TANGLED DESIGN HIERARCHIES

Hacking the Essence of Software

ENCODING AND DECODING TURING MACHINES AND META-MACHINES

Kent D. Palmer, Ph.D.

Orange CA 92856 USA
714-633-9508
kent@palmer.name

Started 6/9/2012; Version 0.13; 8/12/2012; th01a13.doc
Corrected 20121009, Added to 20121013

Keywords: domain specific languages, consistency, completeness, clarity, verifiability, validity, coherence, comprehensiveness

Problem Statement

Years ago in Wild Software Meta-systems I created a Design Domain Specific Language which went against the grain of UML/SysML graphical representations for Software Design. Then much later I did my second Ph.D. dissertation on Emergent Design, and took what I learned back to my languages and improved them, but the number of statements in the language went from 700 some to 1700 some statements. And while I showed the consistency of the original language via various diagrams in the original definition, I did not know how to show the new version of the languages was self-consistent. I have been looking for a way to do this ever since. Design languages are not closed like General Programming Languages and so the same techniques that suffice to show their consistency does not suffice for this new class of open and extensible languages that could exist at the design level as Design Domain specific languages.

This paper explores a new way to validate the Design DSLs of the ISEM language. It uses tangled Hierarchies first seen in the InteGreat tool from eDevTech, but it goes on to look at the relation between the State Machine, Petri Net, Turing Machine and the addition to that of the Capsule as a way of looking at the synthesis of the computing infrastructure underlying the tangled hierarchies. These two approaches are then reconciled though the various ways of looking at the minimal system that were developed in my Emergent Design dissertation.

Recently inadvertently on my search for a way to validate the ISEM Design Domain Specific languages I ran into a tool called InteGreat created by Asif Sharif which showed me a possible path forward in solving this problem for which I could find no solution in the literature. This tool has tangled semantic hierarchies. Basically what you do is create various hierarchies of different types related to Business Analysis and then you can drag and drop these into hierarchies that establish relations between analytical elements from those separate hierarchies. This establishes semantic relations between the elements that are related to each other in the tangled hierarchy that

represents the composite model. I was given a briefing by Asif Sharif of eDev Tech concerning the semantic structure of the tool, which is not featured but is treated as a background resource in the tool which gives it a lot of representational power that is drawn upon to support document generation, querying, and simulation capabilities. He expressed the wish to take his tool to the next level of representational power and I mentioned that this would probably be best expressed in the tradition by the work of Charles Peirce in the idea of Thirds as expressed in the hierarchy of philosophical principles Firsts (isolate), Seconds (relata) and Thirds (continua). To these B. Fuller added Synergy (Fourths) and Integrity (Fifths). And we talked about the fact that the solution to this problem perhaps resides in the idea that tangled hierarchies of various types themselves could be tangled at the next level. However, on further thought this does not take into account that the next level is emergent and has its own properties and although it may be a tangling of tangled hierarchies it must also be more than that. And one of the things that is needed is to work out what that more could be based on the theory of Peirce and others as noted in my Emergent Design dissertation.

So the immediate interest here is to explore the relation between the idea of tangled hierarchies as a way of creating sematic models due to Asif Sharif and embodied in InteGreat, and the idea of Domain Specific Design Languages such as the new version of the Integral Software Engineering Methodology (ISEM) languages. To do this we need to give some theoretical background. From the beginning I have advocated using George Klir's Architecture of Systems Problem Solving (ASPS) as a basis for thinking about Software Design structures. And my own work on the relation between Systems and Meta-systems as well as General Schemas Theory comes directly out of attempting to work with the Formal Structural System of Klir to describe design possibilities. This grounding of Software Engineering in General Systems Theory I believe is a crucial move that few make, but it increases our leverage on the design problem space. So I take Klir's ASPS as the assumed background for everything that I am saying. And the crucial piece that I take from Klir is the Methodological Distinctions which relate to the ordering of background variables within our systems models. These orderings are: no order, partial order, partial order with distance and linear order without distance, full linear order with distance, and adding to those combinatoric orders. This forms a lattice in which linear order without distance and partial order with distance are duals within the lattice, while all other elements in the lattice are self-dual. What we note is that in Software Engineering we are progressively adding order to systems we design, and so we start off with requirements (functional and qualitative or performance related) which have no order. But then Agent and Function viewpoints are partially ordered. Data and Event viewpoints are fully ordered. Test is combinatorically ordered. Minimal methods are bridges between viewpoints. They are all conditioned by the duality between linear order without distance and partial order with distance. They are all either two way bridges or two one way bridges. So for instance there is a single one-way bridge between data and function, but if you look at it from the point of view of data you see object oriented paradigm, but if you look at it from the point of view of function you see dataflows. Between Agent and Data there is the DARTS method of representing parallelism of Gomma. Between Agent and Event there is the Worldview/Scenario sequence diagram that allows relativistic structuring of systems in the face of a lack of a global clock. Between Agent and Function there are however two bridges, on is the Virtual Layered Machine and the other is the Use Case. Between Event and Function there is the two bridges of State Machine and Petri Net. Between Data and Event there are the various ways of representing spacetime intervals which are either in terms of Riemann (spacetime) or Minkowski (timespace). For in computing Event is the representation of Time and Memory with Data is the representation of Space.

Now the whole purpose of Design Languages is to get above the delocalization that occurs in Code where objects are smeared around within the code in various places. Note that code is linear

and the distance between any given line of code and the next is a measure of performance. Note that relations between lines of code spread across agents or processors are partially ordered. If code threads are switched out by the operating system the relation between any one line of code and the next at execution time is indeterminate. But it is the performance given these partial orders that is significant. On the other hand individual lines of code although linearly ordered have no necessary measure between them. So the duality between these two orders in the lattice comes to play a very significant role with respect to delocalization of objects within the code and their representation and interaction during execution. Design attempts to take this delocalization out of play so we can see the design objects all in their set-like purity above the fray of the execution of the actual code on the hardware with its operating system. But as soon as we start executing the code as a binary then the code becomes a mass-like blob and essentially at that point we lose control and all kinds of unexpected things occur during execution. These unexpected side effects are mostly problems and so we consider them defects. But these defects occur because we cannot clearly relate the set-like designs to the mass-like executables in their actual operation on hardware and within operating systems (i.e. meta-systems). Many of these problems occur because in our tradition there is a set bias and mass-like phenomena are suppressed. But the reality is that sets and masses are duals, and each has their own logics, but these are not developed in our tradition. Thus in some way the duality between set/mass, or system/meta-system is only historical and cultural and is not ultimately real because masses have logics too, and meta-systems can be formalized using Turing machines just like systems/processes can. Basically in Design we are trying to get a view of what the structure of the system is prior to delocalization and mass-like executability constraints.

At this time our designs are for the most part by consensus of the Software Engineering community represented by UML and perhaps SysML or other similar representations. My concern with this is the poverty of the semantics which is based on entities with types and relations which for the most part are binary relations or via connections though is_a or has_a relations between objects. My own suggestion is that we construct a Tractatus like world of design facts as suggested by the early Wittgenstein. These facts are represented by statements because it was Frege that said that meaning is anchored in the whole statement rather than any of the parts whether they be words, or phrases, or the syntactic relations between these. These statements can be made regular by imposing a particular grammatical form that can express 80 percent of the necessary meaning for expressing design facts. We express this minimal grammar in columns in a spreadsheet so that it is easy to construct new fact statements and keep the grammar straight. The basic idea of a domain language is that you can create new domain specific statements on the fly to express what is needed to describe a particular domain or proscribe a particular design solution. As technology changes we need to continually adapt to new circumstances in terms of the technological infrastructure, and also we need to continually adapt to new contexts as we create new applications in new domains. Even recreating an application with new technologies, methods and practices demands new domain specific descriptions. But the problem is that if the languages are not closed then they cannot be compiled or interpreted, without changing the compiler or interpreter. The problem of how to make general purpose programing languages extensible to cover new kinds of domain statements or design constructs is still an open research problem. But this problem is exacerbated if we try to make the semantics richer than merely composing components or producing network relationships. Basically we need to be able to construct multipart relationships in single statements. And it is this that takes us into the realm of what Peirce called the Third, i.e. of mediation or continuity, beyond mere relationships. Since the entity is First, i.e. an isolatable element, and the Second is a relata, which is a connection between isolate, then the Third is a mediation, a third kind of thing that is neither an entity nor a relation, and these are represented by Peirce in his theory as signs. Basically Peirce is a Kantian who is attempting to deal with the dialectical mediations uncovered by Hegel that

were implicit in the categories of Kant and in Kant's critical method. Peirce was a logician and basically invented modern symbolic logic, but saw that in order to connect logic to the world it was necessary to develop a semiotic. Just like the isolata of the design entity is a point in geometry, and the relata of the relationship is a line, so the continua of the Third as a mediation is like a triangle, i.e. the two dimensional minimal solid. Thus, the sign is threefold composed of the entity, its interpretation, and the sign element. What we need to understand is that for Peirce it is logic that is the embodiment of the three fold relation of the continua. We see this in the syllogism. Peirce noticed that we can take the statements that are in the syllogism in different orders to represent deduction, induction and abduction. Abduction is the production of the hypothesis and thus the basis of the Scientific method. Thus for Peirce it is logic that is the motor of scientific progress. The combinatorics of Logic establishes as triangle of triangles. Two of the paths sport entailments and the third is a speculative projection. Both entailments and projections are continua. We connect the symbols of symbolic logic to the context of the statements via the semiotics which are themselves triangular connecting interpretants to objects via the sign that indicates. It is via preci**ss**ion that we understand the parts of the synthesis without taking it apart via analysis. And thus we recognize the articulation of the parts (isolata) within the embrace of the continua without the precision of analytical dispersion. Relata only really appear between the isolata, and fields are really not reducible to discrete relata, even if they are n-ary. We can see this in the compound statement which holds a field of meaning within a complex n-ary structure. It is a scandal that we have no real understanding of meaning, but that is due in part to the fact that it is a field like phenomena which is at least a third, as Peirce suggests. Field phenomena are mass like, and in this way it is like the attributes that span in a mass like way the isolata. When we think about it characteristics of attributes that span isolate is one bracket and the relata are equally definitive, but the field of the continua is again mass-like and is the other capstone which is just as obscure to us as the characteristics of attributes. The syntax is the structural articulation of the patterning of the words but the semantic lode reverberates in the sentence which when complex contains multiple clauses indicating multiple simultaneous relations. It is this ability to give sematic depth to our architectural structures that we lack when we use representations with only relata and no continua. The ISEM language allows prepositions to establish these multiple simultaneous relations. It is a pigeon language targeted to express domain relations in a restricted grammar.

But the problem is then to show that the statements of the language are comprehensive. We are not really expecting completeness because we expect new statements to be created on the fly to express new domain or architectural facts. We are not really expecting consistency because domains are bundles of points of view, and therefore there can be inconsistencies between different points of view. We are not really even expecting clarity, because one could depart from the grammatical schema if necessary when inventing new statements. Thus such a language is by its very nature para-complete, para-consistent and para-clear. Since it is not a closed language we cannot appeal to meaning as use either thus disappointing Wittgenstein. However, there may be a way to view this comprehensiveness of the language that takes into account the necessity of relating to the field of meaning produced by the continua seen as preci**ss**ion. A hint comes from Asif Sharif and the use of Tangled Hierarchies in InteGreat. This specific semantic structure has not been seen in any other tools to date by the author. Basically what occurs is that there is the creation of architectural hierarchies such as process (function), network (agent), data and event. These are then dropped into a shared hierarchy such that dropping one element below the other creates a different sematic relation. In other words if we drop hardware agents, then processes (functions), then data, then events, then that creates different meanings then if we interchange say data then functions, or events then data. The semantic relations are implicit given in pre-analyzed combinatorics of possible hierarchical positions. In a way this uses the combinatorics of Test to produce the possible semantics of combination of architectural elements. We postulate that it is

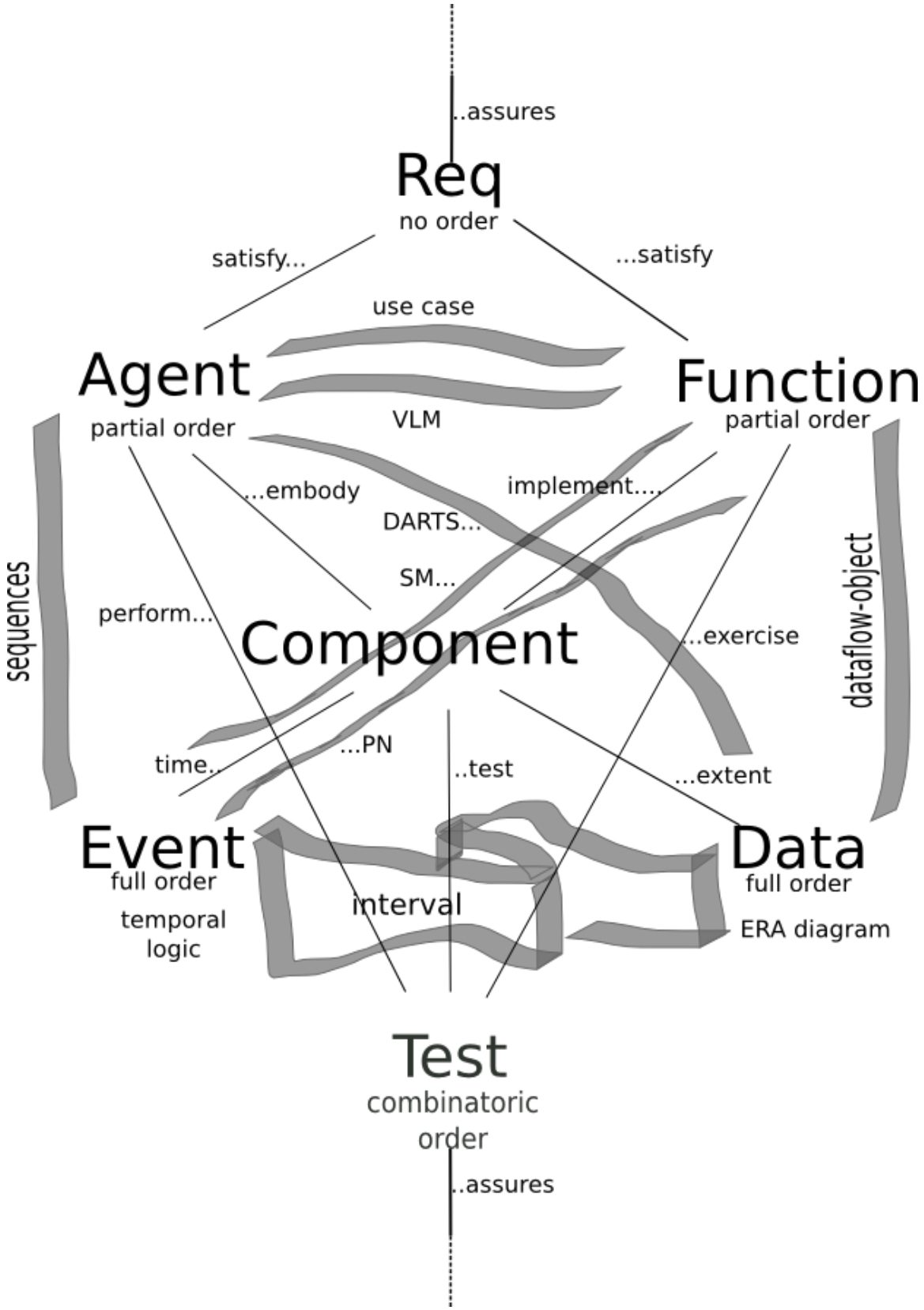
tangles of tangles that carry us to the next emergent level beyond mere relata of the sematic load that different combinations produce. However, this must be an emergent level because the various tangled hierarchies must themselves be different from each other, like the untangled hierarchies were. Somehow second order tangling needs to produce the continua not merely higher order relata. We posit that it is the second order tangling, tangling of tangles, must have a difference that makes a difference that is synthetic. When we close tangles we get rational knots. We have run into rational knots before in our definition of the structure of the worldview, so that makes us hopeful. There are in fact both rational and irrational knots, the difference being whether the knot can be reduced to a tangle with an equation for the cross-over moves. Continuous tangles that re-tangle are braids. So we posit that the difference between tangled hierarchies at the level of cross tangling is that there are created knot, which is itself the model of self-organization. Knots are organized against themselves. They are continuous yet discrete in their crossings. Thus we introduce continua, which at the same time has an inherent syntax that is fixed by the table of knots. So there is a mathematical basis for believing that the tangle of tanged hierarchies are in fact knotted hierarchies and so there is a higher order effect to take advantage of to get some leverage on continua. However, how to take advantage of this emergent effect in the mathematics is not an easy question.

Hypothesis

The hypothesis that we would like to advance is that just like in InteGreat there are hierarchies associated with the points of view: Requirements (no order), Function and Agent (partially ordered), Linear Order with out distance and Partial order with Distance are associated with minimal method duality and decoherence and delocalization. Event and Data (full order), and Test (combinatoric order which we add to the Methodological Distinctions of Klir). Requirements are isolated statements in natural language. Functional Hierarchy is a decomposition of functions. Agent Hierarchy is a decomposition of agents. Agents are associated with concurrence. Data is a heterarchy of entities depicted by entity relation attribute diagrams. Events are a heterarchy related by temporal logic. Test is a hierarchy of test cases. In the midst of this lattice of methodological distinctions there is a component hierarchy that represents the Architecture in which the viewpoint related elements are tangled. The basis of this component hierarchy is the network topology of the hardware upon which the software is layered. A system hierarchy includes both hardware and software elements. The software hierarchy takes the hardware computing infrastructure as given. Notice that this is a change from previous conceptualizations. Here component is not identical to the agent hierarchy as is normally assumed. Rather we are saying that the component hierarchy is the place where the tangling of the other hierarchies occur. The component hierarchy is a packaging of various relations between elements that appear from the various perspectives. The component hierarchy is composed of hardware and software components that are layered. So strictly speaking the component hierarchy is in the hole in the lattice while the test hierarchy is a kind of order not considered by Klir in his methodological distinctions.

Both components and test hierarchies are complex but in different ways. The component hierarchy is complex because it layers elements from the other views. The test hierarchy is complex due to combinatorics. But we still hold that these hierarchies are linked by crosslinks. So the *satisfies* crosslink connects requirements and functions. The *implements* crosslink connects to the component hierarchy. The component hierarchy connects by *test* crosslinks to the tests cases. The test cases connect by *assures* crosslinks back to the requirements hierarchy. Now previously we have been connecting the component hierarchy directly to the agent hierarchy. And this simplification could still be done as each agent could be considered a component. But this really belies the difference between modules and components because modules are mere packaging

constructs and not agents. A generalized component can be a module, i.e. just a packaging construct. This simplification makes things easier to explain but this asymmetry is really probably more exact and also justified. So we can see from this that there is probably also a mapping from requirements to agents and these are probably non-functional requirements related to performance and other Quality of Service issues. We can associate this agent hierarchy with performance budget hierarchy as the agents are the elements which are the embodiments that are measured with respect their performance. Components to the extent that they are just packaging really have no performance per se other than their gross running speed. It is different agent architectures that changes the performance. So this mapping is also a *satisfies* crosslink to non-functional requirements. However, the mapping to components we will call an *embodies* crosslink rather than an implements crosslink which comes from the function.



Once we introduce the *embodies* crosslink as the dual of the *implements* crosslink then we can ask

what the relation between them and the tests might be. Tests can either test performance or test functionality and so this split should be noted. Agent to Test is a *performs* link while Function to Test is an *exercises* link. Data and Event hierarchies can also connect via extent and time links. Extents of Data are represented by Entity Relation Attribute (ERA) Diagrams. Times of Events are captured by Interval Logic of James Allen or another Temporal Logic. Note in the diagram we show the various minimal methods that connect the viewpoints on a realtime system. Virtual Layered Machine and Use Cases connect Function and Agent that are both partially ordered. Two partially ordered systems together are called a Domain in mathematics. Sequence Diagrams in UML which we call Worldline and Scenario diagrams connect Agent and Event. DataFlow or Objects connect Function and Data. Gomma's DARTS connects Agent and Data. Between Data and Event is a relativistic Interval, which can be represented either as timespace (Minkowski) or spacetime (Riemann). If there is no global clock then realtime systems appear as relativistic in the sense developed by Einstein.

The key point here is that the structure of Software and even Systems architecture is driven by the possible orders. Orders are layered on as development proceeds. We start with no order and that appears as requirements statements that are axiomatic to the system. Then we move to partial ordering, and there are two types of partial orders Agent and Function. The relation between them can be seen as two one-way bridges. One is the use case, and the other is the virtual layered machine. Use case connects the agents to the functions in scenarios. User Stories are slices of use cases. But the virtual layered machine connects functions to agents. The VLM is the set of operations that would be used to implement the operations as functions within the agents as computational devices. For any given Use Case there should be a computational machine at a given level of abstraction that accomplishes the work needed to provide the capability to perform the functions requested by the user. Thus the VLM represents the computational capability to perform the functions that support the features needed by the agent. Capability and Features always go together, and they are what the customers and users see of the combination of agent and features which are both partially ordered in a domain. The partial ordering allows the flexibility to do multiple things in different orders and thus what makes the system under design useful. Since from Wittgenstein's point of view meaning is use, then how the agent uses the functions confers their meanings on them. The meaning of the system will come in its use in its actual operating environment. But meaning becomes possible because there are multiple ways that agents can use functions that are supported by virtual layered machines that supply the capabilities that result in features that the users and customers get benefit out of. Use Case is a particularly strong technique because it takes advantage of the dual partial ordering of agent and function to not only shape the system from the users perspective but also allows the system functionality to be developed in different orders according to convenience, and priority rather than internal dependencies. Where internal dependencies are satisfied then the order of development can be varied and still attain the same result. Similarly if we know a VLM is needed to supply the capability that will be used in different orders and with different inputs to supply the capability to implement the functions needed by the agents then the operators of that VLM can be done in different orders depending on convenience and priority. However, in both cases we must take care of the internal essential dependencies in order to avoid having to do so much refactoring which comes from violating development dependencies and precedents in the order that parts of the system (operations) are produced. In a sense this flexibility in the order of the production created by the dual partial ordering of the agents and functions is what Agile methods are taking advantage of when they break what is to be done down into stories and begin creating parts of the system in a random order looking for quick and easy things that can be done and demonstrated to achieve the necessary commitment to move forward in creating the whole system. The inherent flexibility of systems both in use and in development orders are due to their double partial ordering. Agency is partially ordered and Functions are partially ordered. The two partially

ordered sets of elements together produces a domain which is partially ordered with a lot of possible orderings both for use and for production sequences. In a sense this generates freedom for the users and the developers too. As long as intrinsic architectural precedencies are fulfilled which are at the core of the system, then any order of use or development that are allowed are possible. Even when there are these architectural precedencies prototyping and stubbing can give further freedom to develop in a different order than that dictated by the internal structure of the application in which some part of it depends intrinsically on another part to be able to work properly.

Now here we mention two techniques that can help to further enhance this indeterminacy of use order and development order. One was developed by R. Taylor called **C2** which is to suppress call structure. The bane of reuse and of refactoring is the changes in call structure in applications. Taylor invented the idea of self-talk, i.e. where modules in the system talk to themselves, and where other modules listen on broadcast busses to this self-talk and respond. This limits necessity the call hierarchy structuring the application making reuse much easier because the mutual dependencies are decreased or made implicit rather than having explicit representations. This is a brilliant use of indirection in the construction of systems that has not received the recognition it should have. The other technique is to use expert system structuring to regulate the order of execution of instructions in the virtual machine. This is done by making it so that the various operators in the virtual machine have guards on their operands so that the operator only fires when its guards are activated. This prevents operators within if statements from firing if they do not have sufficient information to produce a correct answer. Operators are in a while loop of if statements and order their firing themselves based on the availability of all inputs signified by the activation of all their guards. This is another form of indirection related to the operands rather than the operators. But it means that the sequencing of the operators is implicit rather than explicit in the program and thus the programmer is not determining the order of the calls of operators, but operators are called in whatever sequence that they can given the availability of the information they need for their parameters in order to produce a correct result. When that result is calculated it resets its own guard and thus triggers downstream calculations that are dependent on it.

When we combine these two techniques with the domain of dual partial ordering of agent and function then we get a slightly different picture of computing based on indirection rather than direct calls and determination of the order of programming constructs by the programmer. An Agent has within it a virtual machine (set of objects with their methods) or the agent is accessing a virtual machine. Either way we want the agent to engage in self talk, and we want the virtual machine sequences to be determined by guards on operands. This means that the agent does not have its own ports by which it gets messages to activate its methods via method calls from other objects. Rather the agent is connected to a broadcast bus on which it places its self-talk, or it is listening on a bus to which it has subscribed. The agent does not call methods of other agents directly. Rather it says to itself what it would like to happen, and those listening can respond if their constraints are satisfied. This allows new agents with different virtual machines to be added and subtracted to the system without the necessity of refactoring. It makes the system inherently adaptable because different agents that respond differently to the self-talk of other agents can be substituted into the system easily. The direct call structure is suppressed in favor of an implicit call structure that is based on saying things to themselves and broadcasting it in a form that others understand. Refactoring means substitute into the architecture agents with virtual machines that react differently to the self-talk of other agents. VLMs that are called by other agents that are outside the agents are services and do not react to self-talk of others. On the other hand we also want to suppress the explicit ordering of calls of operators in the VLM, and we can do that via guards. In other words we do not place in the code direct calls in a specific sequence for any

VLM. Rather we augment each operand with guards and we allow the VLM calls to sequence themselves. So for instance, an agent hears via the broadcast bus that another agent has prepared a model, and say that model needs to be traversed in order to produce a result. The responding agent has to call a method with a sequence of calls to respond to that self-talk of the model building agent. But instead of calling those methods in a given order, rather it places guards on those method calls so that merely by calling the first in the sequence the rest of the sequence fires on its own determining its own order of calls. This means that if there is a change that requires the VLM operators to be called in another order it will adapt itself to that change and it will flag what variable is not fulfilled in case the change causes it not to be able to fire the sequence as causation ripples through the system.

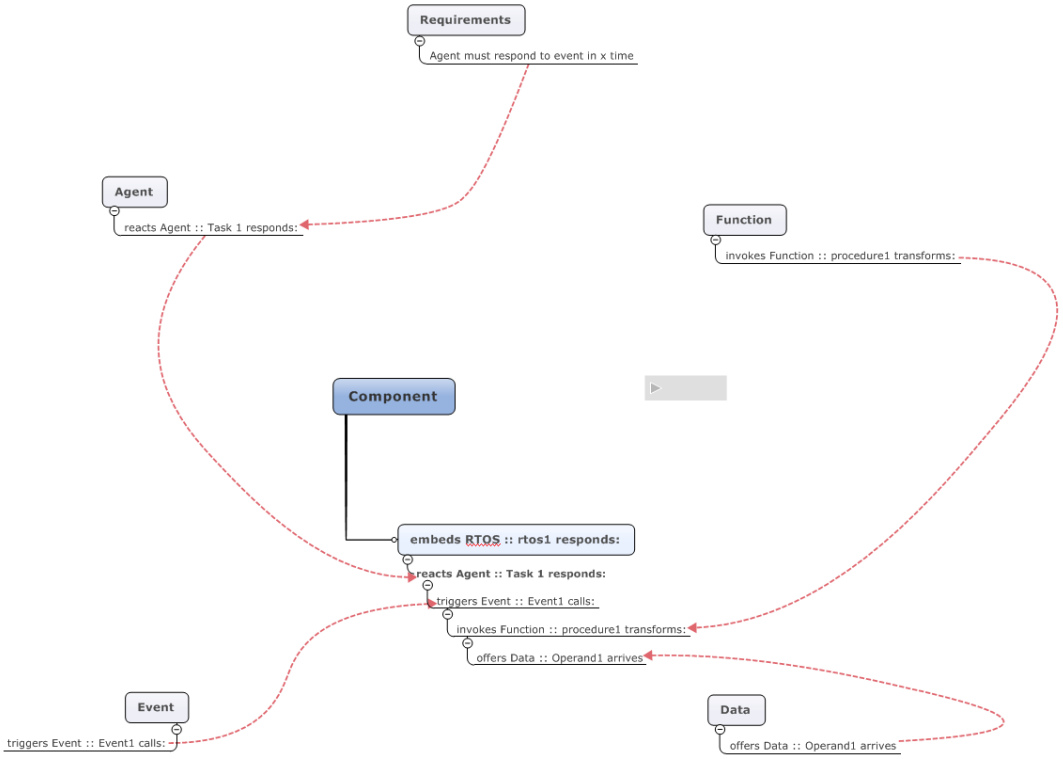
Now in general we will use Gurevich Abstract State Machine method to describe a given system at a certain level of abstraction. Gurevich has proven that rules can be used in this way produce Turing equivalent machines. In general we use macro rules to represent the causality in the application. Given a set of inputs then certain VLM operators are called, which call others and so on until an output is obtained. In general we work backwards from outputs to verify all the necessary inputs to make the machine work and thus show completeness. Completeness means a complete set of causal chains from beginning to end of the machine as defined by rules. Consistency means that the inputs of the Machine is commensurate with the outputs, and vice versa. Wellformedness means that the entire system is described by rules that represent the causality within the application. Agency can be represented by concurrent ASMs. Or we can think of agency as being contained in a machine. The assumption of Gurevich is that all computations are instantaneous and that is how he puts concurrency out of play. He assumes that all rules can be executed in any order in the GASM. These simplifying assumptions address precisely the problems we were addressing with the two techniques we mentioned previously, i.e. freedom from explicit ordering, and freedom from an explicit call structure. GASMs are rules but they are implementing State Machines. A state machine is a combination of input state plus input variables that entail a function call that produces outputs and an resulting state. Basically queue plus a state machine is a turing machine. The queue would be the inputs that arrive in a given order to drive the GASM. We use the Pattern Method of Pieter Wisse to determine the identity of elements in different contexts. We do not assume that something is identical though all contexts as is normally done, but rather we assume that as behavior changes in different contexts then identity is transformed. Wisse gives us an alternative context driven way of determining the identity of objects that are dealt with by the GASM, and make up its structure. But for architecture the important thing is how agency is distributed in order to enhance Quality of Service once we have a GASM that works (proof by existence). And it is in the introduction of agency for performance reasons that we begin to apply the paradigm of self-talk and guarded operands in order to make sure that the physical machine by which we implement the logical or essential machine is free from pre-determined order as possible in order to make refactoring less onerous. Extra structure is needed to produce agents that produce and consume self-talk, and also extra structure is necessary to guard operands that prevent internal rules from firing, so that the sequence of firing is not determined beforehand. This maintains the freedom of the partial ordering of agent and function to the maximum degree possible even in execution. Once we know the ordering of the firing we can reorder the code so the firing is in the normal sequence of firing. But this does not prevent other sequences of firing that may occur given different input sequences. Because agents are engaged in self talk overheard on party lines to produce implicit causation, then causal lines are not manifest in the call structures and so it is much easier to plug and play different agents together to produce systems that may evolve in a freer manner with minimum refactoring.

Now of course what we want to add to this is the ability to generate code rather than writing

every piece of code from scratch and so as we go along we will attempt to show how this structure lends itself to code generation techniques using configuration files and standard models in order to produce the programmatic structures with code templates that are filled in using models and configuration files. Possible changes are isolated to configuration files as much as possible so that we can change the behavior of the system by changing the configuration file rather than either the code or the domain specific models. In our case the domain specific languages are architectural design specific, but application specific domain languages can augment these using the ISEM statement format. Each design domain specific language describes a minimal method bridge between viewpoints on the real time system. These are the VLM, Use Case, State Machine, Petri Net, Sequences, DARTS, Dataflow/Object languages in ISEM format, as well as others. Mostly these minimal methods appear in UML and SysML. We recommend TextUML for models when UML conformance is important. It is supported with TextUML toolkit and AlphaSimple from Abstratt which gives examples of code generation from TextUML models.

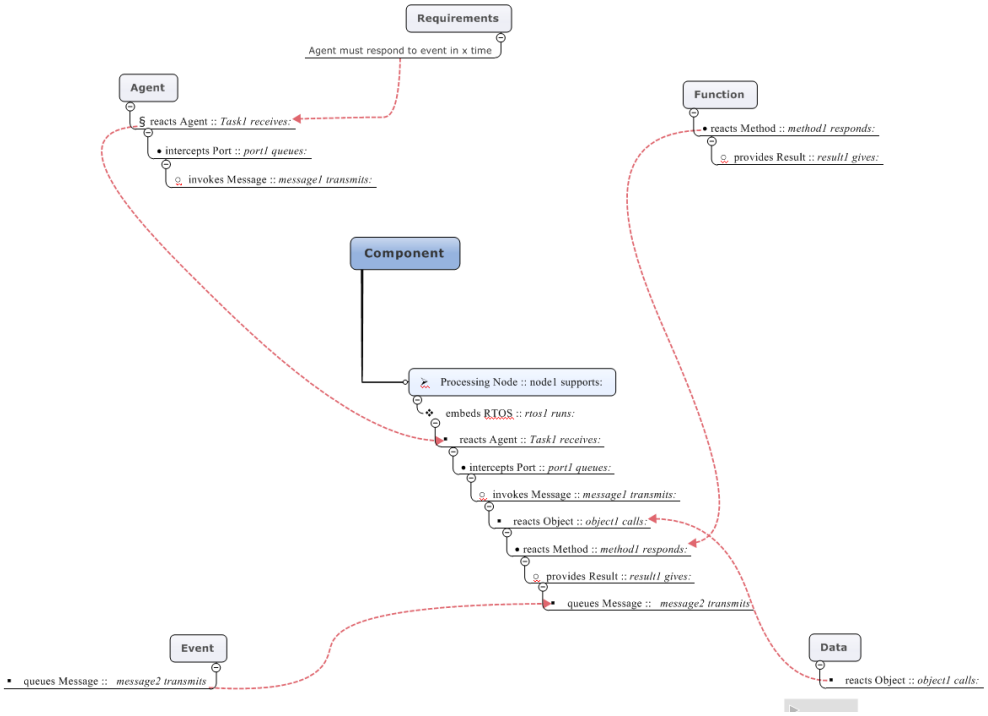
Here we are concerned with understanding the relation between the models represented by the minimal methods and the viewpoints on Realtime software. Now that we know the viewpoints and know how each is defined, for instance the functions and agents are defined by a hierarchy. Events and Data are also defined by a hierarchy but further specified by Interval Logic and ERA diagrams. In each case there is an anchor node for a given hierarchy and then the entities of the type in a parent child relation that forms the hierarchy. But the Component Hierarchy that represents the whole architecture of the system also has an anchor and then component entities. But the difference is that in the component hierarchy there is a mixture of types that can be dragged and dropped onto a higher hierarchy element to become a sub-element in the component hierarchy. We can have a separate set of network components that produces a network topology which can form the basis of the component hierarchy. So if we drag and drop the network components into the hierarchy first that represents the hardware infrastructure. This can be seen to work in the Business Analyst model in InteGreat. However, that model does not serve Software Engineering as well as might be hoped. Instead we would expect a UML/SysML like model for use in Software Engineering development modeling. The point is that just like in InteGreat we can then drop on top of hardware network topology elements the various software elements such as agent, function, data, event. Each combination of these has a separate meaning.

- Processing Node *supports:*
  - embeds RTOS :: *rtos1 runs:*
    - reacts Agent :: *Task1 responds:*
      - triggers Event :: *Event1 calls:*
        - invokes Function :: *procedure1 transforms:*
          - offers Data :: *Operand1 arrives:*



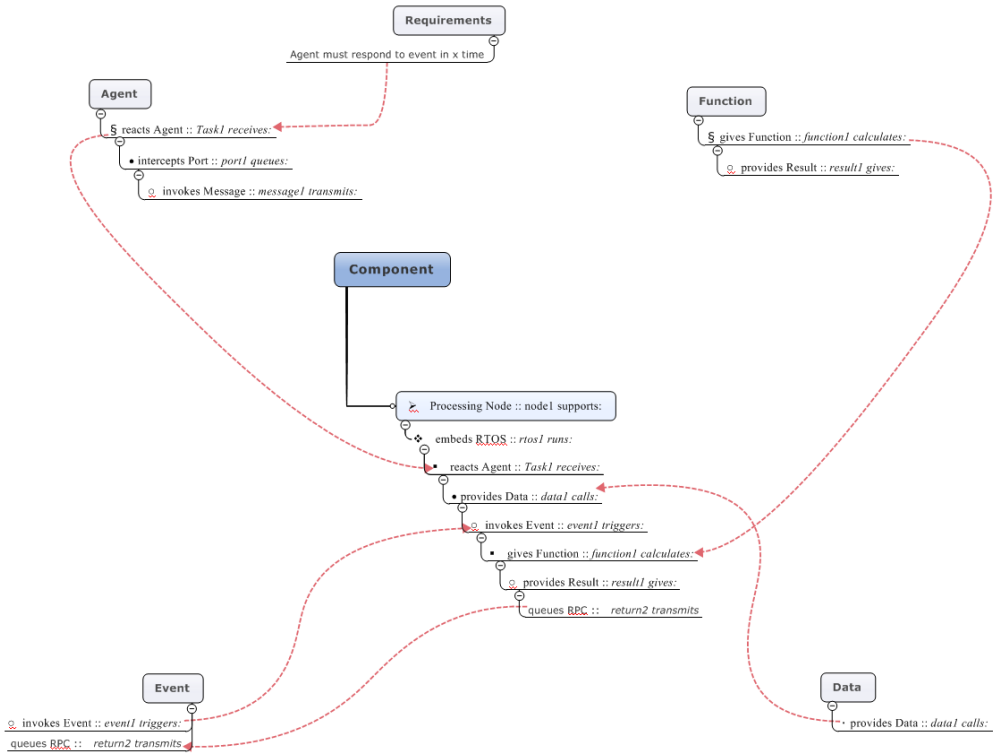
This is the nominal configuration in which there is a hardware processing node which has a Real Time Operating System (RTOS) on which are agents as tasks which are assigned functions that support dataflow triggered by events.

- Processing Node :: node1 supports:
  - embeds RTOS :: *rtos1 runs:*
    - reacts Agent :: *Task1 receives:*
      - intercepts Port :: *port1 queues:*
        - invokes Message :: *message1 transmits:*
          - reacts Object :: *object1 calls:*
            - reacts Method :: *method1 responds:*
              - provides Result :: *result1 gives:*
                - queues Message :: *message2 transmits*



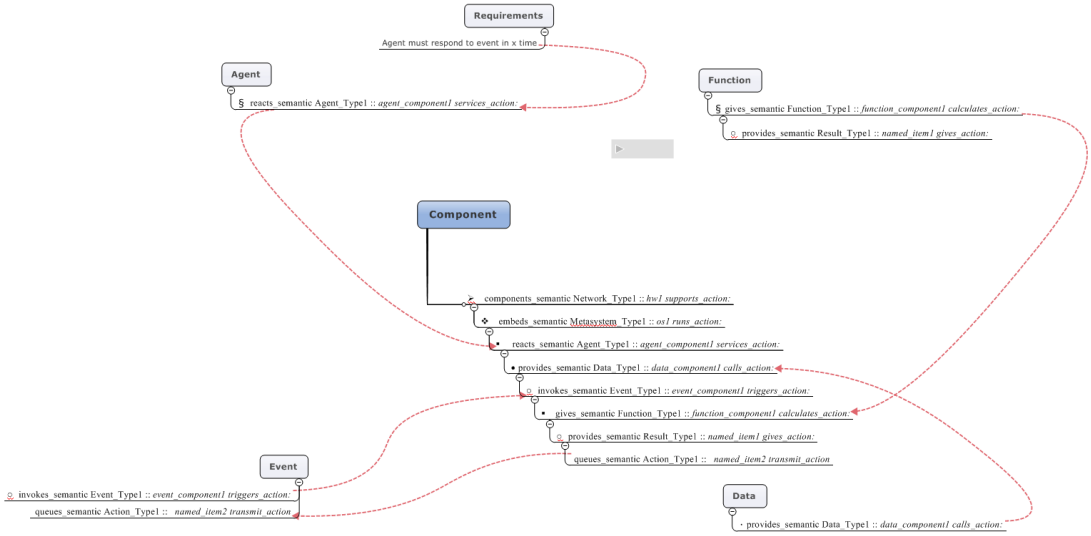
If we reverse data and function layers we get encapsulated data with method functions driven by events of messages arriving in a queue.

- Processing Node :: *node1 supports:*
  - embeds RTOS :: *rtos1 runs:*
    - reacts Agent :: *agent1 services:*
      - provides Data :: *data1 calls:*
        - invokes Event :: *event1 triggers:*
          - gives Function :: *function1 calculates:*
            - provides Result :: *result1 gives:*
              queues RPC :: *return2 transmits*



If we place data first then we get another configuration where data is a service request returned by the agent. This is what the generic structure looks like

- components_semantic Network_Type1 :: *hw1 supports_action:*
  - embeds_semantic Metasystem_Type1 :: *os1 runs_action:*
    - reacts_semantic Agent_Type1 :: *agent_component1 services_action:*
      - provides_semantic Data_Type1 :: *data_component1 calls_action:*
        - invokes_semantic Event_Type1 :: *event_component1 triggers_action:*
          - gives_semantic Function_Type1 :: *function_component1 calculates_action:*
            - provides_semantic Result_Type1 :: *named_item1 gives_action:*
              queues_semantic Action_Type1 :: *named_item2 transmit_action*



These are tangled hierarchies. It is the perspectival hierarchies that are tangled together to produce specific architectural configurations of named components. Now InteGreat only gives the lower relation and not the upper relation in each case. The concept is that all possible drops into the component hierarchy have a specific relation, but in InteGreat you can change the name of the relation from the canned one if not appropriate. The upper level action relation is not known until the drop is made, and may be selected from a list of actions.

- [relation_semantic] Object_Type1 :: *component_name1 [action]:*
  - [relation_semantics] Object_Type2 :: *component_name2 [action]:*

Relation semantics may be of different types between two objects, and actions may be of different types between objects so these would be lists of possible relations and actions. However, there can be a default that allows them to be set upon the drop into the component hierarchy. All the various permutations of drops into the hierarchy must be analyzed in order to make sure that they make sense, but a particular drop is meaningless then it should be prevented. But we do not expect there to be any illegal drops, but this cannot be determined until the combinatorial analysis is done. We must note that there is an implicit combinatoric of possible hierarchical relations within the component hierarchy whereas this is explicit in the test hierarchy. We expect there to be a duality that can be taken advantage of between the two hierarchies in terms of test generation possibilities but until this is made explicit it is hard to say exactly how that would work.

At this point what we are wondering is if this scheme would be a way to prove consistent and complete the design ISEM sub-languages. I have been looking for a scheme to show the consistency of my the ISEM sub-languages since they were up graded based on what I learned from doing my dissertation. The language now has about 1700 statements which is about a thousand more than the first version published in Wild Software Meta-systems. It is amazing to me that no genuine design language of this type has been created in all this time. I more or less assumed that other would have the same idea and do it better than I had done it originally. But the human readable languages that have been created still smack of programming languages or formal languages and nothing clear and simple of this type has been created so far to my knowledge. When I ran into InteGreat and it was explained to me how it worked I saw that it was a semantic tool like no other I had seen, and it appeared to me that perhaps it might provide a way to test the completeness and consistency of my language. Of course since it is a Design language

and thus is not convex and closed it can only be shown to be para-complete and para-consistent. In general the idea is to tie the language structures to the hierarchies and to make sure that the permutations of the hierarchy relations is mapped out by the languages. The languages are more expressive than the hierarchies and their entanglements, but the combinatorics of entanglement gives some expression to the core relations that the language needs to provide which is independent from the expressions of the languages. from a list of actions.

- [relation_semantic] Object_Type0 :: *component_name1 [action]:*
  - [relation_semantics] Object_Type5 :: *component_name2 [action]:*

from a list of actions.

- [relation_semantic] Object_Type3 :: *component_name1 [action]:*
  - *Begin Component 1*
  - [relation_semantics] Object_Type4 :: *component_name2 [action]:*

from a list of actions.

  - [relation_semantics] Object_Type5 :: *component_name2 [action]:*

from a list of actions.

  - *End Component 1*
  - *Begin Component 2*

  - [relation_semantics] Object_Type6 :: *component_name2 [action]:*

from a list of actions.

  - [relation_semantics] Object_Type7 :: *component_name2 [action]:*

from a list of actions.

  - *End Component 2*

- [relation_semantic] Object_Type8 :: *component_name1 [action]:*
  - [relation_semantics] Object_Type9 :: *component_name2 [action]:*

The entanglements can be packaged and given component names that combine different sub-components into larger scale components. In this way entanglements can be hidden and only seen as necessary. This packaging helps to make the components generic and reusable or copyable.

We note that the component hierarchy and each sub-hierarchy within it can be a bus to broadcast self-talk upon. Also actual function calls with operands can have guards so that this structure becomes one which could support the elements that would make refactoring less necessary. Refactoring when it is necessary would be accomplished by dragging and dropping the elements within the hierarchy. If code generation became possible from these models then this would also be a way to refactor the code without having to touch it.

Hacking the Essence of Software

It is clear that the core of software from a theoretical perspective is the Turing Machine. And since this is the Turing centenary we should focus to some extent on this core and attempt to understand it with respect to the problem we are attempting to solve which is how to represent the design of software, and how to show that Design DSLs are para-consistent and para-complete and para-clear. What has been suggested so far is that there are hierarchies of entities in the related to requirements, agents, functions, events, data, and tests and that these become tangled in a component hierarchy, and it is the relations and actions in this tangled hierarchy which is the means of establishing the para-characteristics (clear, complete, consistent, verifiable, valid, and coherent) related to the aspects (real, true, identical, present) of Being. We note that the component hierarchy appears in the hole in the lattice of the methodological distinctions and between the dual measures of linear order without distance and Partial Order with distance. These two types of order stand as a way of talking about delocalization and decoherence as well as the basis for the structure of the dualities of the minimal methods. So for instance linear order without distance is a description of the Code we create when programming. Each statement in the code is in a linear order, but the performance distance between each statement is unknown. We have to make external measures of performance to determine that distance, and because there can be context switching between statements it is not sure that a given statement will actually be executed immediately after the next when considered from the point of view of actual timing, we just know that if it executes at all it will be sometime later. The other type of order, partial order with distance says that programming constructs that are being executed are not necessarily ordered in a strictly linear way, and may if say they are being executed in different tasks, or by different hardware infrastructures be such that they will actually execute in different orders during different runs but the effects may be measureable on some background variable such as time or memory or population (which are the sources of the hierarchies that represent the viewpoints on realtime systems in Klir).

Both of these ordering types can have many different relations to each other and the result of that is decoherence and delocalization that occurs in actual programmed, so called hacked, code. The use of the term hacking highlights the very pragmatic nature of all our attempts to get things to work in spite of decoherence and delocalization. Delocalization reminds us that references to design elements may be spread out or smeared out in the code as a linear static organization. Decoherence reminds us that just because something is near to something else in the code it does not mean it will be executed in a way that we expect, discontinuities can occur between statements when they are reduced to assembly code and executed, and the fact that there are multiple hardware processing elements, or multiple tasks may mean that the order of actual execution is not set, even though it can be measured so that we are dealing with probabilities and not determinate results when we are talking about executing masses of binary executables. A lot happens when code is compiled and what that compiled code does in actuality may be different from what we expected it to do when we encoded it. In a sense the actual result of running code must be used to decode what we encoded. So there is a continual interpretative process that occurs as we write, execute and debug the code we are hacking. To say that code is decoherent and delocalized is merely to say that it represents an example of what Merleau-Ponty called Hyper Being, or what Derrida called DifferAnce, or what Heidegger called Being crossed out. The pointers and accumulators in the hardware represent what Heidegger calls Present-At-Hand (pointing) and Ready-To-Hand (grasping) in Being and Time and which are interpreted psychologically in Phenomenology of Perception. Software is the only cultural artifact that has as its essence Hyper Being or differAnce. DifferAnce means differing and deferring all the time, which Paul Simon calls “slip-sliding away” and which Plato in the Timaeus called the Third Kind of Being. Software is something written that executes and as it does so it can rewrite itself. And in

fact it is precisely a machine that can rewrite its own tape, and ultimately its own program, that we call a Turing machine. And there are two types of Turing machines, the normal ones and Universal ones that run other Turing machines which we now call operating systems. But we should really call them operating [meta-]systems, because they go beyond the system of the turing machine and are actually models of its environment. The dual of differing and deferring of software is effectiveness (agile) and efficiency (lean) which together give us efficacy. Software can differ from itself as it institutes differences and its execution can be measured in terms of its effectiveness and efficiency. And as humans that produce software as an allopoietic product, we can be lean and agile in that production process which together make up what Reinertsen calls Flow. What we are saying is that there is some mirroring between the software product and the software development process that we need to take into account when we consider it as a pragmatic human activity. These computing infrastructures we build are far from autonomic, and so we have to build them piece by piece and then we have to maintain them for them to continue to work, and they are very fragile, and that is why the testing needs to explore as much as it can the combinatoric order of testing possibilities in order to assure robustness to the extent we can. But combinatorics are so vast that we need special ingenuity to make sure that systems are well tested, because it is many times impossible to test all paths beforehand. So that means we really need to understand the nature of software in order to produce good software. Part of that is understanding each of the minimal methods that can be generated out of the duality between the two orders as they are projected as bridges between viewpoints on the realtime system.

So now in order to try to extend our understanding of the Turing machines and State machines that are the core of Software, i.e. the place where Design meets Programming, which in its pragmatic aspect can be called hacking which seeks ultra-efficacy in the development of working software as seen in the Agile paradigm that emphasizes hyper-effectivity and the lean extension that emphasizes hyper-efficiency. A key question in this regard comes up when we try to understand the nature of Peirce's Firsts (isolate, points), Seconds (relata, lines) and Thirds (continua, surfaces) are in relation to the Turing machine and its state machine. A state machine plus a list or queue or tape is a minimal Turing machine. When we look at a state machine we see that it is normally a set of vectors composed of input, entry state, output, exit state. These can be expressed as rules ***if*** *input and current state* ***then*** *function producing output and new state*. Gurevich showed that we can use rules such as these to describe any system at any level of abstraction and it would be Turing equivalent. Thus it is not necessary to reduce something to a Turing machine to show it is computable. And Computability reduces to knowing the causality running from outputs back to inputs through the system and knowing that all those threads of functionality are complete and consistent and are well-formed. So that means we can abstract from low level Turing machines and just use the Gurevich Abstract State Machine method as our representation of the process of computing. Executing software ultimately reduces in its essence to one syntactic construct which is the if…then… statement. Execution of software means to execute rules. All software can be represented as a stepwise refinement of rulesets from any level of abstraction down to the level at which the rule can be represented in a general purpose programming language. And we can use the Pieter Wisse's Metapattern method to understand how to derive the objects that the rules are referring to at the various levels of abstraction. Guervich ASM and Wisse Metapattern methods are duals of each other in this regard, one giving the causal structure and the other the contextual basis for the identification of objects based on their different behaviors in different situations, which amounts to the identification of discontinuities in the identity and the behavioral response of objects.

Now what we notice is that actually no matter how many inputs we have, and no matter how many outputs we have, there is a three way relation between the inputs, the outputs and the states, and that the two mentions of the state, i.e. the self-reference of the state machine providing a

pivot of identity still makes only a third element. And this is related to what Peirce calls the structure of the sign. A semiotic relation is a Third, or continua that is an object, and interpretant and the sign itself. In this case we have the object as seen in the input, and we have the interpretation as seen in the output, and we have the sign in the state which is transitioning within the state machine that produces an algorithm that converts from input to output based on state. The transformation between input and output is performed by a function. Agency is represented by the infrastructure that is performing that computation. For instance we might have the same statement performed in different tasks or on different hardware platforms, and thus they can be performed in parallel. It turns out that if you represent a simulation of a system with a refinement of the Gurevich ASMs all the way down to the code what results is very inefficient, even though it may be functionally effective. So, performance improvement comes from introducing architecture which usually means distributing the functionality among various agents, i.e. into tasks or among processors in a distributed system.

The state machine is in fact made up of a three way relation between inputs, outputs and states. There is a triangular surface that connects these three elements and we call that a Third or a continua. It is what Steven Wallis calls a robust theory[1]. One way to see a state machine is to think of it as having anchors of functions between input and output, but that it changes the functionality, based on its state and thus providing a different layering surface to the state machine triangles. Data from input to output will flow a certain way until there is a state change, in which case it will flow differently in dataflow systems. States change transformations from inputs to outputs, but this can be seen as a three way semiotic relation with different computational surfaces being actualized giving the state machine an identity as a single machine as it executes on various input data transforming it into different output data based on the state of the system. Now since this surface can be represented as a rule we will call the surface itself the Rule. The arrow of functional transformation of inputs to outputs is complemented by an arrow from input to state, and from state to output as the state machine determines its own state for the next input session. The rule is a surface, and its boundaries are the functional transformation, the if part (left hand side) queries the state to determine the function, and the then part (right hand side) that sets its own state for the next round of inputs. We can then see that the data of input, output and state are the discrete isolate of the First, and that the function, and the self-querying of the *If S* and the response of the *Then S'* are the seconds or the relata that bound the surface of the rule R. Rules are surfaces or continua. I think this is a new way to look at them in terms of Peircian principles that I have not seen in the literature yet. It also shows why states are signs, and that state machines are semiotic machines. They take in objects (inputs) and they use signs (states) to interpret them giving outputs via functions. If we understand that state machines are semiotic machines then I think it clarifies why we call their production encoding and the interpretation of their execution results decoding. And this way of looking at it probably came directly from Turing's working on codes during the Second World War. When we are coding we are setting up a sign system and that involves taking in information and transforming it and putting out our interpretations along with the product of the computations. The internal state of the state machine is what gives it an identity. It is the identity that is preserved by the various rules that make up the machine as its sensing-action vectors. It is sensing what is present, i.e. the inputs it is given. When we combine the aspect of presence and identity with truth we get a formal system. A formal system has the properties of completeness, consistency, and clarity (wellformedness). The rule set of the state machine (its vectors as a set) need to be consistent and complete for the state machine to function properly. Wellformedness comes from the fact that all the vectors are expressed in rules. The lowest level of Truth with respect to Pure Being is

[1] From Reductive to Robust: Seeking the Core of Complex Adaptive Systems Theory Steven E. Wallis 2008 DOI: 10.4018/978-1-59904-717-1.ch001 http://www.igi-global.com/chapter/intelligent-complex-adaptive-systems/24182

verifiability. That means that we can compare the reality of the results of execution to the statements themselves and show that the statements do in fact express what the machine does. So the truth of the state machine has to do with the gist of the statements and their mutual interoperability and the wholeness of their organization indicating a singular unified totality, i.e. a synthesis, which is complete (Truth related to Presence) and consistent (Truth related to Identity) and clarity (Presence related to Identity).

Now we know in the Turing Machine that the state machine is related to a tape, and that the tape is a series of places with symbols in them. The Turing machine takes in the symbols and produces other symbols. There is a pointer that indicates what place with a symbol that we are talking about at any given time. This is called the tape pointer. Tapes are finite on one end and infinite on the other end in the original conception of the Turing machine so that it can handle infinite computation. The tape is an extent and this is the representation of space. The pointed to symbol is a gestalt on the background of all the other symbols on the tape. Now the input and output for the state machine comes from the tape. So the symbol on the tape is a fourth entity producing a minimal system with the input, output and state. So what we need to explore is what this fourth entity gives us beyond the state machine. Since the other entities make up the formal system of the state machine, then we would expect the symbols on the tape to stand in for reality. Reality is related to the other aspects of Being by giving us verifiability, validity, and coherence characteristics. The state machine can read and write symbols to and from the tape. So there is a directional line from the tape to the inputs and from the outputs to the tape related to read and write operations. This creates another triangle which is composed of read, function, write operations. The focus of the read and write is where the tape pointer is pointing at any given time, and this is the point in the extent where timing occurs. In other words the pointed to cell becomes a spacetime nexus within the worldline of some agent. The surface that is defined by the triangle of input, output and symbol in place on tape (gestalt) with the read, function write directional relata (arrows) is an interactive flow.

Once we have defined another surface which relates the state machine to its tape which also relates the formal system to reality and thus generates significance. We can verify the statements of the state machine against the tape by watching what is written to the tape. And we can validate the state machine by looking at the results of the execution of the state machine through the results on the tape. By relating the state machine to the tape we also get coherence because the state machine state is an identity and that identity gets reflected back onto the tape though the outputs of the state machine operation which can be seen as coherent if it does what was intended and so we start to see agency in the coherence of the operation of the state machine as reflected on the tape. There are two other surfaces that related to this effect. The first is the surface related to reading input. Associated with this input is the state we see in the left had part of the rule and that is completed by an interpretation of the symbol on the tape that is the figure of the gestalt. This surface is hermeneutical. On the other hand there is a surface related to writing output which is associated with the right hand side of the rule and signifies intent. So interpretation takes the symbol as a sign of some significance and the intent gives a sign of some significance. Both of these semiotic characteristics are signs of agency, which is the dual of functionality. But the interesting thing is that there is a duality between ‘interpret’ and ‘intent’, while functionality is unified. The surface related to writing outputs and intent is causal. Now we have four surfaces rule, interactive flow, hermeneutics and causal intent (or affect) that are all what Peirce would call a third or a continua. Interactive Flows relates the state machine to the tape and thus relates it to spacetime creating a worldline of an agent through the controlled and organized operation of the functionality of the state machine as it relates to the contents of the tape which can contain either encoded data or algorithms. The organization of the state machine is seen in the relations of its rules to each other that reasserts its identity. So the State is related to identity and the Tape is

related to Reality. Presence is related to inputs and truth is related to outputs. Inputs are what is present in input variables. Outputs are what show the organization of the system via the state machine that is true, where true means going in a straight line based on criteria that are used to determine that it is straight. So for instance any linear system is true. i.e. it is producing straight line output. All non-linear output is judged on the basis of the true coordinates, i.e. orthogonal straight lines.

This association of the isolate with the aspects of Being (tape=reality, state=identity, input=presence, output=truth) comes from the fact that the various surfaces (interaction, rule, hermeneutic, causal intent) intersect by threes.

Surfaces: Interaction, rule, hermeneutic = input isolata -> Presence aspect

Surfaces: Interaction, rule, causal intent = output isolata -> Truth aspect

Surfaces: Interaction, causal intent, hermeneutic = symbol on tape isolate -> Real aspect

Surfaces: Hermeneutic, causal intent, rule = state isolata -> Identity aspect

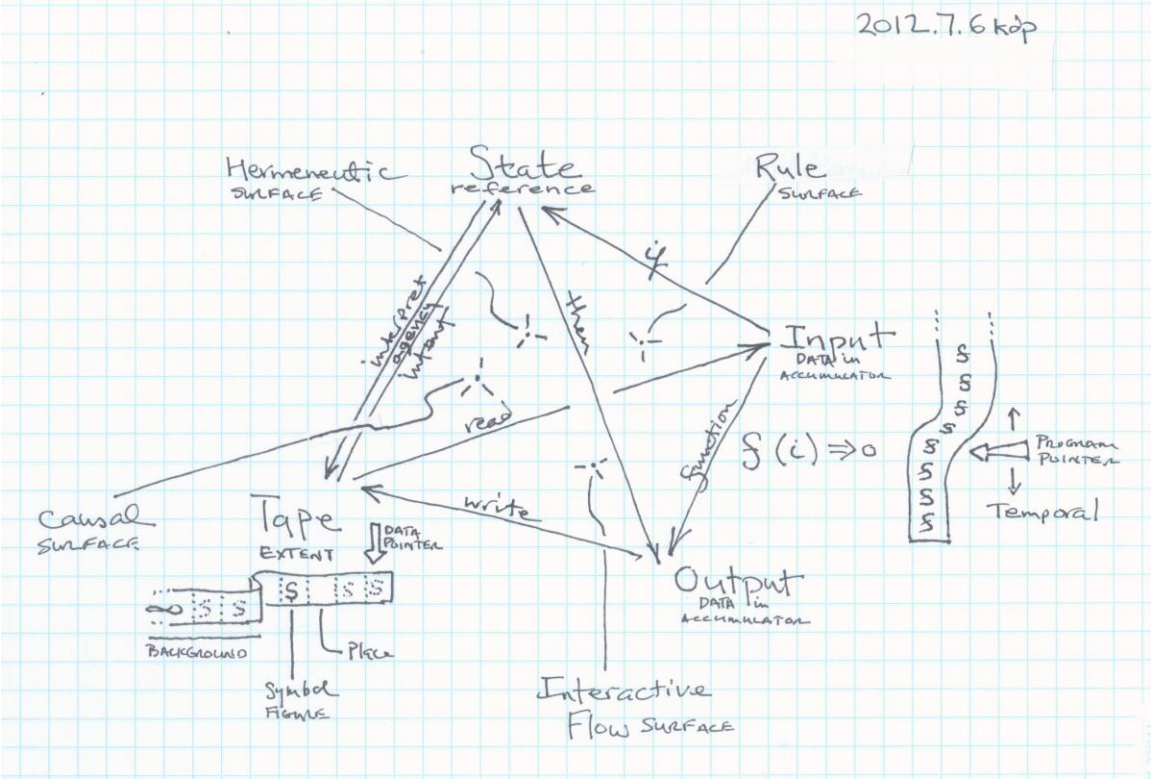
Similar things can be done by looking at relata:

Read, If clause, function = Input

Write, Then clause, function = Output

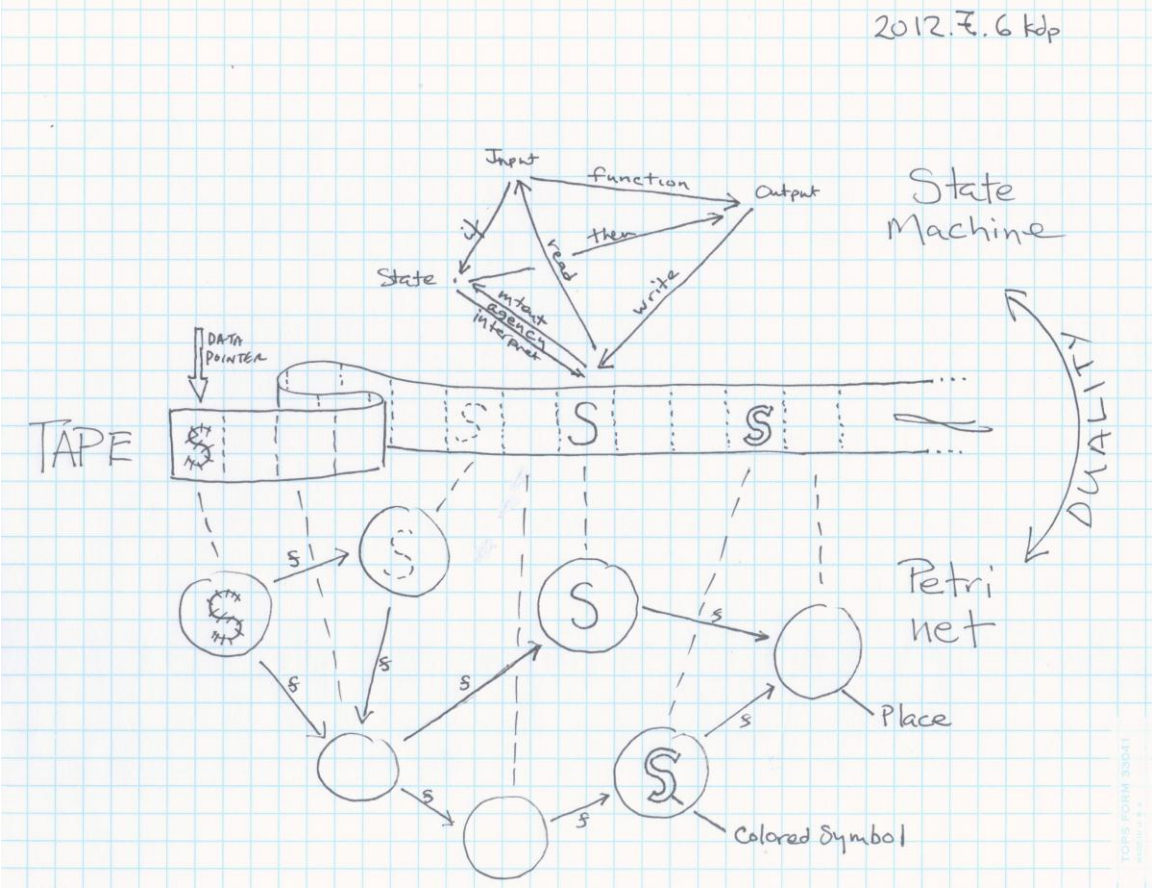
If clause, Then clause, semiotic = State

Read, Write, semiotic = Tape symbol



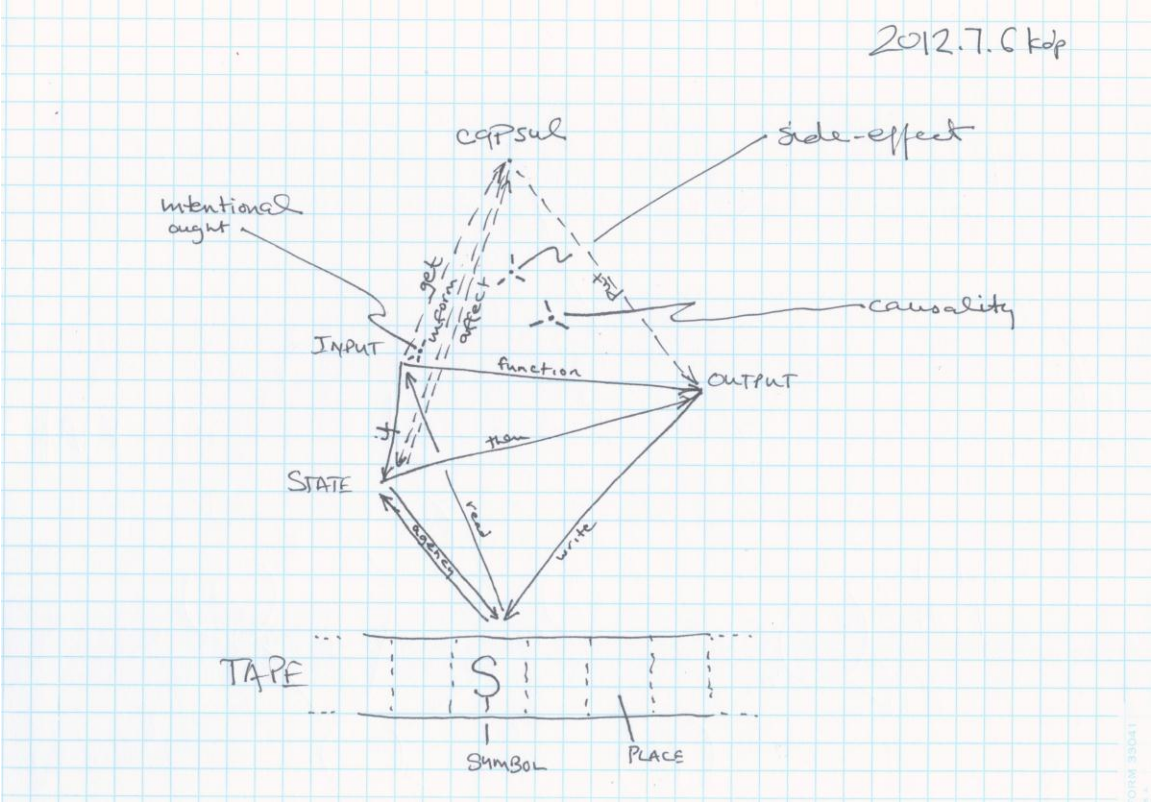
This is a minimal system as defined by B. Fuller. All the elements are informed by all the others diacritically. It is a tetrahedron of concepts composed of four isolates (symbol, state, input, output), six relata (function, read, write, if clause, then clause, semiotic), and continua (rule, hermeneutic, causal intent, interactive flow). Now what is surprising about this extension of the concept of the Turing Machine is that it is semiotic and thus connects directly with Peirce's idea of semiotics as a threefold relation. In it Rules as a surface mediates between the hermeneutic surface and the causal surface. Both of these surfaces are based on and define the surface of interactive flow, which is the basis for positing the gestalt of the symbol on the ground of the whole tape. It also produces a double bridged line of agency existing in a tension between interpreting (taking for a sign) and causal intent (giving a sign). This double action of the agency is the dual of the orthogonal line of function, which is also a method for objects. It is interesting that the agent line is composed of two oneway bridges while the function is a single oneway bridge. There are various compositions of directional arrows bounding each surface. All the isolata are variables of different kinds. All of the lines are directional. Two of the surfaces form circuits around their parameters. The oneway arrows of the function and the clauses of the rules forms a circuit with the tape. It is the dynamism of the tape that allows the machine to work. The state machine itself is reactive. The dual of the state machine is the petri net which is proactive. But also there are multiple petri net representations for a given state machine kernel. Petrinets are more proactive but also more superficial. State machines are condensed representations that are most efficient and effective. You can get this kind of proactive structure from two state machines that are interlocked each feeding the other. Colored Petrinets are better at exhibiting control structures that are self-starting. The colored Petrinets operate more like cellular automata using markers in places to activate transitions. Petri Nets look at function from the point of view of event, while State Machines look at events from the point of view of function. The event is a triggering of the transition when the marker is in the place and the function occurs in the transition. The colors of the markers are the inputs, and the colors of markers are the outputs too. State machines on the other hand transform what functions are called given monadic state identity

operations whose differences can be used as a controller. In state machines function is central and in petri nets it is event that is central created by the marker being in a place, like the symbol is in the place on the tape. When we put together the petri net and state machine then we have the tape as active, and the symbols being triggering mechanisms to change the colors of the symbols. So the two dual mechanisms can both work together without interfering. The petri net merely colors the symbol. This is an autopoietic symbiotic relation between the two archetypes of computation.

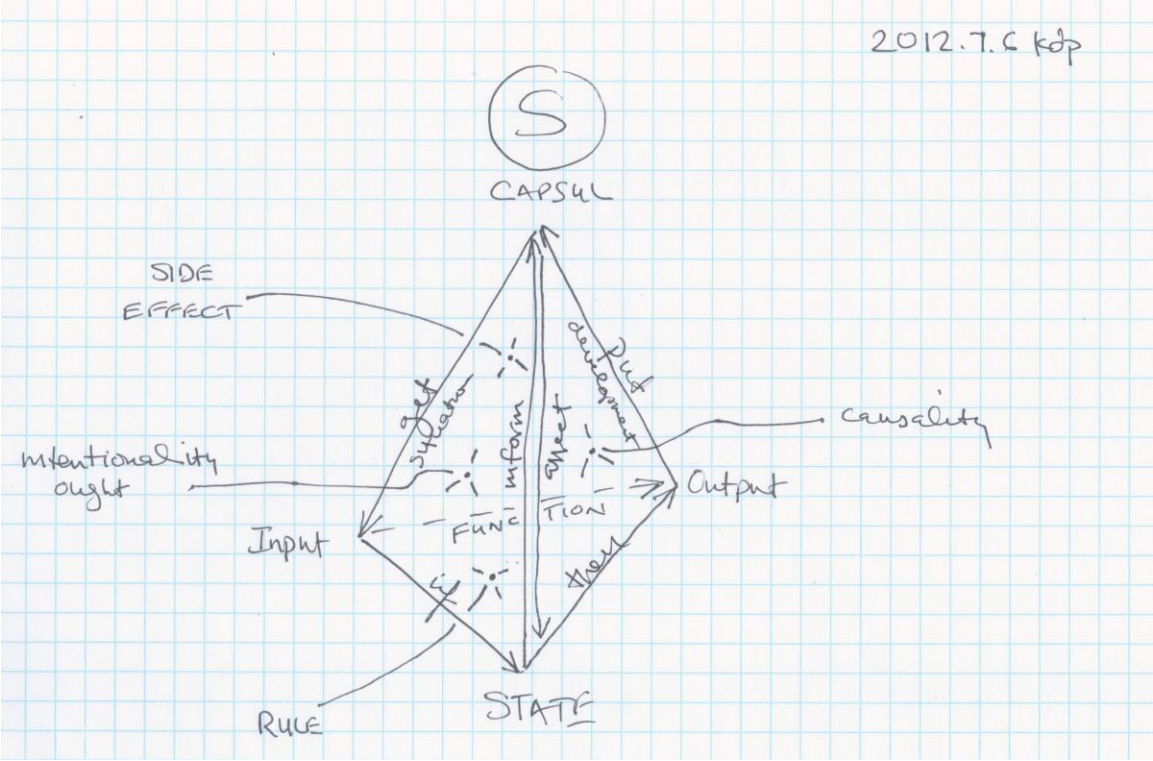


Once we have realized that the two archetypes of computing can coexist together manipulating and using the same tape in an autopoietic symbiotic relationship where one manipulates the symbol and the other manipulates the color of the symbol, one treats the symbol as an existent maker with color which causes transitions to fire, a firing transition is just a function that takes colored makers as input does a transformation with side effects and then places makers in their output places. On the other hand the State Machine treats the symbol as a character and reacts to its characteristics as a symbol which informs how the function will treat it as an input symbol which is read from the tape and then an output symbol is written back to the tape, perhaps after moving the data pointer backwards or forwards. The state machine has a direct relation to the place on the tape that is pointed to and it reads and writes symbols based on where the data pointer is pointing. The petri net on the other hand has an indirect relation to the tape where certain cells are treated as places into which existent markers are placed and these places form a network that is activated by the existence of a symbol in a place of a given color. So there is a superimposition of color on symbol such that the two computations can be separate yet indirectly interact. What would happen if you had such a computational setup is unknown as each assumes the stability of the tape but the petri net would be shuffling the symbols and tuning them different colors behind the scenes from the point of view of the state machine, and from the point of view

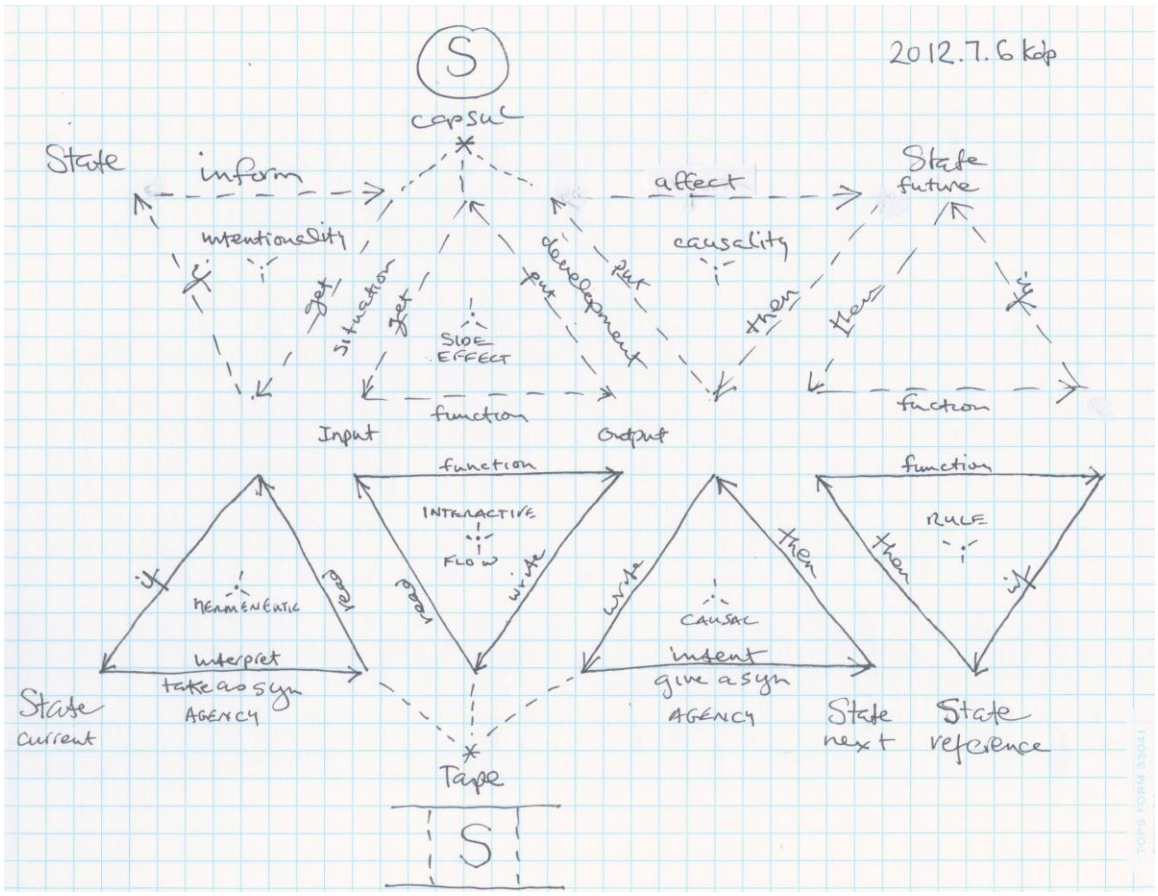
of the petri net markers would be coming into and going out of existence suddenly. From the point of view of the state machine symbols would be appearing and disappearing. What is interesting about this is that Petri Nets are active and State Machines are passive and so they have completely different characters, and the Petri Net could act as the controller for the State Machine jump starting and boot strapping action by the State Machine. Also the Petri Net is better in modeling protocols than the state machine. So it could be that the petri net could act as the protocol between two state machines within the universal Turing machines that run separate Turing machines. This thought of the Universal Turing machine (meta-machines) takes us into the modeling of the meta-system by the addition of a capsule to the state machine minimal system to form its dual.



We will think of the capsule as the encapsulated data of an object, but we can also think of it as a functional programming monad. We do a get operation in order to take the contents from the capsule and we do a put operation in order to place new contents in the capsule. This is a side effect that is placed in the capsule or monad. The surface from surrounded by put, get and function should be thought of as the side-effect surface which is different from the rule surface or the interaction surface. Once we realize that there is another surface related to capsule side-effects then we must ask what the other two surfaces represent.

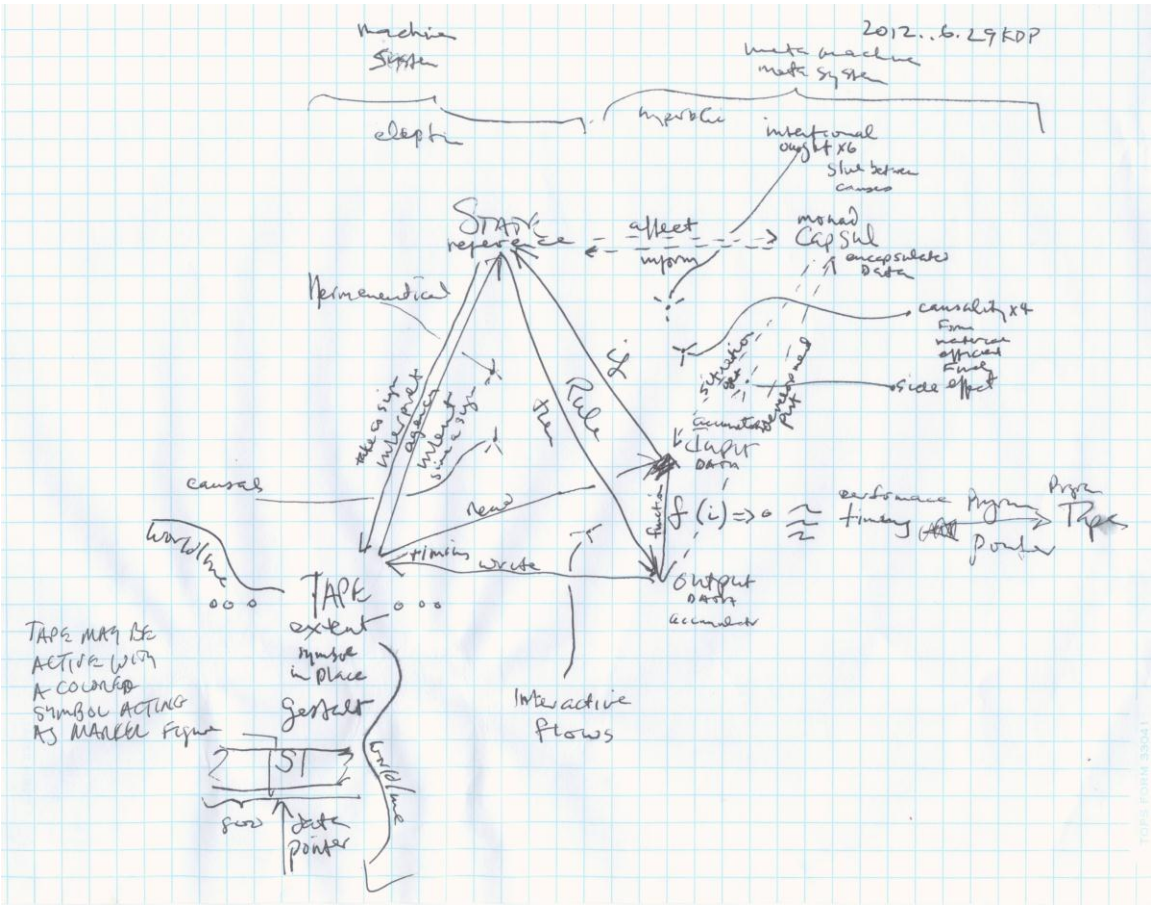


Interestingly the other two surfaces impinge on a line between State and Capsule similar to the line between State and Tape. This line is a two way bridge, so that there is one surface that is Get, If, and Inform, and another surface which is Then Put, and Affect. Let us call the Get, If, Inform surface Intentionality for the time being. Let us call the Then, Put, Affect surface Causality tentatively. We note that since the capsule can either be inside the object or outside the system as a monad it can be interpreted as either inward or outward, so we can think of the two way bridge of inform and affect as either Subjective or Objective depending on whether the capsule is inside or outside the system. So it is hard to interpret exactly what is the next higher thing from agency that is being revealed here but let us call it Dasein following Heidegger who was following Hegel. Dasein is the projective capability posited by Kant. Dasein informs and then affects, just like agency interprets and then intends. The informing cycle is related to the intentionality surface and the affective cycle is related to the causality surface. Intentionality and Causality stand over and against the side-effect surface.



These three new surfaces are meta-systemic, whether that meta-system is seen as within or outside the state machine system region. Systems nest and Meta-systems nest. But they also interleave in their nesting like Russian dolls where the dolls are the super-system, system, subsystem and the interspaces between the dolls are the meta-systems. Meta-systems are operating systems for applications and they are modeled as Universal Turing machines. A given meta-system can run multiple applications. Those applications are all state machines, that communicate with each other via protocols represented by the petri nets. The System as a bubble between higher and lower Meta-systems can see the meta-system as within or on the outside, and thus the capsule can be on either side, either within or on the outside. If it is on the outside then it is a monad. If it is on the inside then it is an object. So, monads and objects are duals.

For historic purposes here is the original diagram in which these relations were first explored.



The point of these musings is that I have long wondered how to apply Peirce's insights regarding continua or thirds to fundamental structures, and there is no more fundamental structure than the Turing Machine for Computer Science and Software Engineering. Gurevich generalized it so that we can take arbitrary levels of abstraction and see whether they are computationally and thus causally complete by expressing them as Rules. Here we see why this works which is because the rule is the surface circumscribed by If, Then and Function. Notice that the If and Then arrows on this surface are both go the same way as the function edge. This is an asymmetry within the structure. The other three surfaces are bounded by circuits of arrows, and the line that is opposite the function that stands for agency is a double bridge in order to allow these circuits to exist within the structure of the tetrahedron. I was thinking about Steven Wallis' idea of robustness which counts Newton's law and Ohm's law as robust theories, and I realized that state appears twice and that really the relation between input, output and state as a robust relation if we thought about state in terms of identity. And then I realized that all we needed was the Tape to have a Turing machine and that meant there was a minimal system. Between the tape and the input and output variables the interactive flows were defined between the state machine and the tape. The next step was to figure out the nature of the other two surfaces. Hermeneutical and Causal is what came to mind. One surface is involved in interpreting the tape, and the other surface is involved in reacting based on that interpretation. But it was surprising that agency was reflected in a dual bridge of interpret and causal intent (or affect). It is even more surprising that if we extend this to the meta-system beyond the Turing machine (the meta-machine) then we get something like Dasein and there are surfaces for intentionality and causality, which are opposite the side-effect

surface. And this interpretation is forced on us by the fact that the capsule can either be seen as inside or outside the system, because meta-systems can be nested within or as environments outside the system. The interesting thing is how when we flesh out this robust structure we see higher concepts come into play like agency and Dasein where we do not expect them. We also see how double bridges arise as a result of asymmetries in the way that arrows are configured along the edges of the tetrahedral diamond. It is only the octahedron that has perfect flow along its arrowed edges, so there has to be asymmetries in the tetrahedral system.

The tetrahedrons we have uncovered are the three dimensional and thus related to a philosophical principle beyond those that Peirce adhered to which are fourths which signify synergy and fifths which signify integrity developed by B. Fuller in Synergetics. Synergy is the reuse of parts within a whole. We see that in the reuses of state to emphasize identity across the changes of state. Integrity is tensegrity which is flexible and inflexibility mixed to give resilience. We see then that the Turing machine and the Universal Turing meta-machine tetrahedral have synergy by the reuse of the rule surface, we also get reuse of the state variable within the rule and reuse of the agent and Dasein asymmetric paths by their doubling. The capsule gets reused because it can appear as an object inside or a monad outside the system. So there are many aspects of reuse showing synergy in the diamond of the Turing machine with capsule configuration that unites system with meta-system. Integrity specifically appears as the combination of replicated and non-replicated elements in the Turing meta-machine representation. Via repetition some give or dynamism is allowed in the structure that can allow it to be dynamic and thus give software the adaptability or resilience we find in its essence. This diamond is a picture of the essence of software and is founded on the ability of software to rewrite itself and thus on the differAnce of Derrida. By using Peirce to understand the essence of software anew we are in effect hacking the essence of software itself by changing our concept of it and reaching more deeply into what it means by using the principles of Peirce and Fuller to understand this unique cultural artifact that embodies Hyper Being and that is changing our world profoundly by its incorporation into all manner of devices that are in turn change the available affordances and thus transform our world.

Frederick P. Brooks, Jr. in his famous article on the Essence and Accidents in Software Engineering called “No Silver Bullet” identifies what he believes is the fundamental and essential characteristics of software which are Complexity, Conformity, Changeability, and Invisibility. Our new view of the diamond of the System and Meta-system interface between the tetrahedral of the Turing machine and the capsule that share the rule surface does not change any of these characteristics. But what it does is explain the structure of the building blocks that when put together in ingenious ways result in complexity, and have the ability to conform, and control changeability, and inform the invisibility of the conceptual and theoretical structure of software as well as the praxis producing source code that embodies that structure effectively and efficiently. Software only seems werewolf like because it appears alien to our conceptual apparatus. But Kant placed rules at the center of reason in his first and second Critiques. But he maintained in the third critique that there are no rules for formulating rules. And when we can put this together with the idea of Wilden that The Rules are No Game. Then we see at least three levels, that of the game, i.e. the rule governed activities, the rules themselves and that which produces the rules which escape representation by them. The essence of software points to the non-representability of software design a subject that I cover in my dissertation on Emergent Design[2]. The characteristics of software come from the relation of the theory of design to the delocalization and decoherence of the code as we attempt to play the game by the rules we make up as we attempt to continue to indicate the non-representables. The point made in Scrum is that we can always change the rules and thus get an emergent event that transforms the nature of the work we are

[2] http://about.me/emergentdesign

doing and the means of achieving our goal. This is the pragmatic aspect of our play of the game in practice where we seek hyper efficiencies and effectivities and thus ultra-efficacy. Understanding the essence of software synthetically rather than analytically via the philosophical principles of Peirce and Fuller give us a better appreciation of how the various characteristics of the software essence interact to produce its intrinsic difficulty for which there is no Silver Bullet. Now we can think not just about variables within our source code and how they are algorithmically connected to each other, but we can think in terms of lines of flow, surfaces that are bounded by these flows, and the solids that bring together these surfaces into System and Meta-system spanning models. Thinking about the essence of software in this more elevated way should help us deal with the problems of decoherence and delocalization that make the essential characteristics of software intractable.

Now I am not sure that the characterization and naming of these surfaces are quite right as yet, but it is a breakthrough to be able to describe them as surfaces and thus think of them as continua, and of course because they form a tetrahedron of concepts we also have what B. Fuller calls Synergy that comes in the their dimensional platonic solids as he shows in his masterwork on Synergetics. To realize that the State Machine is a minimal system in B. Fuller's sense is important. But we can also see why the meta-system is less than the system because all it needs is three sides to form the tetrahedral diamond formation. It builds directly off of the Rule surface. So meta-systems and systems, Universal and normal Turing Machines, meta-machines and machines both share the rule surface and thus can be described by the Gurevich ASM method equally. The Meta-system has the capsule, i.e. the elliptic realm within it, that can be seen as the niche for another system. As we have seen in other papers there are three types of system, elliptic, hyperbolic and openly-closed. The hyperbolic is the complement of the elliptic. This hyperbolic entropy is what threatens the bounded system. What balances them is the openly-closed system with an oracle by which the autopoietic system knows what is happening outside without its surface being breached, by some higher dimensional sleight of hand. We see that the meta-system projects the capsule. But in doing that it sets up something higher than agency which we are calling dasein that balances inform and affect. Dasein is opposite the function arrow, the fundamental asymmetry that in this case forms a cycle with get and put. The ante is upped with intentionality and causality surfaces superseding hermeneutic and causal surfaces. It makes sense that dasein would show up here because we have seen previously that the hardware's accumulators and pointers (of which we see two each) represent grasping and pointing as Merleau-Ponty says that represent Heidegger's ready-to-hand and present-at-hand. Dasein has this strange Metaleptic characteristic of projecting its own world which it finds itself within which Kant posits. That projection boils down to intentionality and causality, but they only appear as opposites to the unintended side effects that is the surface that connects the function to the capsule. Indirection is the nature of the meta-system always. But those side effects only show up because projection is opening up the meta-system within which the system is posited as the ground for the capsule. Projection is parabolic, which is the balance between the elliptic and the hyperbolic. The spreading of side-effects is hyperbolic. The capsule is a place to put the genie as if in a bottle. But because the genie has been put in the bottle it can escape. The escaping of the genie from the bottle is like the proliferation of problems from Pandora's Box. Parabolic projection is the balance between complete out of control escape and imprisonment forever. Dasein splits the orthogonal projection into intentionality and causality. When it finds something in the bottle then that is when it expresses intentionality, and when it puts something into the bottle that is when it expects there to be a cause downstream as someone opens the bottle again. Intentionality is subjective and Causality is objective, the doubling of this bridge is the split between inside and outside. Projection itself is neutral between inside and outside.

One thing that I have been advocating for a long time is the idea that the difference between the

system and the meta-system is the difference between conjuncting the Godel statement with the system or not. If you conjunct it you get a whole greater than the sum of the parts and if you do not then you get a whole less that the sum of the parts. Now let us look at the capsule if it is within the system then there is a extra something hidden in the system that produces a different result via side effects than the system would on its own as just a state machine. If on the other hand the capsule is outside in the meta-system then there is no such emergent effect. The capsule is the place from which the oracle of the openly-closed system can give us information from within that does not come across the boundary of the system as Turing machine, and that is just what objects do they are encapsulated, but they can be messed with by round about means via other mechanisms besides get and put methods. And so it is possible to have not just storage of information that comes though the get and put methods but via other means that reach in behind the scenes to change the data stored in the object, that would be a representation of the fourth dimension as a special access route into the bubble of the capsule. Now basically the capsule is the same whether it is inside or outside the system as Turing machine. If it is inside it is embedded in the inner operating system within the system that allows subsystems to interact. If it is outside then it becomes a resource that the systems depend on for their sustenance, the boundary of the capsule is what allows the meta-system to remain separate from the system and thus to be independent from all systems that share the same meta-system. But because the capsule is the same whether it is inside or outside it is via the capsule that the information surreptitiously entering the system passes through. The capsule is the secret passageway between the inside and the outside of the Turing Machine System, it is because of this secrete passage way that openly closed systems can be modeled, and emergence can be simulated. Openly-closed systems described by Victor Frankl in somewhat different terms are the median between Closed systems and open systems. Both closed and open systems are immersed in a hyperbolic space which has the nature of entropy. Closed systems do not take into account context and are the darlings of physical science which treats everything as a closed system and thus a mechanism in order to try to understand it. But as Rosen shows in Life Itself entailment has a rich structure that will allow Biology to be understood within science via entailment structures that are circular. We can think of these as hypercycles which allow biological systems to maintain homeostasis. Open systems live in streams of energy that are far from equilibrium and can induce negative entropy or order and organization. They draw on the capsules as resources, as sources for what is needed like CPU cycles or memory or information radiators. We read information from the surface of the capsules. Information we would not know otherwise. So for instance in programming there is the model of the blackboard system which agents use as an information radiator. In the C2 system of Taylor there is the broadcast of self-talk along busses to which agents can subscribe. Open systems absorb this energy flow and other resources as well as information from external sources and via its internal negative entropy it uses these resources to build its own complexity using hypercycles as a means of self-organizing, and thus giving us an autopoietic system.

Autopoietic systems are symbiotic in as much as they are combinations of two dissipative ordering structures that are negentropic. So we can image two Turing Machine systems that have within each other the capsules of the other dissipative structure. Interlocking capsules allow the dissipative structure to become stable. Each one reaches into the other to give it information that it would not have otherwise from within about the other. Each one is in the meta-system or meta-machine of the other. They are entangled with each other forming a solitonic breather or a Kleinain bottle configuration and acting like Cooper pairs in superconductivity. A reflexive system is the combination of two autopoietic systems, or four dissipative structures. The dissipative structures that are negentropic are like what Deleuze and Guattari call Desiring machines, but we can see these as really Desiring/Avoiding, or Disseminating/Absorbing machines, and following Foucault see them as Practices rather than machines. But the metaphor of machine is apt when we are talking about State Machines of Turing. In this case we have two

tapes and two states, but we can have one capsule consulted inwardly by both as an oracle. Outwardly the tapes can use the petri net protocol for coordination. But inwardly both are getting information from nowhere about the other, and thus are able to act in Sync allowing ultra-efficacy and laminar flow in their behavior, what Bergson would call pure motion. Outwardly there is the reflexive social special system that embraces the symbiotic autopoietic special systems that are made up of conjuncted dissipative special systems. These four dissipative special systems have six possible relations, only two of which are embodied by the autopoietic systems which leave four as virtual. It is this virtual minimal system that allows the projection of the reflexive social special system as an emergent over and above the autopoietic relations between concrete dissipative structures.

Many of the interesting ideas of Deleuze come from Bergson. One of those is Intuition by which one realizes the nature of Duration. In intuition Bergson is exploring the same territory that Peirce is exploring with the idea of Preci**ss**ion. And this bring us back to our key point which is that we need to understand how continua exist and beyond that synergies that we see in the two tetrahedra that are fused into the tetrahedral diamond. [Beyond that of course is the vajra which places an octahedron between the two tetrahedra.] We have explored the petri net and state machine because they are what exist from a methodological perspective in the place where the entanglement of the hierarchies as components occur. They are dual bridges that are opposite the DARTS of Gomma which allows us to describe how agents interact as separate tasks, through messages through queues and using semaphores. DARTS is a single bridge that is the dual of the 'dual bridges' of Petri net and State machines. The tangling of perspectival components occurs exactly in this space of embodiment. So what is entangling in terms of the hierarchical relation of perspectival elements is represented as a state machine and tape, or petri net with colors (like Conway's Game of Life or Wolfram's New Kind of Science simulations that also can be Turing complete). So this excursion into attempt to understand the continua and synergies of the state machine and Turing machine and universal Turing meta-machines relates to the embodiment of designs with respect to the tangles. But also we need to go beyond this to understand how the tangles become rational knots via this embodiment.

The minimal methods arise from the duality between Linear without distance and Partial with distance which is the only separation point in the lattice of orders Klir calls Methodological Distinctions. Between each pair of viewpoints are either oneway or two way minimal method bridges. State Machine and Petri net is just one of these between function and event. DARTS crosses the same area between Data and Agent. But note Partial Order separates into Agent and Function perspectives as Full order separates into Event and Data perspectives. And then after this separation then the minimal methods are generated by the interaction of the perspectives. And so the crossing of the space in the lattice where the component hierarchy of tangles exists occurs after this splitting of the perspectives from the types of order on either side of the split in the types of order into Linear without Distance and Partial with distance. And as we have noted it is these types of order that stand in also for decoherence and delocalization effects in the source code which we are trying to avoid in the design, so the very pairing of orders that represent Decoherence and Delocalization also are the archetype for the differentiation of the dualities of the minimal methods that attempt to abstract above the level where Decoherence and Delocalization occurs. All the bridges between viewpoints taken together give us slices of the Turing machine. So Sequence Diagrams (Worldline and Scenaio) between Event and Agent, Use Cases and Virtual Layered Machines between Function and Agent, Dataflow and Object between data and function, and the spacetime intervals between Event and Data all exist as a frame around the central crossing duals of Petri Net / State Machine and Darts. Minimal Methods are precisely the minimal representations that can be repeated which are practical outlines of essential features above the threshold of Decoherence and Delocalization. They are engineering discoveries mostly

codified in UML and now also SysML. But these graphical representations only allow entities and their relations to be represented. ISEM on the other hand allow complex relations within single commonly structured language that are textual. As Frege said the unit of meaning is the sentence. If you want to have complex structures with meaning you need complex multi-relation representing statements. Wittgenstein in Tractatus on the other hand gives us the model of the world with Facts, each sentence representing a fact about the design. And of course we have Schemas Theory that says that all designs will be based on one or more schemas which has now been put into the ISEM language. ISEM gives a common regular syntax for complex design fact statements which allow the embodiment of Turing machine slices describing spacetime intervals between interval temporal logic and Entity Relation Attribute Diagrams that represent Space and Time. Then the other various Turing Machine slices build off of the way that the partial orders of Agent and Data relate to those relativistic spacetime intervals. But there is a split in the kinds of order that generate the archetypal duality that informs the production of the minimal method duals or self-duals. And exactly at that split is where the Decoherence and Delocalization in the Source Code exists, and where the tangled semantic hierarchy of components exists. Understanding this complex structure is at the heart of all design activities. UML and now SysML gloss over this conundrum by merely positing the various minimal methods but without explaining their relations and origin. At least if we can recognize that the minimal methods are slices of Turing Machines then it is possible to understand why they together can describe a computational structure above the level where Decoherence and Delocalizaton take hold, i.e. at the design level. Decoherence and Delocalization are like Quantum Mechanical effects, i.e. like entanglement and superposition, which we assume will become computing concepts when Quantum computers become a reality. But at the level of computing with Von Neumann machines in General Purpose Languages (GPL) these are the effects we find that embody the Differing and Deferring of what Derrida calls differAnce which is embodied in Software as a cultural artifact. Decoherence is like Linear Order without Distance, which is the effect that the actual distance between two lines of code in terms of execution is unknown. Delocalization is like Partial order with Distance in which we do not know when a given element will take precedence in the executional sequence given a lack of constraint but that the whole execution can be measured to produce a performance probability. Decoherence means that due to the Differing and Deferring we cannot be guaranteed at execution time that design elements will operate coherently as they were designed within the static design structure. Delocalization means that in spite of Encapsulation of Objects and Aspect oriented gathering of cross-cutting concerns we cannot be sure that Objects and Aspects are not smeared out both in the code and at execution time.

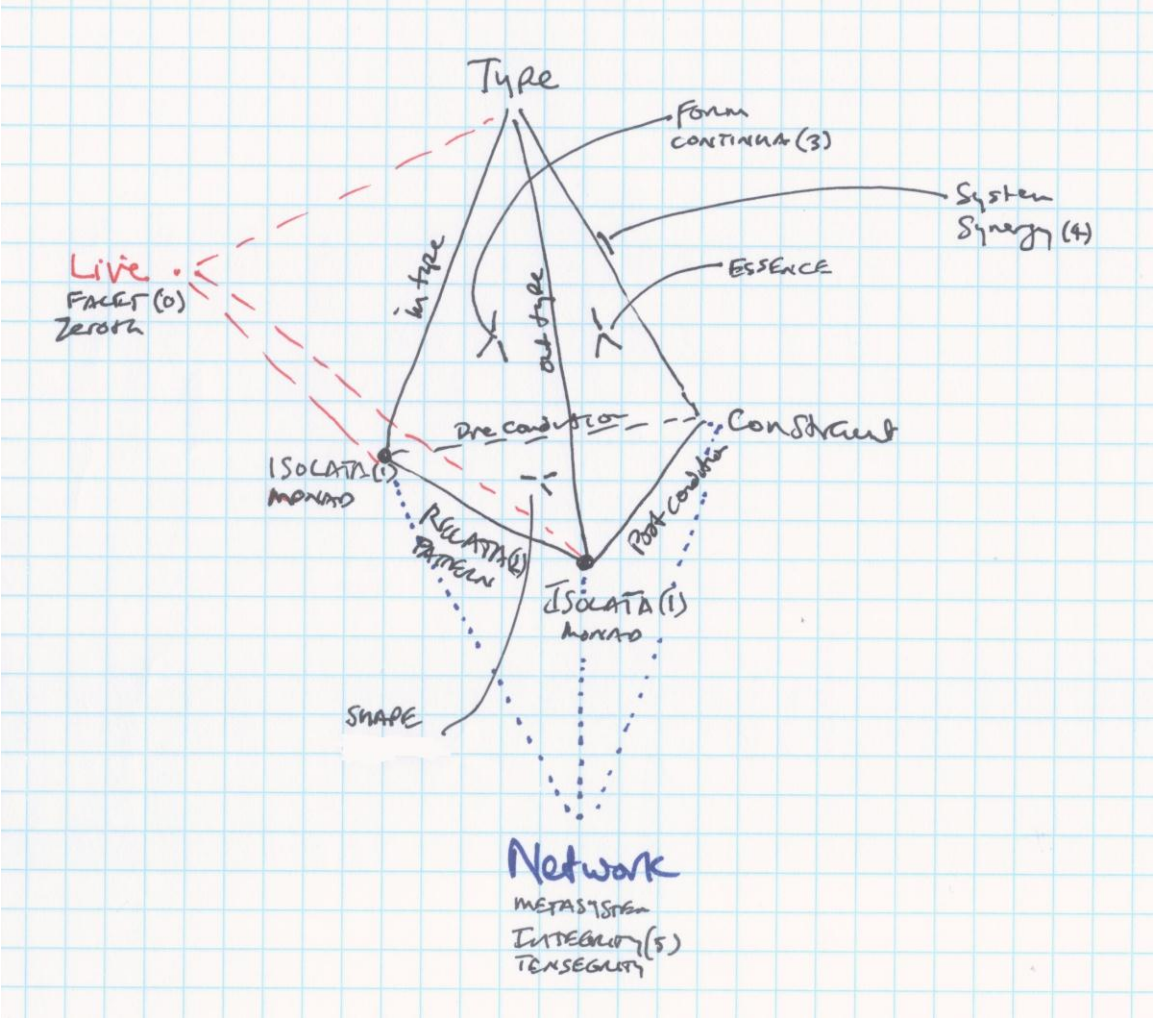
Part of the problem here is the duality between Set and Mass. We design at in Sets but we execute in Mass. Each has their own logic. Set logic is syllogistic which is the main logic of our tradition. But masses have pervasion logic (like Venn Diagrams, or a boundary logic) and so it is that separate logic that needs to be applied once we have compiled the source code into executables. By pervasion logic objects and aspects may pervade certain regions of the executables and this pervasion is their delocalization. Due to execution constraints any given line of code is broken up into assembly code or p-code within the virtual machine and many factors can intervene between the execution of one instruction and the next at the assembly level given task switching and context switching and other execution specific effects that are not foreseen by the programmer or software engineer when they are writing the code thinking about the design. These other factors only get recognized when the debugging is occurring due to an error in the compile or a glitch execution of the source code. These are not superposition or entanglement which are even more extreme phenomena related to super-rationality on the one hand and paradoxicality on the other, i.e. the limits of the divided line. Decoherence and Delocalization are breakdowns of schemas with decoherence being a breakdown in time and delocalization being a breakdown in space. At a given level of schematization we cannot trace to the lower level schemas without encountering

unanticipated breaks in the execution continuities or dislocations and discontinuities between lower level schematic levels, for instance when we go from Source Code text to assembly language operators and operands within an executable. From the higher schematic of source code syntactic patterning we see only a monadic mass in which differentiations we wish to make and can understand at the design level are lost. Of course, with sophisticated debuggers and other tools we can bridge this gap but it is extremely difficult and a single bug may take a long time to resolve. When we try to debug quantum computers that operating at the limit of our reasoning and understanding this will probably be even more difficult.

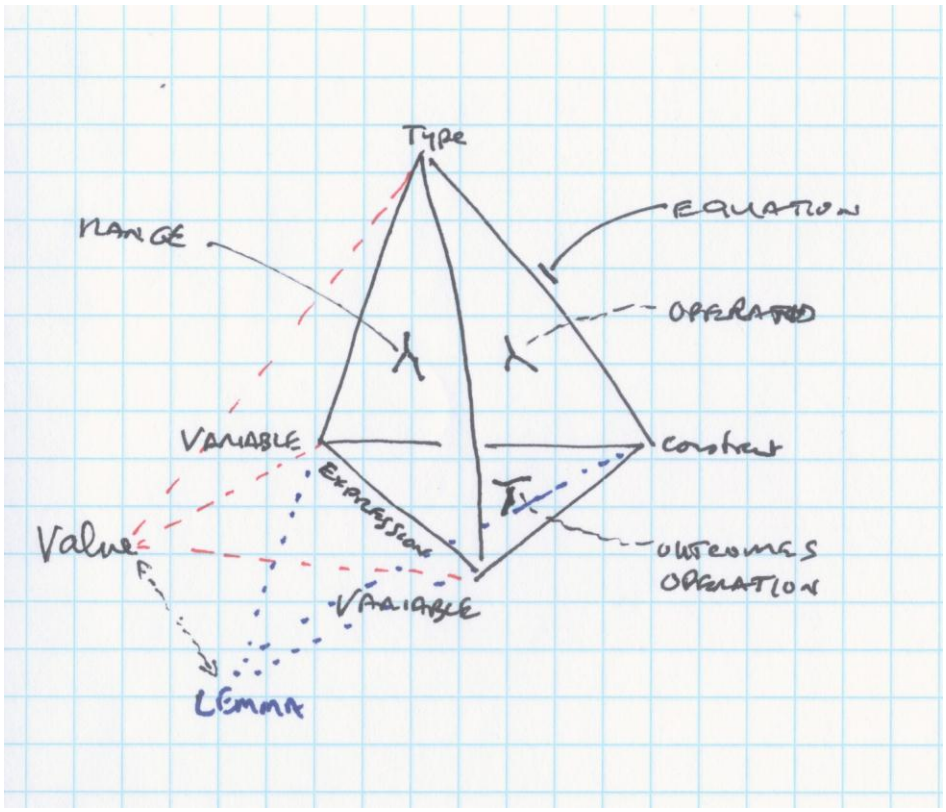
So the hierarchies of components are tangled but not entangled, and states are isolated and determinate not superpositioned within those tangles. But recognizing that the limits of the tangles that would appear at the quantum computing level helps us get our bearings in as much as we are dealing with a Gordian Knot which is within our reasoning and understanding territory not at its limits. However, that Gordian Knot can be inherently complex not just complicated depending on the cycling of causal circuits within it. The knot is a flow of gestalts rather than a single thread. And for this reason the relations between the duals of gestalt and flow, or proto-gestalt and proto-flow apply phenomenologically to our ability to comprehend the tangling. However, as we have proposed we believe that these tangles at the next level up in synthesis become knots, and thus take on the continuity and discontinuity of self-organizing knot structures. Thus the next emergent level these tangles do not just become tangles of tangles but instead become knots. Those knots are circles of interlocking causality in the execution flow that exhibits various isomorphies of cycles, and feedback/feedforward, and other specific isomorphies that appear in the work of Len Troncale who has cataloged examples of Systems of Systems Processes discovered by various systems sciences. It is these interactions of isomophies that Len Troncale calls Linkage Propositions that combine to create specific Systems signatures and can lead to combinations that do not appear in nature but which we can Artificially simulate in our systems simulations. I have at the 2012 conference given a briefing on my extension to the work of Len Troncale by producing a draft identification of the schematic levels at which various isomorphies come into plan and also by providing linkage propositions that relate to systems other than the system and especially to the meta-system. I have developed the hypothesis that the Systems Processes appear within the field of the Meta-system which is of course the de-emergent system. But many of the isomorphies and their associated linkage propositions operate in Software systems as well. I have marked the ‘Discussants List’ of the Isomorphies and see that either Isomorphies are directly related to software or simulatable. This means that these systems processes can be combined and new linkage propositions can be discovered in an Artificial Systems (Schemas) Simulation programme. It would be exciting to discover new types of Schematic structures that do not occur in nature but are possible though some sort of genetic algorithm based on isomorphies that generate linkage proposition sets that define new schematic cohesion possibilities. And of course we know that software systems demonstrate many different pathologies when various linkages between isomorphism break or become dysfunctional.

Generalization

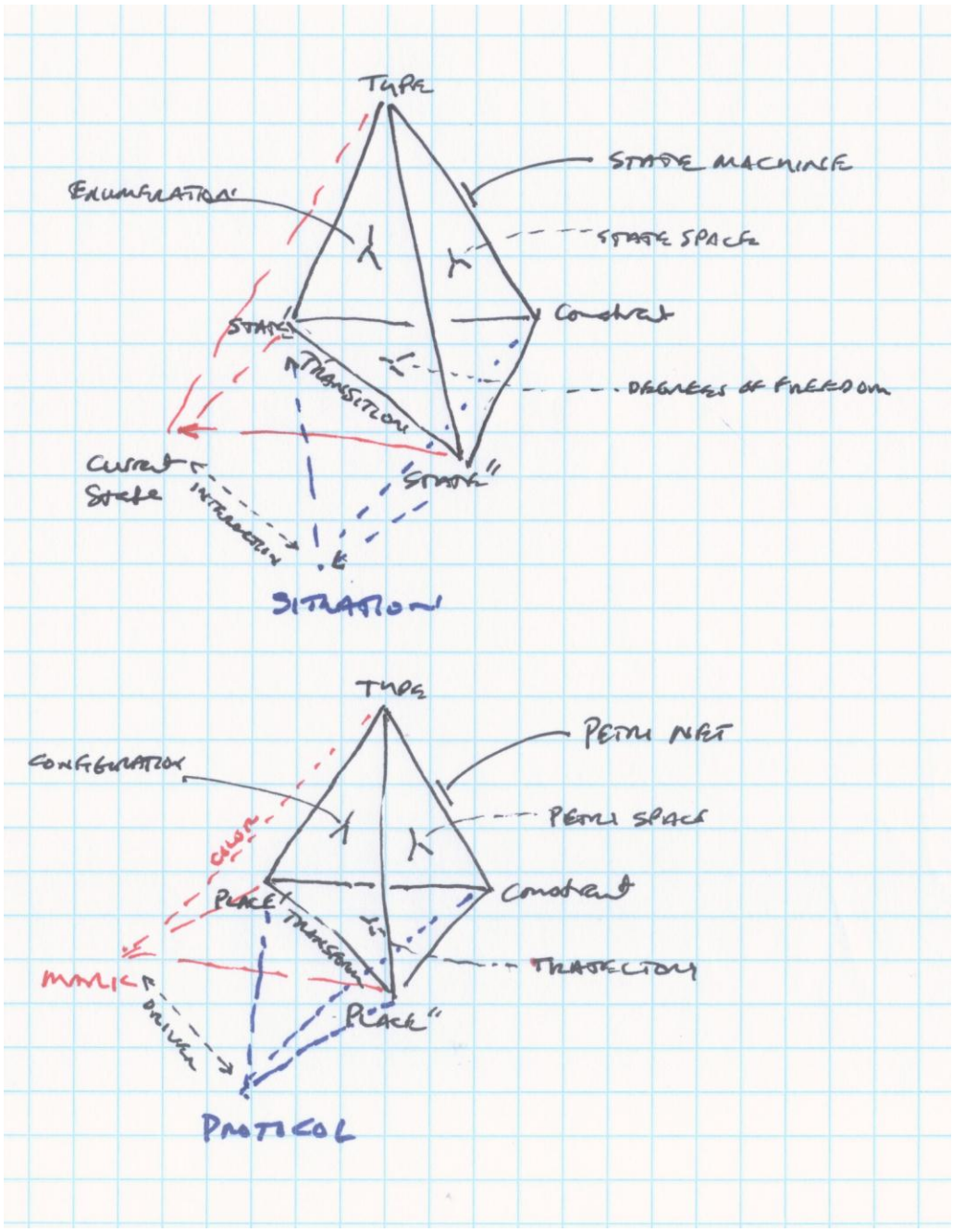
We can generalize what has been said above by aligning it with General Schemas Theory and the principles of Peirce and Fuller.



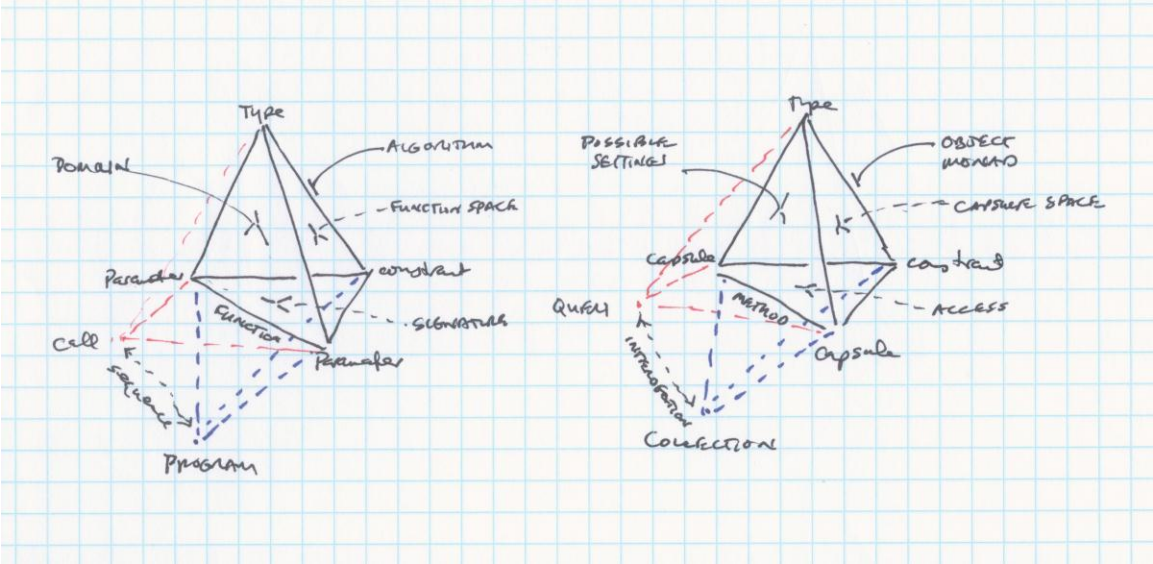
What we see here is that there is always a live aspect of the minimal system and if we align that with the facet which is zeroth principle then we can see the isolata as monadic points which has relata between them that form a line. The form appears between the intype and the outtype. On the other hand the Essence is the mediation between Type, Constraint and Isolata Monad. The shape is the mediation between the two isolata monads and the pre and post conditions related to the constraints. The two Isolata Monads that are repeated in relation to the Type and Constraint forms the System which has synergy. Network is projected from the shape as the meta-system with Integrity. On the other hand it is the form that projects the Live Facet. The representation of the Form comes from the pattern of repetition of the Isolata in relation to the type and constraint.



Here is a Variable and its relation to an Expression in the same format.

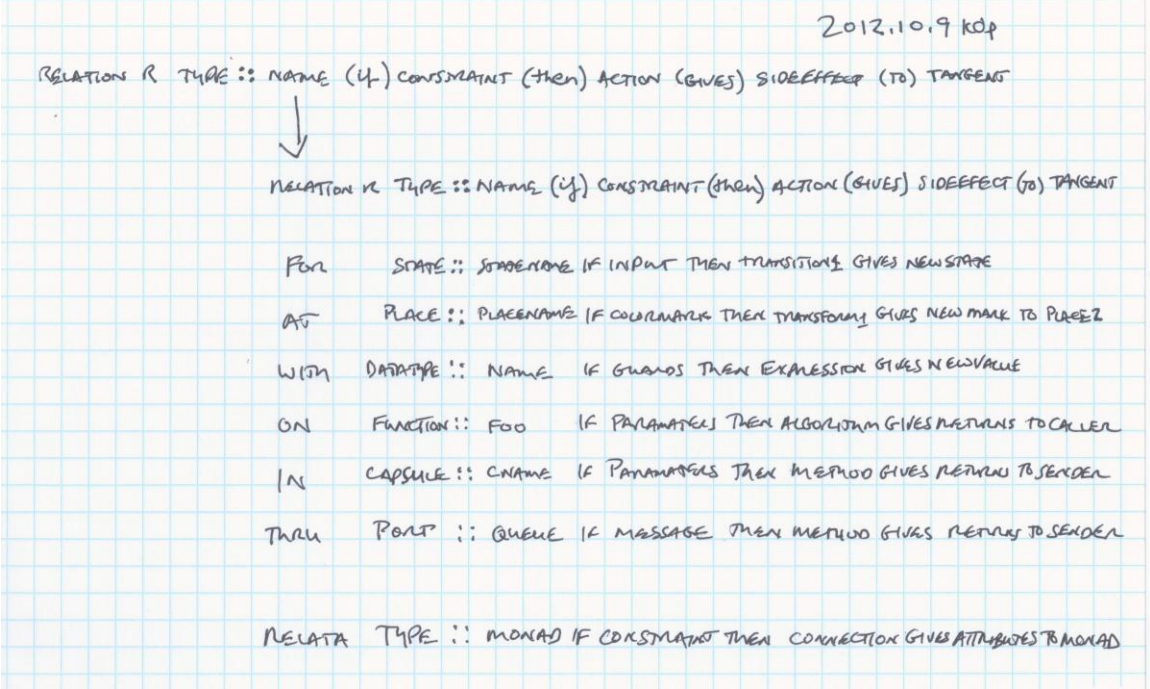


This is the state machine and petri net in the same format.



This is the relation of a Function and a Capsule in the same format.

This is the idealization as a field that describes the core of software in terms of the principles of Peirce and Fuller.



This is how the various idealizations could be formatted in similar template.

Schemata Language

We propose a new schemata language which would allow these relations to be developed with the following template:

Type Schema1 Type relational-Schema2 Type Schema3 if constraint then action gives side-effect to Type Schema4

So for instance:

Facet1 Monad Facet2 if constraint then action gives side-effect to Facet3

Monad1 Pattern Monad2 if constraint then action gives side-effect to Monad3

Pattern1 Form Pattern2 if constraint then action gives side-effect to Pattern3

Form1 System Form2 if constraint then action gives side-effect to Form2

System1 Meta-System System2 if constraint then action gives side-effect to System3

Meta-system1 Domain Meta-System2 if constraint then action gives side-effect to Meta-system3

Domain1 World Domain2 if constraint then action gives side-effect to Domain3

This new language is prototypical in as much as it can capture all the relations that we have been talking about when applying the Peirce/Fuller principles to understanding the State Machine, Petri Net and other constructs like the capsule.

By generalizing the language to schemas of given dimension we realize that one of the things that we get from the fact that there are two schemas per dimension is that these statements remain in the same dimensional layers even though there is a synthetic relational schema between two lower level schemas. But we also think that this could be reversed:

Pattern1 Monad Pattern2 if constraint then action gives side-effect to Schema3

In this case the two patterns overlap in Venn diagram style in the same monad. Also the last schema to which the side effect is assigned may be any schematic level, in other words we do not have to stick to a given dimensional level. For instance we might have . . .

Form System Meta-System if constraint then action gives side-effect to Pattern.

In this case the System-relational schema relates a Form to a Meta-system with a side effect assigned to a pattern.

The key is that the schemas naturally have the ability to subsume computing languages within them, but if we use the schemas directly then we can relate them to the principles of Peirce and Fuller and thus get greater leverage than we might have, and by having a single statement form we get the leverage that Gurevich ASMs have in terms of computability, but we have added to that the context outside of the left and right hand sides. We have added a schematic context and we have also separated out the side-effects from direct effects of actions.

Another point is that the hanging file structure relation between statements can be simulated by merely having the higher level object as the first element in the statement so that it is not necessary to have the file structure or mind map hanging outline structure in order to encode the Semantic relations. We are assuming that every schematic object can have its type and attributes.

For the state machine then we could have statements in the schemata language like this:

State1 Transition1 State2 if inputs then action gives State2 to self

A list of these transition relations between states would comprise a state machine.

Place1 Transformation1 Place2 if marker_colorX then action gives marker_colorY to Place2

Similarly a list of these transformations between places for markers would comprise a petri net. Expressing a protocol.

These statements take an if…then..else type rule and gives it a semantic context and a dynamic within that context separating out the side effects from the direct effects and giving more complex semantics by connecting multiple schematic objects through higher schematic relations.

One thing that comes out in this which is of interest is the idea that there is an interface between the actions and the schematic relations, and in fact we can say that action across schematic relation is causality, so this interface becomes very important. And in fact this interface relation is precisely what needs to be combinatorially explored when we attempt to relate the various pure hierarchies of agent, function, data, or event within the tangled hierarchy. There will be a separate relation for each possible element that can be combined that has to be defined explicitly. What we see in the schemata language is that two different schematic objects have a schematic relation though a higher order schema, but that this relation may be actualized by inputs that meet the

constraint, and if it is then an action is pushed across that relationship and the relation becomes active and side-effects of that action may be produced which either effect one of the objects in the relation or may effect a different object all together. From this a lot of meaningful structure could be created which is semantically rich without having a syntax that is as complex as the domain specific relations. What we are suggesting now is that the schemata language is the intermediary between the tangled hierarchy representation and the ISEM domain specific languages, and this intermediary language because it is regular like the if…then statements of Gurevich ASM language (ASL) that it is amenable to analysis that could attempt to prove consistency, completeness, and clarity and the other desired properties that we have been wishing to instill in the Domain Specific Languages.

Approaching again the Central Question

We have noted that the Component Hierarchy where the entanglement of separate hierarchies occurs is precisely in the same place where the State Machine / Petri Net // Turing Machine / Capsule structure appears. We can see the capsule also as the elements that allows DARTS to work, in as much as tasks inverted capsules (in which the state machine is in a capsule itself) and it is via the semaphores that allow the global data to be encapsulated with a safety mechanism that would allow the capsules to be shared among tasks. We have seen how we can apply the Peircian categories of firsts, seconds and thirds to the Turing Machine as a structure that is the basis of computing. We have noted that the tangles themselves must become knots at some level because it is the knot that exemplifies self-organization and is non-dual between continuity and discontinuity in as much as the knots have crossings. What is left for us to understand is how this occurs. How do we transition from our Domain Specific Design Language to the tangled hierarchies to tangled knots. And the answer to this is the one given in my Dissertation on Emergent Design (UniSA 2009), which is that the minimal system of B. Fuller that is the tetrahedron we see in the state machine, has other forms which are the knot, the torus and the Mobius strip. We have associated these with the whole form, the picture, the plan and the model in the dissertation that appear when we relate the representations with the repetitions between schematic dimensions. So for instance the torus is the whole form, mobius strip is the picture, the knot is the plan, and the tetrahedron is the model. The combination of the picture, plan, model and whole form as wrapped gift is the super-synthesis, which we unwrap to get the whole form back. It is impossible by mere repetition to create the whole form directly, rather we must indirectly produce it though the conjunction of it with the wrappings of the gift. So the Turing Machine is the tetrahedron. But another completely different representation of that is the knot which is made up form the tangles in the hierarchy when they are involuted, i.e. the tangles form closed loops. This would mean that the tangled hierarchies correspond to the plans. This leaves us to think about ‘pictures’ (Mobius strips). And perhaps we can think of the domain specific language representation as being written on these Mobius strips. The point of the Mobius strip is that it reconciles the global with the local. Each of the domain specific constructs has separate sentences that together form a whole description of something, like the state machine, petrinet or some other of the representations made possible by the minimal methods. In the design language we allow any fact known to be recorded, like a state, or a transition, a place or a marker. But then as we know more we can condense these descriptions so that we get a synthesis that has global meaning and that can be further condensed into a set of rules and then perhaps just a set of vectors. So the domain specific language constructs have this way of balancing the local statement about part of the minimal object with the whole of that object. This is very different from how the Tangled Hierarchies interact, and is even different from how the model of the state machine as tetrahedron works. All of these are various representations or repetitions of the whole

form which is like the torus. The torus is a cycle within a cycle that is fused together as a direct sum and thus as a synthesis. Programs are normally cycles within cycles within cycles. The cycle of the main program contains many different sub-loops and perhaps some of those loops are orthogonal to each other, for instance the loop of the operating system is probably orthogonal to the loops of the applications. So the running system is more like the torus which is a whole form of orthogonal or nested loops. But we cannot get to the running program without having plans that appear as tangled hierarchies, pictorial visions that reconcile local with global, and synthetic Turing machine representations. So after these machinations it turns out that the relation between the tangled hierarchy and the Turing machine with state machine and petrinet duals are easy to resolve the way it was done in my dissertation. The minimal methods are slices of Turing machines. But also besides the slices the Turing machine as a tetrahedral model minimal system also has other transmutations such as knots as plans, Mobius strips as pictures, and torus as orthogonal loops (from the scheduler of the operating system, to the main cycle of the application as well as loops within loops. So the central question of the way that the transmutations of the minimal system fit together has already been dealt with in the relation of knot, Mobius strip, torus, and tetrahedron that correspond to the plan, the picture, the whole form and the model. All of these together form a synopsis (super-synthesis) that gives rise to the whole form by the unwrapping of the gift rather than by construction.

This is fortunate because this structure has already been fully explored in my dissertation on Emergent Design. There it was shown that there are four different versions of the Minimal System which are the Tetrahedron (model), Knot (Plan), Mobius Strip (Picture) and Torus (whole form). The key is that these figures all contain 720 degrees of angular change within them in radically different ways, and it is precisely this amount of change (4piR) that is necessary to stand still in spacetime. We know that real time systems without global clocks are relativistic. So these various versions of the minimal system are what is necessary to model a relativistic system in a stable way, i.e. from an inertial frame. This makes a minimal system of minimal systems, which we now know represents the connections of the tangled hierarchies into rational knots, the tetrahedron of the Turing Machine, the Mobius strip of the Domain Specific Languages that reconcile global and local structures, and finally represents the torus of the operating system (meta-system) cycle with the individual cycles of the applications and their Main Loops. So this is a complete theoretical model of the real-time Software System from different inertial frames via fundamental transmutations of the minimal system. This of course is what appears as an A posteriori synthesis in the neo^2-Kantian meta-episteme model from the Software Engineering perspectives. This is the moment of views in the Emergent Meta-system cycle. The four categories of Kant is at the moment of seeds, the four causes are at the monadic moment which is the basis of the fourfold that expresses the relation of algebra/geometry, analysis/synthesis. Then we get to the four views which are Agent, Function, Event and Data, and finally at the moment of candidates we have the four moments of time which includes the co-now, or the virtual moment which is the fourth moment of time that puts us in the Heterochronic era rather than the Metaphysical era. The third moment of the Emergent Meta-system cycle is where the tangled hierarchies that become knots and the tetrahedron of the Turing machine exist along with the domain specific architectural design languages and the whole form of the computation that links universal (metasystem) and particular application Turing machines (system) together into a functional computational whole.