp
```

The code for a standard one-bit ±1 sensor:

```
1 thread Sensor(X,Name,Bag)        – X is a raw sensor tuple.
.   –"Bag" is a user-defined sensor-category, cf EW's.

2 own
3   Flag = ['L',1,0]                –This is a Screen#1, 0-level sensor.
4   Plus = [Flag,[X,+],Bag]                –Sensor's
5   Minus= [Flag,[X,-],Bag]               – phases; Goals on X:
6   PlusGoal = ['!',Minus,Plus]           –   Minus to Plus,
7   MinusGoal = ['!',Plus,Minus]          –   Plus to Minus

8     MinusBubble = ['^',Plus,Minus] – Flip new Plus to previous Minus

9     PlusBubble = ['^',Minus,Plus] – Flip new Minus to previous Plus

10  begin

11    opposite Plus, Minus        – Short-circuit EW hit.
12  Out Minus                   – Sensor initially off = X is not present

13    Rd X                        – Block til X shows up, presence = '+' and absence =
'-'.
14  forever
15     In Minus                 – Retract Off.
16     Out Plus                 – Indicate On.
17     AntiRd X                 – Wait for change.
18     In Plus                  – Retract On.
19     Inp PlusBubble
20     Out Minus                – Indicate Off.
21     Out PlusBubble          – Go back to Plus
22     Rd X                       – Wait for change.
23     In PlusBubble
24     Out MinusBubble
25  loop
26 end Sensor
```

The above *Tlinda* for Sensor (and other later code) is been cleared of administrative detail. In principle (and beta-implementation fact), all sensors run just this thread.

Similarly, *effectors* - entities that cause an associated sensor to change - all consist of the following two threads. The first of these, just below, establishes the *grounding/relevance*[21] of the effector to the current state of the system's surround. For example, if the system's Hand is Empty then it

21 Terminology: *relevance* has been replaced by *grounded*, as more precise and evocative.

can Grasp an object, but in this very state it cannot logically *also* Release an object because there *is* none in the Hand. So the Hand is *relevant* (flag "R") for Grasp if it is Empty and irrelevant (flag "r") if not. The goal for Grasp is namely ['!',Minus,Plus], meaning to cause the Empty Hand to be non-Empty. The code below is independent of the particular ±values assigned to S and NotS - they need only be opposites.

```
1 object Make_Effector – Implements effector relevance and physical effect.
2   thread Effector(S,NotS,X)        – X is the raw/physical effector id.
.                                  – S/NotS = corres internal-sensor states
3  own
4   HereIAm = ['D',S,NotS]                        – Advertise ourselves.
5   Rel = ['R',S,NotS]                            – X's relevance
6   IrRel = ['r',S,NotS]                          – and lack thereof.
7   external
8   HoldS = ['!',S,S]                 – => don't change S => Irrelevance
9   begin
10  spawn Effect(S,NotS,X)            – Start partner thread (below)
11  Out HereIAm                       – Advertise ability S –> NotS.
12  Out IrRel                         – Initially irrelevant.
– The uninhibited use of Inp and Outp below is exceptional!
13   forever                             –Establish current Ir/Relevance, 4 cases:
14     if Cop S, HoldS then                – Both S & HoldS present.
15       Inp Rel
16       Outp IrRel                     – Show Irrelevance.
17       AntiRd HoldS                   – Block till HoldS disappears.
18     end
19     if Cop NotS, HoldS then            – No S, but HoldS present.
20       Inp Rel
21       Outp IrRel                     – Show Irrelevance.
22       NotCo NotS, HoldS              – Block till one is gone.
23     end
24     if Rdp S and not Rdp HoldS then          – No HoldS, but S present.
25       Inp IrRel
26       Outp Rel                                – Show Relevance.
27       AntiCo S,HoldS                          – Block till both are gone.
28     end
29     if not Rdp NotS and not Rdp HoldS then   – Neither.
30       Inp Rel
31       Outp IrRel                              – Show Irrelevance.
32       Rd S – Block on S.
34     end
35   loop
36 end Effector
```

The second effector thread, below, is tied to the preceding thread by the latter's Ir/Relevance tuples (lines 5,6; 26). Line 43 is the nub of the matter: Co Rel, TriggeringGoal, S ties the relevance of the effector, Rel, to the goal of inverting its state, TriggeringGoal, by insisting on their simultaneous presence. [The additional presence of S avoids an obscure possibility that I leave to the curious reader to identify.] When this occurs, a command is output (line 44) to the physical effector to accomplish the specified state change. Line 45 then blocks until line 43's Co is broken by either S changing (ie. the goal was successfully achieved) or the disappearance of the inciting goal (line 46) which results in an attempt to retract the command of line 44.

```
37 thread Effect(S,NotS,X)            – Propagates S —> NotS to physical effector.

38   external
39   TriggeringGoal = ['!',S,NotS]
40   Rel = ['R',S,NotS]

41   begin
42   forever
43     Co Rel, TriggeringGoal, S                         – Ready and wanted?
44     Output "(E:", &X, ")"                             – Yes ... achieve S.
45     NotCo S,TriggeringGoal                            – Wait till obviated.
46     if Rdp S then Output "(E:",&X," Oops)"            – (Retract request X)
47   loop

48   end Effect;

49 end Make_Effector
```

The above code is the bottom level of our ultimate goal system. All it really does is insulate the rest of the system from X's realities, offering instead a clean internal interface of idealized Sensors and Effectors.

===============

The self-organizing aspect is inspired by the Coin Demonstration, especially the fact that there are two co-exclusions, $a + \tilde{b} \mid \tilde{a} + b$ and $a + b \mid \tilde{a} + \tilde{b}$, both of which allow the instantiation of a *new* action, ab. The obvious implementation, good for either case, is

```
.          Co A,B
.          Co NotA,NotB
.          eval Action(A,B,NotA,NotB)
```

But this approach must unfortunately specify which of the two co's is to come first, and much worse, must specify exactly *which* tuples A and B are to be watched to (maybe) exhibit a co-exclusive relationship. Since for n sensors this implies a minimum of $\mathcal{O}(2^n)$ threads, this approach is clearly not workable.

Instead, we introduce the idea of an *event window* (*EW*) of size Δt time units. Into this window are put (selected) tuple *changes*, with the result that, to a resolution of Δt, every tuple in the window has changed its value *simultaneously*. Furthermore, since (say) A and B just changed value, then *before* they did so they must have formed the co-occurrence $\tilde{A} + \tilde{B}$. Thus every pair of tuples in the window is *automatically* a co-exclusion! A tuple ages, and is discarded when its residence time has reached Δt, and if it is changing faster than Δt, only the latest version is kept.[22] The *opposite* command (Line 9 in thread sensor above) prevents a given co-exclusion from forming more than one action instance.[23]

Thus, the following Tlinda code implements the creation of actions based on observed sensory input from the surround:

```
  thread DiscoverActions(DeltaT)
. X: EW = [...]
. forever
.   Rd X(A,B)
.   eval MakeAction(A,B...)
. loop
```

That is, every time event window X finds a co-exclusion, it spawns a thread, MakeAction, that will instantiate the further multiple threads (below) that consitute an *action* before settling down to an unending existence as the action's GUI hook. In this way, each of the two co-exclusions is instantiated as an independent action, these being mutually-exclusive dynamically via their *relevance*.

Also, each such action has a public face similar to our earlier sensor/-effector pairs - the $\pm$ *orientation* of ab (aka. its *spin*) translates to Tlinda as a (meta-)sensor that is in every way equivalent to a primitive sensor.

---

22 If the sampling frequency f were 50ms, then $\frac{1}{f} = \Delta t$ would be 20ms. Δt is the uncertainty/"Planck" constant for this window.

23 It's not difficult to construct a situation where despite appearances only one of two sensors in a pair *actually* changed, ie. the EW really only has a state and its conjugate. One can worry about this, or just instantiate both co-exclusions, the latter being the more realistic resolution.

That is, the instantiation of an action adds *new* sensors to tuple space, which in turn can find themselves in event windows and forming even higher-level actions. The actions so formed *all* use this same Tlinda code.

Thus, with very little "putting in by hand", Topsy automatically erects a coherent and interconnected hierarchy of wave-like actions. These are themselves built out of mutual exclusions, with the result that the actions (ie. threads) able to be active at any given point in time *by definition* cannot violate mutex requirements. That is, the execution regime implicitly supports flat-out concurrency.[24]

Here is the code that instantiates an action:

```
thread MakeAction(Parent1,Parent2,Lvl,NewGrade)
. own NewAction = [Parent1,Parent2,Lvl,NewGrade];
begin
. Inp [Parent1,"I'm on Top"];
. Inp [Parent2,"I'm on Top"];
. Out [NewAction,"I'm on Top"];
. eval MakePlusMinusTupleSensor(NewAction);      -- Make NewAction's Sensor
. eval Object MetaSensor threads;                  -- Start up all the
. eval Object Bubble_Impulses_Up threads;         -- threads that make
. eval Object Trickle_Goals_Down threads;          -- up an Action. etc.
...
end MakeAction;
------------------------------------------------------------------------------
  --   META-SENSOR THREADS      Bubble sensory data up      --
  ----------------------------------------------------------------------------
  object MetaSensors                          NB: "Relevant" == "grounded".
     -- An Action multi-instantiates these to manage its state, especially
     -- as seen by other threads (cuz broadcasted via tuple space).
thread MetaSensor(S1,S2,State)
  begin                          --Show MetaSensor's State based on S1+S2.
  . forever
  .   Co S1,S2;                               --S1+S2 present
  .   Out State;                              -- => show State.
  .   NotCo S1, S2;                           --S1+S2 not present,
  .   In State;                               -- => remove State.
  . loop;
  end MetaSensor;
thread MetaSensorRel(S1,S2,S1Rel,S2Rel,On,Off)
                 --Shows On if S1 + S2 + (S1Rel | S2Rel) are present, else Off.
  begin
  . forever
  .   if Cop S1, S2 And Not AntiCop S1Rel, S2Rel then
  .   Out On;                                     --Show On.
  .   NotCo S1, S2, S1Rel;                       --Await.
  .   NotCo S1, S2, S2Rel;
  .   In On;                                       --Retract On.
  .   else Co S1, S2;                             --Await.
```

24 Tlinda's underlying thread management system cannot deadlock, but parts of the action hierarchy can, reflecting conflicting user-demands (see below).

```
.    AntiNotCo S1Rel, S2Rel;
.  end;
loop;
end MetaSensorRel;
3. Meta-Effector threads: Bubble Impulses Up & Trickle Goals Down
-- This code does (X, Y) <--> (-X,-Y)    Same      2x2 instances = one-half Action
--                OR  (X,-Y) <--> (-X, Y)    Code              2x2 for this half too.
object Bubble_Impulses_Up                  -- Bubbles Impulses up and Back-chains
.                                               -- lower level to Current/Rel states
.                                             -- "Sub" is one (=X) of the 2 parents
.                                              -- of action XY => This thread x 2,
x 2.
external
.  Sub = ['S', +/-, X],                        -- Current value of meta-sensor S.
.  SubImpulse = ['^', X, -X] ,               -- Impulse to invert S.
.  MeOnTop = [Me,"I'm On Top"] ;                -- Me is Top of (its part of the)
hierarchy...
own
.  SubGoal = ['!', X, -X],
.  MeImpulse = ['^', XY, -XY], MeRel = ['R', XY, -XY],
.  MeRel = ['R', XY, -XY]                       -- Ie. GROUNDED
.  Merel = ['r', XY, -XY],                      -- Ie. NOT GROUNDED
.  MeGoal = ['!', XY, -XY],
.  BackLink = ['->', ?Predecessor, Me] ;     -- Matches ANY predecessor.

begin        --        Thread 1:                    -- Accept Impulse from Subaltern

.  forever
.    Co Sub, SubImpulse                           -- Await need DIRECTLY from
below.
.    if Cop MeRel, MeGoal then                     -- I can do this directly:
.      Out SubGoal                                    -- Issue the sub-goal,
.      NotCo Sub, SubImpulse, MeRel, MeGoal      -- await developments,
.      In SubGoal                                     -- and retract sub-goal.
.  else
.    Out MeImpulse                                 -- Bubble Impulse further up.
.    Out BackLink                                    -- Originate back-chain.

.    NotCo Sub, SubImpulse                        -- When need disappears,
.    In MeImpulse                                    -- remove impulse for Me
.    In BackLink                                      -- and retract back-chain.
.  loop
--thread 2:          -- Accept Back-chain Triggers to Me = this Action
   -- NB: The initial Rd binds just ONE Successor; any others piggy-back
   --   on the Impulse it generates and back-chain from here automatically.
.  forever
.    Rd ['->', Me, ?Successor]                  -- Some successor seeks Me.
.    if Rdp Merel Then                              -- We're not relevant...mebbe fix
this:
.      Out ['^', NotMe, Me]                     -- Trigger an Impulse Bubble for Me.
.      Out ['->', ?Predecessor, Me]             -- Extend chain horizontally.
.      AntiRd ['->', Me, ?Successor]            -- Wait for all chain requests to
.      Inp ['^', NotMe, Me]                     -- disappear, then retract Me-bubble
.      In ['->', ?Predecessor, Me]              -- and chain extension.
.  loop
-- thread 3:          -- React to goals from above
.  Forever
```

```
.    Rd ['->', Me, ?Successor]              -- IF some successor seeks Me, AND
.    Co MeRel, MeGoal                        -- If Grounded & Goaled, reverse
.    Out SubGoal                             -- the chain & start its forward-execution.
--  Out ['D;', Me, Successor]               -- Record TIME-LIKE action Me;Successor
.    NotCo Sub, SubImpulse, MeRel, MeGoal   -- Await developments,
.    In SubGoal                                            -- and retract goal.

. loop
--thread 4:                   -- What to do at the Top
. forever
.    Co MeImpulse, MeOnTop            -- If I'm at the (local) hierarchical top
.    Out MeGoal                          -- Convert the Impulse to a Goal
.    NotCo MeImpulse, MeOnTop      -- NB: not MeOnTop drops the ball,
.    In MeGoal                           -- or at least causes a hiccup.
. loop                                   -- This is also the place for human control.

                              -- -- -- --

object Trickle_Goals_Down(X)           --"Sub" is one (X) of the (two) constituents
                                        -- of action XY => These threads x 2.

external
. SubImpulse = ['^', X, -X],             -- Impulse from below to flip X to -X
. XRel = ['R', X, -X],                       -- Action X --> -X is grounded
. Xrel = ['r', X, -X],                       --  is NOT grounded
. MyRel = ['R', XY, -XY] ;                  -- Action XY --> -XY is grounded
own

. SubGoal = ['!', X, -X],                   -- Goal to flip X

. MeGoal = ['!', XY, -XY], ;                -- Goal to flip Me = XY "from above"
-- Thread 1:                Issue Goal on Sub
. forever
.    Rd MeGoal                                -- Wait til we're wanted.
.    Co XRel, SubImpulse                    -- When Sub=Rel & Sub desire still there,
.    if Rdp MeGoal then                       -- (and MeGoal still present)
.      Out SubGoal                             -- issue goal on Sub.
.      NotCo MeGoal, XRel                    -- Await developments.
.     In SubGoal                            -- Whatever happened, SubGoal is obviated.
. loop


thread Corm(Lvl)
--     This code relies on the TLinda run-time system to ensure
--     only ONE instance of any given Corm (cf. parameter Lvl).
. Own CormEvtWin = [...Lvl, grade={1,2,3},...]; --Define ONE EW for entire Lvl
begin
. forever
.    Rd CormEvtWin(...); -- A Hit defines Parent1 and Parent2
.    NewGrade = Parent1.grade + Parent2.grade; -- Calc NewAction's grade
.    if NewGrade = 4 and (Parent1.grade=1 or Parent2.grade=1)
.        then skip;                                          -- Disallow 3+1 Actions
.    if ShareNodes(Parent1,Parent2) then skip;      -- No shared parents
.    else if NewGrade <=3 then
.       NewAction = [Parent1,Parent2,Lvl,NewGrade]
.       eval MakeAction(NewAction);                    -- Make new Action at same Lvl
.    else
```

```
.       if NewGrade = 4 or NewGrade = 5 then NewGrade = 1; -- mod4, mod5 thing
.       if NewGrade = 6 then NewGrade = 2;
.       NewAction = [Parent1, Parent2, Lvl+1, NewGrade];
.       eval MakeAction(NewAction);                         -- Make new Action for
Lvl+1
.       eval Corm(Lvl+1); --Make Corm for (new) Lvl+1 – system prevents duplicates
.       inp(Parent1,MeOnTop); inp(Parent2,MeOnTop); – Parents lose top spot
.       out(NewAction,MeOnTop)                  -- Child = NewAction is the new boss!
    loop
```