A Critique of SysML from the point of view of General Schemas Theory

The Application of General Schemas Theory

Kent D. Palmer, Ph.D.

P.O. Box 1632
Orange CA 92856 USA
714-633-9508
kent@palmer.name

Keywords: General Schemas Theory, Systems Engineering, Systems Theory,

SysML extends UML

Currently under development is an extension to UML 2.0 called SysML which attempts to make UML more useful for Systems Engineering. UML is the standard design language for Software Engineering. This paper is using Draft 1 version 3 of the SysML specification as a starting point. It is clear that the SysML is still under development so this paper seeks to give some background information and some arguments that seeks to influence the design of SML and ultimately UML 3 that might incorporate SysML.

Phenomenology of Graphical verses Textual Design Languages

Our first point concerns the whole idea of UML itself which has grown out of an attempt to consolidate the myriad design representations in Software Engineering around a single set of diagrams which could then be used by various methodologies. The goal was to attempt to keep the change in the Software Engineering tools to a minimum where each methodologist would use different diagrammatic conventions for very similar concepts. UML does not specify the methodology or steps by which the diagrams are applied, but attempts to give a universal set of diagrams congealed around the object oriented approach by which systems designs could be expressed when combined with a methodology.

The first point of critique is against the entire concept of graphical representation of designs as the only means of communication of design theories. Graphical representations are excellent for some purposes but they also have limitations that are seen in the UML language but are not often mentioned because this is the dominant paradigm. It is strange that we do not have text based design languages as well as graphical design languages and that there is no way to transform back and forth between graphical and textual languages. Perhaps one reason for this is that Textual design languages would put the Tool vendors out of business. This is because one would not need to buy fancy graphical programs in order to record textual designs. There is beyond this quibbling with the economic motives a fundamental fact about graphical representations of designs as opposed to textual representations of designs, which is that graphical representations are semantically poor compared with textual representations. We know this from the comparison to software languages. Graphical programming environments have never really taken off because the graphical representation takes up too much space and gives too little pay back in terms of leverage over the program being written. We continue to write programs in textual languages because of their combined compactness and semantic richness. But when we do Software Design we expect to be able to make the design visibly graphically but that graphical representation has little connection with the implementation other than through template generation capabilities in most cases. What is happening is that ULM in its second version itself is growing in complexity and as such the models are becoming harder to understand because of the number of graphical elements that are needed to represent a design is growing. Textual design languages are needed in order to represent the designs more compactly than the graphical representations and also in order to increase the semantic richness of the description of the design. But most of the textual design languages are formal languages which merely represent the next higher level from the software languages constructing a scaffolding around the design rather than representing it more efficiently. The key problem here is that a design language needs to be syntactically open, such that the designer can invent new semantic and syntactic relations on the fly in the process of design. This makes these languages very difficult to parse and to use because suddenly they are context dependent rather than context independent.

Thus it is an open problem how to create a context dependent readily extensible design language for systems and software engineering. An attempt was represented by the Integral Software Engineering Methodology (ISEM) invented by the author. This language took many ideas from Systems Theory and attempted to create a language that was exhaustive and methodologically complete. It still stands as a valiant effort to codify what was known about software design methods at that time and to present them in a form that was based on the most sophisticated systems theory that then existed. The challenge is how to take UML 2 and SysML and attempt to extend them so as to incorporate the advanced concepts from ISEM which are still not apparent in the literature on Software Engineering Methods and Design representations. But that new incarnation of ISEM also needs to be textually based to provide an alternative to the semantically poor, and monitor real estate expensive UML and SysML graphical paradigm. However, in order to introduce the advanced concepts of ISEM it is better to build upon the knowledge that designers already have of the design methods and tools that exist now rather than producing some non-standard alternative.

The whole problem with UML and SysML is that there has been no phenomenological analysis of the key problem of design. That key problem is the role of the kinds of Being in the Design process. We run right up against human finitude in the process of design, we can only see so much material at the same time, and then one hits a barrier in which showing and hiding relations become manifest between the designer and their design. How ever big the screen we end up navigating though the design eventually and that leads to our getting lost in the plethora of objects that appear in the design. We then have problems seeing the patterns into which these objects are collaborating across the large scale system. UML is fine for small systems but for large systems there is a fundamental problem of our getting lost in the designs which have become too complicated. Thus there is a horizon to the usefulness of UML for large systems despite the use of hierarchy to ameliorate these problems within UML. In this lostness during the design process we encounter Hyper Being. Pure Being is the whole design seen from Gods eye that beholds the whole all at once. Process Being is the showing and hiding relations that are made necessary by our human finitude. We have to navigate the design and remember what we saw in various places and construct a theory about how the design works in our heads. But when we get lost in the design then that is when Hyper Being is encountered. Hyper Being appears as our indecision about the design or our inability to formulate a theory of the workings of the design. Wild Being, which is the highest kind of Being, appears as the basic opacity of the design that does not allow it to be represented. We consign that opacity to Artificial

Intelligence and try to get rid of all the attributes of software that is indeterminate by that move. Software itself embodies Hyper Being which is what Derrida calls Differance. It is one of the only Cultural Artifacts in existence that embodies Hyper Being and so it is rather unique in that respect. It is interesting that this artifact that embodies Hyper Being is becoming ubiquitous in our society through the proliferation of machines controlled by real-time software. What we need to understand from a phenomenological perspective is the relation of the Kinds of Being to our Designs and how that fits into the relation of graphical and textual representations. This phenomenological analysis has not been done and that is the source of many of our problems working with the designs of complex systems.

Robert Rosen has some interesting things to say about the relation of complexity to simple machines. He basically says that living systems are non-computable and says that they cannot be reduced to Turing machines. He calls the aggregation of Simple Machines Complicatedness rather than complexity. The key point is that he says that there is something beyond Turing compatibility which we run into in Artificial Intelligence and Life which is the non-computable nature of some systems. Peter Naur says that there are theories of design that cannot be reduced to representations and can only be learned by interacting with the designer of those systems. It is human living beings that make the designs which are non-representable and thus probably non-computable as well. So it could be that in design of software and systems we are running into a fundamental horizon of non-computability when we are dealing with our designs. When we do design we are attempting to introduce synergy into the system we are designing in order to produce emergent properties. If emergent properties are described by Godelian Statements that are neither inside nor outside the System being described then it follows that perhaps the theories that embody those designs are themselves not computable, and it is through the non-computability that we introduce synergy into the system that leads to the establishment of emergent properties. If this is the case then we are actually facing a very difficult problem when we attempt to do design and produce representations of it. If software itself is an artifact that embodies Hyper Being then the design of that software must at least be one meta-level higher than the software itself, and that takes us into the realm of AI where there are variant opaque techniques, such as neural nets, self-rewriting code and other strange techniques that are incomprehensible to the human mind. It is strange that we call these opaque techniques Intelligence because phenomenologically for us concepts are transparent. When we combine different opaque AI techniques together we get even more strange and more deeply opaque software systems. So this appeal to the meta-level beyond software itself leads to the idea that designs are perhaps opaque. But the non-computability threshold actually comes at the fifth Meta-level of Being called Ultra Being where our own projections appear to us from the outside rather than the inside. Ultra-Being is when all the opacities of the various AI techniques are fused into one super-opaque singularity which occurs after we pass the threshold from Being to Existence.

When we consider UML and SysML in relation to this phenomenological analysis that appeals to Fundamental Ontology then we see that representations of designs have a very tenuous place in our understanding of designs. In other words, design representations are expressions of Pure Being, which is to say that they are formal and determinate assuming continuities of experience for their articulation that we relate to Parmenides view of Being. The representations themselves are produced by the design process which is an expression of Process Being which we know of as Heraclitian Flux. But as we design we are continually engaging in a showing and hiding process where we have to keep the whole design in our mind because we cannot see it all at once. So we are continually roaming around in the design representation, which in turn is situated somewhere in the design landscape of all possible designs. Hyper Being appears as the different possibilities that appear at each stage of the design. But it also appears within the design as its own indeterminacy where we have not yet designed or where we have lost the meaning of the design we have already done. Wild Being represents the opacity of the problematic into which our design enters and which we attempt to solve by our design. But that opacity represents the parts of Software that are rejected and are thrown into the vat of AI methods so that they do not infect our software with defects. Software defects are representations of these opacities that have not yet been scourged from our software. But still the AI techniques do compute something even if we cannot understand it. But when we take all the possible opacities and put them together to get Ultra Being then at that level we have something that we can not just no understand but which is utterly alien to us, because it includes all the isolated non-intelligible AI techniques in one conglomerate. When we cross the threshold from Being into Existence at the fifth meta-level of Being then we enter into the realm which Rosen calls utterly incomputable. Nauer warns us that our designs are representable. But if that is so then they are also non-computable, because what cannot be represented cannot be computed. But the reason it is important to enter into this realm of the non-computable is that it is there we garner the synergetic qualities that allow emergent properties of our systems to arise. But in order to create those synergies we need to move into an unknown land beyond representation and computability. Our representations and methods need to be understood against this phenomenological background that takes into account fundamental ontology in order to get the proper perspective on our design representations. In other words we want the best that representability can offer us but we do not want that to be a barrier to our own creativity by which we produce synergistic systems with emergent properties. What happens is that we extend UML to encompass new diagrams in SysML but we have no idea why these diagrams are right and not others. We have no fundamental theory of the relation between our representations and our designs. Why is it these elements seen in UML and SysML and not others that are right for design. We make UML extensible to allow others to add to it their own extensions and thus allow for their changes when they hit some limitation. But this does not allow us to understand the nature of methods and their relation to the diagrams that represent software or systems level entities within design.

Some headway in understanding this problem was made by the author in Wild Software Meta-systems where the theory of design methods was propounded. That theory said that there were four basic views of Real-time Software designs which included data, function, agent and event. Another view from requirements was added as an auxiliary. This was grounded in Klir’s theory of methodological distinctions with respect to ordering. It says that every design has five viewpoints on it, requirements and the four other design viewpoints. The minimal methods are the bridges between these viewpoints. And it is UML that more or less in a haphazard way approximates these minimal methods. The minimal methods that I discovered as central were data flow, virtual layered machines, mapping (use cases), darts, worldline and scenario, state machines and Petri nets plus those relating events and data. The coverage of UML of these methods is pretty good, and what we see is that SysML is attempting to balance out some of the distortions in UML that comes from an over emphasis on the Object Oriented Approach which got rid of all functional representations that were previously very popular. But the difference between the Object Oriented paradigm and the Functional Oriented paradigm is merely whether you look at data from the point of view of function or function from the point of view of data. The set of viewpoints and their methodological bridges explains the needed diagrams in a minimal representation of a software system based on a strong systems theoretic basis. As it is both the designers of UML and SysML are flying in the dark without such an explanation of why the elements that they have in their diagrams exist or not. They can only say that they seem necessary to different degrees. The theory of minimal methods explains what a method is, i.e. a traversal of the tetrahedron of viewpoints in a certain order. Mellor made the observation that traversing the viewpoints in a different order will result in radically different designs. So methods are important although they are left to the user's discretion by the UML language, and no mentioned in the document on SysML. Even though methods are more important than diagrammatic representations they are not being considered in the design of SysML. Methods are the means of our traversing the design landscape. Diagrams are our means of representing some aspect of what we see there. We need many diagrams so render a design in some region of the design landscape. There is an essential fragmentation of design representations because there is the singularity of Hyper Being at the center of the tetrahedron of viewpoints and minimal methods. That singularity amounts to the embodiment of Hyper Being that is the essence of software. If we are designing things with no software in them then at most we only have to deal with Process Being. But the fact that our systems contain software elements not merely mechanical and electrical elements then we are forced to deal with the nature of Hyper Being which is essentially difficult to think about and fragments our representations. Difference as differing and deferring appears in the interstices between the diagrammatic views. The theory of the software design has to connect the various representations and bring them into coherence in spite of the fact that they are fragmented essentially. None of the theorists attempt to explain why we need different diagrams to represent a system. Many times as part of viewgraph engineering we present very complex diagrams that attempt to convey the structure of the whole system, especially at the systems engineering level. But those diagrams are always over simplifications and are too complicated to be helpful in many cases.

If we understand that our representations function at a very low level of Pure Being and are attempting to get their arms around things at a very high level of Wild or Ultra Being then we can get a proper perspective on why the representations tend to fail to convey the essence of the design as Naur claims. The essence of the design is that which gives it synergy which is essentially non-computable but which is what allows the system to have unexpected emergent properties in the end. Human beings must to these designs because only they are capable of entering these non-computable territories where synergy can be wrested in order to produce emergent organization in the built system. If we were smart we would always keep this context of UML and SysML in mind. They do not even address the hard problems, they are merely scaffolding at the easy level were we can create representations in single diagrams. Things start to get hard when we are exploring the design landscape with these diagrams and when we are navigating our design, that is when we encounter Process Being. Things get harder when we encounter the necessity of bridging the gaps between our diagrams which makes us get lost in our own creations and when we encounter the possibilities within the design landscape as panoramas of possible designs. The current representations do not help us navigate the design landscape easily, and they do not allow us to represent our designs compactly so as to reduce the showing and hiding that is necessary to navigate the design. Textual design languages would help us in both of these respects. Textual design languages are more compact and pack more meaning per symbol into the text of the design description. By textual design languages I mean Implementation Specific Languages that implement minimal methods and mimic UML or SysML graphics. They make it easier to navigate the design just as you would navigate code you have written. It would be nice if this text could be transformed automatically into UML diagrams which in their expanded form might make it easier for the one learning the design to see how things related to each other. But for the practitioner who knows the design the most compact form is preferable. But textual languages based on the concept of the fundamental viewpoints and the minimal methods also help with another fundamental problem. It turns out that the different minimal method diagrams are slices of Turing machines. So they are describing the computable system being designed. That system being designed embodies a singularity of Hyper Being, and the diagrams of the minimal methods attempt to abstract views of that singularity that can be understood. The computable system being designed has a certain synergy that is encoded into it from a viewpoint beyond computability, that is why we can produce emergent effects within such a program. If we could not step outside computability then we could not produce emergent effects. So the fragmentation of the minimal methods as diagrams and the traversing them is the means by which we move to all sides of the singularity of Hyper Being in order to get views on all sides of it and thus we are able to project our synthesis of our design onto the implementation which will embody that singularity in programming code. At that level there are many displacements of elements that at the design level are compact that are smeared out though the actual implementation of the system in code. The reason we have design languages is to attempt to view the system as a whole without these warpages that are induced by our implementation languages. When we move into the arena of non-computability then we encounter the opacity of Wild Being. That opacity appears as implementation specific techniques that we must resort to on many occasions to actually get the implementation to work. Many times we see these in our algorithms as incommensurable discontinuities which we find ways around we call hacks. Hacking is programming in the face of Wild Being, introducing work-arounds and ingenious solutions that are local to the problem we are trying to solve. These work-arounds do not appear in our designs, except perhaps as exceptional patterns of objects.

We have noted that there is another perspective, that of requirements. That appears as another method called the Gurevich Abstract State Machine. It is based on representing the system as a set of rules. Rules as if…then… statements contain all four viewpoints within them and thus the rule set that represents a system instead of fragmenting the system into diagrams fragments it into rules. The written requirements are turned into a Gurevich Abstract State Machine model and it is this model that is the dual of the design. It stands between the logos of the written requirements collected in a requirements database and the design made up of minimal methods. There is a series implicit here from:

?? Requirements = Logos
?? Gurevich Abstract State Machine
?? Minimal Design Methods
?? Implementation = Physus

Notice that we are traversing between physus and logos. Physus is the unfolding of the executing implementation. Logos is the description of the system and what it should do. We can get views of the requirements using formal methods which are extensions of logic. In essence we use the three aspects of Being called Presence, Identity and Truth to construct these formal models. They introduce the properties of clarity, completeness, and consistency. However, we must add to these aspects yet another aspect called Reality. With reality we get the additional properties of Verification, Validation, and Coherence. This is where our need to do verification against the requirements, validation against the operational environment of the user, and the coherence of the integration of the pieces of the system all come from. Reality in model theory confers meaning. So it is only when the formal methods are introduced to the regime of testing that meaning of the system is generated.

Verification, Validation, and Integration all come after the implementation and are considered the latter part of the lifecycle. The formal models of the requirements that we might produce with formal methods are rally scaffolding around the system that assures the ranges of inputs and outputs and perhaps flows truth values over the scaffolding to attempt to assure that the formal qualities of completeness, consistency and clarity are satisfied. However, with a Gurevich Abstract State Machine we do not need this scaffolding because we discover those properties about the system itself as represented in a Turing Compatible form at some high level of abstraction. Once we have that Turing compatible representation then we can undertake the design using different minimal methods and looking at the system from different fundamental viewpoints. But eventually implementation must occur were we engage in hacking around the problems that prevent our design from working at the level of code. Then once we have a computable representation then we engage in testing that implementation against the original logos of the requirements, against the operating environment of the user, and in terms of its own integration which belies its coherence.

But all of this takes place in relation to a designer exploring a design space in order to find the synergetic sweet spots that will allow emergent properties to unfold in the implementation once designed. That designer is engaged in a non-representable and non-computable enterprise. It is under sanding this context of design that we need to concentrate on because this is the hard unsolved problematic with which we are engaged. Deleuze talks about the opposite of representation in terms of the idea taken from Lacan and ultimately Freud of *repetition*. Repetition is that which does not repeat. In other words repetitions never achieve the goal of encompassing the singularity that they attempt to approach. Repetition is the rubric under which Deleuze talks about the whole problem of non-representability and non-computability in <u>Difference and Repetition</u>. It is an extremely difficult subject that appears at the Meta-level of Wild Being. Deleuze and Guattari are building a philosophy at the level of Wild Being and so it is very pertinent to our exploration of these issues. But we don't just want to apply the theory of Deleuze to the problem, rather we need a more fundamental way of approaching the problem which attempts to understand the role of non-representability, non-computability, synergy and emergence in the design process. That will allow us to get a context in which to understand the UML and SysML languages and see how far they go in solving the problem that faces designers. A fundamental question that must be addressed is whether Systems Engineering can follow Software Engineering by merely extending UML 2 for their own purposes? Will this address the fundamental concerns of Systems Engineering that must join hardware and software and other elements into an emergent system or system of systems? Is SysML the right direction for Systems Engineering to follow and is it enough?

General Schemas Theory

We will not introduce a new theory that will attempt to paint a larger context in which to understand the SysML/UML quandary. This new theory is a further generalization of General Systems Theory. We engage in Systems Engineering and Software Engineering building "Systems" but we hardly ever are very definite about what a "System" really is. System as a word is applied to everything and thus has very little meaning. In order to understand the meaning of a system we need some context within which to understand it, and that context must be all the other schemas of the type that a system purports to be, such as form, pattern, eco-system, domain, world, etc. There are a series of schemas that form a scalar hierarchy and the term system designates a threshold of spacetime organization near the center of this hierarchy. If we want to understand the nature of what we are designing a very good place would be to start with the emergent hierarchy of the schemas and to understand the relations of the schemas to each other. So General Schemas Theory does what General Schemas Theory of Bertalanffy does at an even higher level of abstraction. That is to say it searches for embodiments of the schemas in various disciplines and then attempts to establish functors between uses of the same schema in various disciplines. General Schemas Theory will consider the natural transformations, i.e. meta-functors between schemas. There is naturally something higher than the General Schemas Theory that looks for modifications of schemas. These modifications appear in the difference between General Schemas Theory and Special Schemas Theory. Thus we can use Mathematical Category Theory to distinguish the various levels of Abstraction that differentiate the schemas within a discipline, from the schema across disciplines, from the differences between the schemas as such, and their modifications. But in this article we will not deal with the higher level meta-categories. Rather we are interested in the schemas themselves and how they relate to other important elements of our worldview that makes possible emergent system design.

A fundamental distinction already made is between physus and logos. Logos are unfolding thoughts and words, i.e. design, and physus is unfolding things, such as executing operational systems. This is a fundamental distinction within our worldview. But we can go on to think about the relation between this dichotomy and the non-dual of order. Order appears both in the realm of words and the realm of things. Order is the bridge by which we relate our theories to the physical world in science. The production of the non-dual of order is Mathesis. But we must ask what the meta-level of physus and logos might be. The logos of logos is grammar. The physus of physus is the laws of physics. But there is also another question we might ask as to the nature of the physus of logos and the logos of physus. The physus of logos has to be logic since it is the inviolable necessity which if we violate our speech becomes non-sense. On the other hand the logos of the physus are the schemas. And here is where we see the rub. We know a lot about Mathesis and Logic but we know hardly anything about the schemas. Schemas were introduced into our tradition by Protagoras and Plato, and Aristotle, Kant and Heidegger all thought about them. But the schemas have not been developed in our tradition to the same degree as the Logic or Mathesis. And so that is why we are having a hard time coming to terms with our need to understand systems engineering methods and languages. In systems engineering we are doing design which is a projection of order on to things. Schemas are an intrinsic projection of orders involuntarily onto things at the same time we project spacetime onto them. These intrinsic pre-orders are exactly what we use when we engage in design. They are the unconscious basis of design, that we project prior to the design on to things and which the design follows as we project it.

Between Mathesis and Logic is Mathematical Model Theory. Between Logic and the Schemas are the Philosophical Categories. Between Mathesis and the Schemas are Representation and Repetition Theory. So from that we can see where SysML and UML fit. They are representations of forms within systems. Each diagram relates several forms to others by specified relations. Since a system is defined as a set of things and their relations that is why the diagrams are effective in showing the internal structure of the system. But these representations such as UML 2 and SysML only function in the wider context of this ontic/ontological triangle. Unless we bring to bear the other parts of the triangle then we will never understand the true situation in which our design of UML 2 or SysML must take into account to be successful.

What is necessary is that we start over and attempt to understand the full context as it bears on the UML/SysML representations. The first point to be made is that the system and form are not the only schemas that exist.

UML/SysML are focused on form and their relations and by that hope to portray the internal workings of the system. This is not based on any well worked out systems theory, such as that of George Klir in Architecture of Systems Problem Solving but in stead has no real foundation. It is just an intuitive guess that these are the diagrams that are helpful and that it is this set of diagrams that will portray everything that needs to be said to give a full picture of the design of a system. It would be nice if there were a more solid foundation to UML. As it is the new version of UML merely cleans up the meta-level of the language and attempts to solve some of the inconsistencies in the model, also incorporating real-time elements from Objectime. But the actual elements that are present in the language are merely those built up over time as historical development and personal opinion of the designers of the language based on refinements of previous methodologies. No real argument is ever given why these elements and not others. It turns out that from a practical stand point the elements are very close to the set of minimal methods because of the pragmatic testing over time of these diagrammatic techniques. However, pragmatic testing is endless and the refinement process may take a long time, and there is nothing to guarantee that something essential is not missing. Thus it is amazing that there have been very few studies of the foundations of these techniques with an attempt to give deeper reasons why individual elements are included or not included. Founding the techniques of Systems Theory as I have done in the ISEM language is one way to make sure that the diagram set is complete and consistent rather than merely an ad hoc combination of elements. The basis which I have attempted to give the methods is based on the idea that the diagram offer slices of the Turing machine and that is how they can represent a computable system. But the Gurevich Abstract State Machine and the Minimal Methods are duals of each other, and it is that duality that helps to show completeness. Understanding the underlying computational mathematics of the Turing machine and how it is generalized in the minimal methods could provide a solid basis for UML/SysML.

But there is a more general problem which is that form and system schemas are not enough. This is to say that there is a whole hierarchy of schemas rather than merely two and if we only use form and system then we are severely restricting our toolset unnecessarily. What we must realize is that there is a whole hierarchy of schemas which include the following emergent hierarchical levels of organization:

?? Pluriverse
?? Kosmos
?? World
?? Domain
?? Open-scape (meta/infra-system)
?? System
?? Form
?? Pattern
?? Monad
?? Facet

This set of levels is a working hypothesis but the key point is that each one has its own organization and presumably its own minimal methods and viewpoints that are true to its own organization. A really robust methodological tool set would allow the designer to use any of these schematic levels to support his design as necessary. Most important of these levels from the point of UML is that of Form and System. This is because software is essentially a pattern of ones and zeros. But applying syntax via the technique of Chomsky we are able to raise ourselves up from the pattern level that software naturally resides at to the level of form. Languages define software operators and statements which give form to the pattern of monadic ones and zeros. Through languages we then abstract up to the level where we describe the design in terms of objects or functions. We produce relations between these objects or functions in order to create systems. Implicitly these software systems interface with software environments and hardware environments when they execute. These software and hardware environments are called meta-systems or infra-systems which together form open-scapes. One thing that comes out very clearly in software engineering is the important role of the "operating *meta*-system" [sic] as the environment for computing application systems. Fortunately there is a good formalism for this distinction in the difference between the Turing machine and the universal Turing machine. The universal Turing machine reads other Turing machines from tape and executes them. The difference between the Turing machine and the universal Turing machine as operating meta-system is that the operating system does not stop. Instead of a halting problem there is a problem if halting occurs, i.e. the computing system as a whole must then be rebooted. So in the computing field there is a good formalization of the difference between the system and the meta-system. But generally in language there is no good word to describe the meta-system. Thus I have coined the word open-scape. I define the open-scape as the environment or eco-system with niches that the systems fit into. It is a panorama seen from a single point in the landscape which has only one horizon. It is the context where the system is fixed within the environment. The open-scape is composed of the meta-system and its dual the infra-system. The infra-system is composed of the system and super-system envelopes and the possibility of holes in the meta-system. The meta-system is the gap between the system and super-system envelopes. As the meta-system is what is seen looking out from the system toward the panorama of its environment the infra-system is what is seen looking inward. We posit that in the nesting of the sub-system, system and super-system envelopes that the meta-systems are the gaps between these envelopes but looking outward from each lower nesting. The infra-system is the inverse of the meta-system and actually encompass all the elements not associated with the gaps between the hierarchical nested system envelopes. Thus a system exists in a meta-system provided by the super-system just as the sub-system exists in a meta-system provided by the system. In each case the higher system provides the operating meta-system within which the lower system exists. The infra-system relates the interfaces between meta-systems to each other and to the holes that are not niches within the meta-system. When we have both views, the meta-system view outward and the infra-system view inward then we have the open-scape which balances the panorama from the viewpoint of the static system within the meta-system with the interfaces and holes in the meta-system[1].

What is key is that we have a good view of the relation between systems and open-scapes (meta/infra-systems) and that we also have a view of the higher level schemas such as domain, world, etc. The systems that we are building today need more than the system and form schemas in order to support their complicated nature. But more than that, the synergetic effects that lead to emergent properties must take into account the organization of the schemas themselves because it is the schemas that underlie our designs. The schemas are projected on nature as ways of pre-understanding it involuntarily by us. Science is the process of looking for the anomalies in our projections and attempting to understand the order of things that is beyond our projection of the order of the schemas which we presume. But when we design the anomalies only show up when our designs interact with their operating environments (i.e. meta-systems). So there is a stronger basis for the schemas in design than within science. Of course science and engineering are intertwined. Engineers build the instruments that scientists need to perform their experiments. Engineers use scientific principles as a basis for their designs. Building technology and the knowledge of nature garnered by scientists go together and are interdependent. There should not be a

[1] See "A Dialectical, as well as Geometrical and Algebraic, Model of System and Meta-system synergies." SETE 2004 at http://holonomic.net also see "Towards a Possible Approach to Metasystems as Escapements : On a Simple Geometrical and Algebraic Representation of Emergence"

master-slave, i.e. dualistic, relation projected between engineers and scientists. Rather building technology and collecting knowledge of nature are complementary enterprises. But in both cases we must know our own projections of schemas which are developed in individual disciplines and are not usually generalized across disciplines. General Schemas Theory attempts to follow General Systems Theory in developing cross discipline generalizations for all the schemas and then goes on to compare the various schemas to each other. Systems Engineering needs to become Schemas Engineering just as Systems Theory needs to become Schemas Theory. In other words we need the higher levels of emergent organization as a resource in our building of more and more complex designed artifacts of global reach. So the first and most fundamental problem with UML/SysML is the fact that it does not treat higher schemas that are needed, but crucially it does not separate the meta-sytem from the system in a clear way and also does not represent the domain level. We need to step up the levels of schemas one at a time. If we had a UML/SysML that was grounded on the understanding of the relation of our diagrams to the Turing Machine, then we could build on that to understand how to model environments and eco-systems of computing and physical systems and then continue to understand their operation in domains. At the level of the world is where the interface with humans becomes crucial as Heidegger teaches us in Being and Time where he differentiates between Ready-to-Hand and Present-at-Hand modalities. Heidegger calls the human dasein (being-in-the-world) which we can generalize to being-in-the-schema. It is the ecstasy of dasein that overflows into the higher dimensional schemas which are the higher organizations of our world and which allow organizational structures that can carry the load of greater complicatedness that we find in our systems. It is also these higher dimensional structures that are the secret to the greater synergies we need to infuse into our designs. However, this means that we must transition into the non-computable realm identified by Robert Rosen associated with life in order to reap those synergies. The design space is that non-computable landscape within which the synergies of emergent systems appear as sweet spots. We need to be able to explore that landscape of design in order to find those knees of the curves of parameters that are optimal and our access to those non-computable higher dimensional spaces are the higher schemas. The meta-system marks the limit of computability with the form of the Universal Turing Machine. As soon as we pass into the level of domains then we find a fragmentation of perspectives which is non-computable. We gain multiple perspectives when we move in our environment, and it has long been known from the time of Zeno that movement is inherently contradictory. Thus there is a wall that prevents computation on a grand scale beyond the meta-system level. Rather computation breaks up into fragments called agents which live in networks of connected computers. This breakup becomes even worse at the level of world, where formalisms breakdown, and produce a fundamental barrier between Artificial Intelligence and Human Intelligence. Human intelligence is transparent, because it is a projection on the world that comes from us, while Artificial Intelligence Techniques are inherently opaque. Live, Consciousness, and the Social can be mimicked but cannot be reproduced due to an intrinsic barrier to computability. They in fact live in a space of non-computability. But the fact that it is not computable does not mean that there is no structure there. In fact there is higher dimensional organizations of the schemas that give these higher dimensions transparency to our understanding in spite of non-computability. When we do design we are facing this frontier of the non-computable and attempting to find niches in which computation can help and support human understanding. But now we are operating blind because we do not understand the intrinsic nature of the schemas and their differences with each other and their relations to each other. Rather we are facing a world with the ideal that everything can be reduced to the computational metaphor. We build larger and larger more complicated systems and produce synergies of such systems. But they are inherently difficult to build and to operate and maintain. But when we manage to get them to work with their emergent properties in tact then we get high leverage from them in terms of the global reach of our action and other substantial benefits. But I believe that if we understood the schemas and their nesting better then we would be able to build synergistic emergent artifacts using multiple schemas better and that they would be more useful and give greater leverage with less harmful effects on the greater environment and higher levels of organization of life on the planet.

So the fundamental critique of UML/SysML is that they only represent the Form and System schemas rather than all the various schemas, and that they have no foundation, but are ultimately an arbitrarily collection of diagramming methods that are merely historically and pragmatically engendered. The tie between the diagrams and the minimal methods can be achieved as I have done previously with systems theory and the concept of the minimal methods as slices of a Turing machine. Building on that formalism then we can extend to the meta-system by using the duality of the Turing machine and the Universal Turing Machine. But then we have to recognize that computability itself deteriorates at the level of the domain and world and other factors become more important. For instance, there is Aspect oriented programming which attempts to solve some of the problems that are domain related. Various work arounds that allows us to confront the fundamental non-computability of higher dimensional schemas need to be explored and understood in a context of General Schemas Theory that recognizes the intrinsic nature of the various schematic organizations. Non-computablity first appears as the fact that computing cannot be context free. It must take into account the meta-systemic environment. Then non-computability appears as the indeterminateness of swarms of agents with different viewpoints acting together that we get at the domain level. Finally at the level of the world we get the structure of dasein who projects Being as intelligibility. Intelligibility is not the same as computability. This is the fundamental problem with Cognitive Science which wants to see the human mind through a computer metaphor. But computability is a reduction of intelligibility. We can see that clearly because the AI techniques compute something but that process is always opaque to us. So computability becomes more and more alien to us as we add AI techniques together. From this premise we get Venge’s idea of the Technological Singularity which is the point where the Alien Intelligences of the computing devices becomes more intelligent than we as human beings. This assumes that there is a class of intelligence that encompasses both the alien opacity of Artificial Intelligence and our own founded on our nature as dasein. This is a premise that has not been proven. Beating chess masters by searching all possible moves is very different than beating them based on insight. Insight is the nature of our intelligibility that we project on the world primarily on the basis of the pre-projection of schemas. Outsight might be used to describe the alien intelligence based on opacity, if that is what it is. All computations based on parameterized functions are merely external mimicry of intelligence. But they are not real intelligence outside the narrow bounds of the computational problem being solved. Change the environment and the situation and the apparent intelligence vanishes. Human beings have a robust adaptability that is based on intelligence part of which comes from the projection of the schemas which are higher dimensional organizations on to a lower dimensional spacetime environment. Our ability to step up through the dimensions, meta-dimensions (standings), meta^2-dimensions (aspects) and even higher meta^n-dimensions is what singles us out as human beings in as much as we are freed in an ecstasy into higher and higher meta^n-dimensions that are finite but on the background of an infinite expansion of those meta^n-dimensions. It is our ability to stand off in the next higher meta-dimension that allows us the intelligence that cannot be achieved in what is computable. Computability is trapped at the level of system and open-scape at the level of the dimensionality transposed into the emergent levels of the schemas. But computability cannot span even to the higher schemas less well into the standings of the meta-dimension or the aspects of the meta^2-dimension, etc. Intelligence is able to move within the meta-dimensions in a finite way and that is what makes us unique and what makes it so cognitive science can never reduce us to the computational metaphor. This is important because as designers we enter the non-computational and non-representable realm and come back with representations and computations that have synergies that allow emergent properties that would not exist otherwise and are not produced in nature. We can only scale into the non-computational and non-representable realm because we have the intelligence that allows meta-dimensions to be transparent to us. The designer confronts these meta-dimensions all the time and we must understand that if we are to design tools and techniques to assist the designer. There is a hope that someday we will be able to produce designs with genetic algorithms. Koza shows how he has done this in the realm of antenna design reproducing some patents and creating new possible patents. But we can only reduce a realm to computational design using AI techniques like Genetic Algorithms because we can see beyond these techniques to define the realm of the problematic within which these techniques will operate. This seeing beyond is based on our finite climbing of the infinitude of meta^n-dimensions, as schemas, standings, aspects and other structural layers of the Western worldview.

Philosophical Category Theory

Once we understand the horizon of the schemas we can begin to move around the ontic/ontological triangle and attempt to say what the implications of each part of that triangle is for the UML/SysML representations. We will move around it in the opposite direction from the representations so we run into the representations last. So the next point is the relation of SysML to the Philosophical Categories. These are the highest concepts and have been identified by Aristotle and Kant. They include things like causality and part/whole relations. It is the categories that connect Logic to the Schemas. That is because the schemas represent pre-synthesis of things, i.e. the ontic, projected from the ontological and logic has variables in propositions that need to be filled in. So the schemas represent what the place holders that might be filled in. But more than just logical relations may hold between things in the world so it is the categories that specify these highest level relations like part-whole and causality that are relations that might hold between things in the world. The philosophical categories are in fact not settled, in fact they have been an ignored part of philosophy for the most part. The best recent category theory is that if Ingvar Johansson in Ontological Investigations. But we must always refer back to the categories of Kant and Aristotle to keep our bearings. Kant said that the schemas were the temporalization of the categories. That is a very important point because time and space are a priori projections for Kant. The schemas are to him the refinement of that projection of space and time in line with the highest concepts, i.e. the highest tools of intelligibility. That is why we call the schemas templates of intelligibility. But it is in the schemas for Kant that time is mixed with pure Being of the categories. This is a very interesting point. Johansson on the other hand assumes spacetime and the very next category is the state of affairs. So Johansson is skipping over the role of the schemas, because we would say that the schemas are the dimensional differentiation of spacetime and that all states of affairs appear already within particular schemas. But all dimensional objects are anamorphs because they are in two schemas at the same time. The different organizations of the schemas nest into one another in such a way that there are two dimensions for every schema as well. But since the category scheme is unsettled and none has been selected by the designers of UML and SysML then it is uncertain what the structure of intelligibility is at its highest level. This is another grounding problem not even solved by applying cognitive science to the problem of design. Cognitive science itself is ungrounded in this respect also. If we do not know the fundamental shape of intelligibility then how can we design tools to help us make designs more intelligible. Implict in these tools are a philosophical stance on the shape of intelligibility and that stance is more or less a reductionist one. In other words it assumes that we must limit separate the methodologist from the designer. Designers must follow the instruction of the methodologist in a master-slave dialectic. Thus the creativity of the designer is constrained from making up new tools on the spot to solve problems encountered in design. This limitation of the creativity of the designer to both design his object but to design the means of designing is not necessary. We are instead in favor of user extensibility of the diagrammatic tools and the methods. UML allows extensibility but it assumes that that extensibility is for the methodologist not the designer. On the other hand we believe that every designer must also be a methodologist. That is because methodology means “the way after” in other words it is a method for leaving a trail so others can follow you, it does not tell you how to cut the trail. Feyerabend was right about this point that in effect *anything goes* with respect to methodology in science. This is the ultimate pragmatic stance. It is precisely the stance that is needed to solve really hard design problems and placing artificial barriers before the designer does not help but rather hinders his progress.

So it is suggested that UML/SysML designers, Methodologists, and designers need to be aware of their categorical assumptions about the macro-structure of intelligibility in order for SysML and UML to be of maximum use without introducing artificial barriers to intelligibility of design.

The connection between logic and schemas via the philosophical categories is the normal projection route which we see in Aristotelian science. Post-Aristotelian science takes instead the long root via mathematics through model theory and representation theory. Aristotle thought that he could capture each category as a way of speaking about a thing (substance). He thought that the human living systems applied to everything in the universe, and thus he projected the human world on the physical world. His view held sway for a long time without anyone realizing that it’s anomalies were nature speaking to us of a different order than the human living order that Aristotle naturally projected and which was followed for so long without challenge. But eventually with Galileo and Newton and others through the use of math it was possible to see that physical nature that was not living and conscious and social had a different form that Aristotle assumed. So after talking about Logic we need to follow the route of science which is indirect projection rather than direct projection of the human sensibilities onto nature.

Logic

SysML and UML is not a formal method even though it is a formalization of design. Formal methods like Z, VDM and others erect a scaffolding around the system to be built so that it can be tested against that scaffolding. Formal methods have logic built into them, while with SysML and UML the logic remains that in the designers head. But what we need to realize is that normal traditional logic is not enough. We need to apply deviant logics in order to understand real world phenomena that are made up of contradictions and paradoxes. Anything that moves is a contradiction we have known since Zeno. All our systems that we deploy move in some respect. So all real systems embody paradoxes. Many elegant methods are used to get around these paradoxes as we can see in TIZZ [?]. But we can apply NS

Hellerstein's Diamond Logic[2] as a way to represent these paradoxes. There are other deviant logics that are useful like the Matrix Logic[3] of August Stern. These deviant logics need to be explored in order to discover a logic that will describe the interface with the real world rather than merely one that is applicable to the analysis of formal models. In my paper on Vajra Logics[4] I advocate a logic that uses all the aspects of Being, i.e. presence, real, identity, and not just truth as a superscript on a proposition. Much work along the lines of Situational Logic needs to be done and combined with our work on representation in order to be able to reason with our representations. At the moment we can reason within our own formalisms but we cannot extend those reasonings very well to our design products. This is an area where a lot of research needs to be done which has not even been begun.

Model Theory

Logic connects to Mathesis via Model Theory. Models represent all the statements that can be made about a mathematical category. Model Theory attempts to build a vision of semantics from a mathematical perspective. Model Theory needs to incorporate all the aspects of Being. When we add reality to the other aspects implicated in the construction of formalisms, i.e. identity, presence and truth, then from that mix meaning is generated. That means that meaning occurs when our formalisms come into contact with the world through testing. To the normal properties of clarity, consistency and completeness are added the properties of verifiability, validity, and coherence. Coherence is where the synergies that allow emergence to appear are packed and it appears in the process of integration. Systems Engineering is supposed to build a set of requirements, and a ConOps. The ConOps is used for validation purposes and the requirements set for verification. Then a functional or object oriented design is created and that is allocated to a physical architecture. Then it is that physical architecture that is implemented by software and hardware specialties. It is the implementation that the systems engineer takes and verifies and validates and checks the coherence of before it is released to the customer. So from this we can see that the Systems Engineer makes a formal model of the system first, then this is implemented, and then he does the reality check on the implementation. By the way his job in the mean time while implementation is going on is to be the steward of the interfaces to make sure the system remains coherent, not to vanish to another program during implementation. At any rate we can see that model theory sets the stage for the structure of the work of the systems engineer. The meaning that arises in the testing of the model of the system is in the surprises that arise from the defects found in the system. Surprise is information. Suppressing the defects produces negative information. Rather we want to find all the defects as early as possible. That is why it is surprising that Systems Engineers rarely do peer reviews of their documents as Software Engineers have learned to do.

What is even more surprising is that they resist building a model that can formally represent the system with the Gurevich Abstract State Machine method that could be reviewed like code is reviewed by software engineers. gASM models can be executed and thus there is an existence proof when they are complete and consistent. There is something like lines of code, i.e. rules that can be counted that Systems Engineers could use to measure their productivity. Thus there is a model missing in the Systems Engineering world that has not been recognized yet. That model exists as a bridge between requirements in words and the design from the various viewpoints that generate the minimal methods and give rise to the diagrammatic representations of UML and SysML.

[2] World Scientific 1997

[3] North Holland 1988

[4] "Vajra Logics and Mathematical Meta-models for Meta-systems Engineering" INCOSE 2002

I advocate the use of the gASM method by Systems Engineers to make semi-formal models of the systems to be build so that the properties of those models can be tested prior to implementation. It is an extra step, but it is really a more important step than doing UML/SysML models are at the next lower level of abstraction and we first need models at a higher level of abstraction, that in which the whole system is encapsulated in a hand full of rules that is expanded out in a stepwise refinement to an appropriate level of abstraction as Börger[5] suggests. By rights an gASM model should be created before any UML/SysML model is created because it is the gASM model that the UML/SysML model should be compared against. They are duals of each other. Note they are on opposite sides of our ontic/ontological triangle. We use reason to think about the properties of our simi-formal gASM model. Reason is different from logic in as much as it considers the reasons for things rather than merely their connections. Thus reason uses model theory as its tool. gASM uses the philosophical categores that give us causality and relates that to entailment in logic concerning the objects found in math. gASM is an abstraction of the Turing Machine at a high level of abstraction. So reasoning using gASM ties together several parts of our ontic/ontological triangle in an interesting way. Model theory is about all the possible ways that logic can talk about categories in mathematics. Reasoning takes these ways of talking about mathematical categories and connects causality and other philosophical categories with entailment in logic as constrained by the characteristics of the mathematical categories. Then when we apply this to gASM we find that we are reasoning about a model of the system which we can refine to any level of abstraction we like by stepwise refinement. It is at a larger scale than the diagrammatic methods that are harder to reason about because reason must deal with one diagram at a time. Here we can deal with the whole gASM model at once. But each rule contains a fusion of all the viewpoints that generate the diagrams of the minimal methods. So there is something higher than the UML/SysML level of representation that is closer to the logos and further from the physus that we should explore first before the UML/SysML modeling. This higher level model allows us to assign to each statement of the gASM model truth, reality, identity, and presence superscripts. We can construct models not just of the system but of its meta-system and of other systems within its meta-system. We can thus create the test environment that complements the system to do automated testing. What is more these gASM models can be built in Expert Systems Shells that failed for use as knowledge machines or one can use the specialized language AsmL for gASM[6]. So the technology for implementing gASM already exists. In fact that technology could be used to implement running simulations of gASM models. So our next critique of UML/SysML is that it is missing its dual, which is the gASM model that precedes it and that should be a bridge between the requirements and the design. The UML/SysML models come too soon, and only because the UML modeling system exists already from the needs of Software Engineers. The systems level is emergent over the software and hardware levels. It has its own organization and we see that in the duality between the gASM model and the UML/SysML models. We can see that our ontic/ontological triangle actually brings these two models together as me move from direct to mediated projection. If we had not developed the ontic/ontological triangle as a context we would never have realized that there is a dual to the UML/SysML model. That dual is not as important for Software and Hardware than it is for Systems Engineering. It is part of the emergent synergetic organization of the Systems Engineering level of abstraction that the gASM model becomes more important at that level and the connection between mathesis and logic becomes more important

[5] Egon Börger and Robert Stärk, Abstract State Machines: A Method for High-Level System Design and Analysis. Springer-Verlag, 2003.

[6] http://www.research.microsoft.com/foundations/asml/

through the connection with Model Theory. Model theory connects us to the Aspect of Being and shows us the importance of the Real in relation to the other aspects of Being related to formal models. If we bring into this milieu the Turing machine as a modeling tool and represent it abstractly then we can begin reasoning simi-formally about the system or meta-system using the Vajra Logic. It is model theory that represents all possible lines of reasoning about a category like sets using logic. When sets are manipulated by a Turing machine then we have a model of a system or meta-system. Model Theory forms the mathematical backdrop of our possible reasonings which then we fulfill by actually reasoning about the system or meta-system in terms of its properties like clarity, completeness, consistency but also like validity, verifiability, and coherence. These later properties arise from the interaction of the formal model with the real, which we see occurring when the formalism is manipulated by a Turing machine. And it is precisely this interaction that is the business of Systems Engineering. Each element is a phase of the systems engineering process. We take the requirements which is the logos and then we would build a gASM model of them, if we were smart. Then we would reason about the properties of the gASM model until we found that that model had all six properties that are the relations between the four aspects of Being. Then we would move from that formal model to the functional and physical models of design that exist at the level of the UML/SysML designs. At that level we break up the whole of the system represented by the gASM model into its constituent parts that must have the synergy necessary to produce emergent effects. Once the design has been created and compared back to the model of the whole in gASM, then it could be given to software and hardware engineers for product and component design and implementation. You see there is now no model of the whole to compare the fragmented UML/SysML diagrams to. The gASM method gives us this model of the whole to compare our object/functional designs to in order to determine if the design adds up to more, the same or less than the whole modeled with gASM. So this duality between the model of the whole and the model of the parts is what is missing and it is this we would never see if we did not look at the problem from the viewpoint of the ontic/ontological triangle.

Mathesis

But this lack of the dual model of the whole to the UML/SysML model is not the only problem. There is a deeper problem that must be addressed which is the problem that a whole part of our mathematical categorical base is missing. This is to say that we have a mathematics based on sets. All the other mathematical categories are derived from sets. But there is a dual to sets which are called masses that are totally missing in our mathematics and this dual category is very important for Systems Engineering. The mass category contains instances that together form the basis of emergent effects at the mass level. Masses are constrained by boundaries. There is a type of reasoning with masses called Pervasion logic that is the dual of syllogistic logic applied to sets. Sets are brackets that hold inherently different particulars with attributes that differ from each other. We use universals to reason about the relation of particulars to each other via syllogisms. Sets are about difference and the emergent properties are at the particular level, while sets are just containers with no properties. Masses are about identity and the emergent properties are at the mass level while instances are all alike within the mass. A list is a combination of set and mass properties. A list is a fundamental computational structure. The solution (as in a mixture of masses) is the opposite of the list where the emphasis is on the mass like properties. An array is a combination of solutions and lists. This is to say that an array brings into account dimensions which are like the different masses that a solution combines. But within each dimension there is lists of elements that are aligned with the elements of other dimension by indexing. So computational structures are mostly combinations of mass like elements and set like elements starting from the fundamental duality of sets and masses that are missing from normal mathematical categories. We need to add to the Mathesis of set orientation another similar mathesis of mass orientation and all the combinations between the two. This is important to Systems Engineering because we are really talking about emergence engineering and sets alone do not help us understand emergence. Emergence is modeled best with masses that have properties that contain instances. Sets do not have emergent properties and thus it is hard to find them in the mathematics based on sets. The point is we design with set like objects but when instantiation and execution occurs then we have masses of instances of the various objects we have designed interacting. We need a model of the interaction of instances in execution and the mass category gives us that. With that category in place we can transform back and forth between the emergent properties that appear in the executing mass of the system in operation, and the set like particular elements of design that engender those instances seen in operation. Systems Engineering deals both with the set-like design and the mass-like execution of the instantiation of the system in operation. We test the mass-like properties and we design the set-like elements that go into creating those mass-like properties. Systems Engineering at this point does not have this mathematical language to talk about its whole problem based on the mathematics that we currently have. The reason that Systems Engineering is an emergent discipline is that it is bringing out these flaws in our logic, model theory, mathematics that would not be seen otherwise because these disciplines have not traditionally interacted with the real world. Systems Engineers interact with the real world by fielding systems that have to operate in that world that do what the users want. Thus the Systems Engineer not only interfaces with the logos of the requirements but also the operational environment. He is also in charge of the integration of the pieces of the system and their internal coherence that results in synergies that produce emergent properties. All these ways if testing the implemented system are ways of connecting to the real by the formal models of the design. We should really talk about schematization instead of formalization, because we can "formalize" according to any of the schematic levels not just that of form. Thus there is patternization at the level of the pattern, systematization at the level of system, meta-systemization at the level of the meta-system etc. If we improve our mathematical categories by introducing the mass-like duals then we will give systems engineering a way of expressing emergent properties. And it is clear that there are kinds of logic associated with each of these approaches, i.e. syllogistic logic and pervasion logic. So there is not only a new way of reasoning about systems available but also we can use these mass like properties in our models, such as our gASM models as well. Thus our models can better express the holism of the system or the meta-system using constraint based logics rather than syllogistic logics. So in the improvement of our grasp of mathesis there is a trickle down effect to the rest of the ontic/ontological triangle where everything is modulated by the existence of the mass-like approaches in its associated logic.

Representation and Repetition

Finally we get back around the ontic/ontological triangle to the position that the UML/SysML representations occupy. But I think that our round about route has been helpful in showing what is missing in our understanding of these representations. But there is also something missing here as well. That is the difference between representation and repetition talked about by Deleuze based on the work of Lacan and harkening back to the use of the term by Freud. A repetition means that which does not repeat. This means that no amount of repetition at one level of emergence is going to produce the next level of whole at the next higher level of emergence. As we go down the dimensional hierarchy then we reduce by abstraction to create representations that are simpler than what is represented. But going up the hierarchy is more difficult because it is emergent and so that repetitions of representations will not achieve the emergence at the next higher level. This is a very important point that is not well understood by modelers using UML and SysML. Modelers think that their endless repetition of diagrams of minimal methods from different viewpoints will give a Pure Being view of the system. This is just not true. The very fragmentation of the diagrams of minimal methods themselves shows that this is not the case. If this was the case there could be one diagram that captures everything. But this does not work. Our human finitude prevents it. We can only see, absorb and understand so much at a time. So the fragmentation into the slices of the Turing machine by the viewpoints on the real-time system is essential. The system itself has an immanence that can never be made manifest. That is what makes its synergy non-computable and non-representable. All the different views and the bridges between them merely mark this essential immanence of Hyper Being. The designer is looking at this synergy as a point of pure immanence in the design space and attempts to bring it out into the design which can be implemented so that emergent properties can appear. The fact that we need multiple models of the design places us in a complexity space not just a complicated space of simple Turing machines from the point of view of Robert Rosen. So the designers of UML/SysML need to be sensitive to the fact that no number of diagrams will capture fully the design, and that as Naur says the design theory is not just non-representable in documentation but also non-computable. This means that the UML/SysML diagrams are allowing us to see aspects of the system through a glass darkly. That is why we need to copare the model of the parts of the system that the minimal method diagrams give us with the model of the whole in gASM. By moving back and forth between these two duals we have a chance of seeing the mass-like whole and the set-like parts in a single vision of the system. If we just have the UML/SysML models then we are like the blind men feeling the different parts of the elephant. So the lesson from Deleuze is a sobering one that no amount of repetition will take you up to the next level of emergence. Rather the next level is a whole greater than the sum of the parts and it is described by Godelian statements that are detergent if left out of the system and emergent if included with the system. Another point is that the design is a point within the design landscape. That design landscape is like a meta-system with the properties that Batille associates with a general economy rather than a restricted economy. Knowing that the design landscape of possibilities is a meta-system as well as the environment that the system will eventually inhabit, however the design landscape is as Stuart Kauffman says much more vast. The landscape of possibilities is much more vast than the landscape of actualities. So we need easy ways to navigate this design landscape. UML and SysML do not give that to us because the representation is not compact enough, nor is it robust enough. It is pointed at the design of the system rather than the lay of the land of the design meta-system. What we need is a model of the design landscape within which we select certain synergistic spots to exploit in our embodied systems. UML/SysML help us define point designs but not whole classes of designs that fill the meta-system of potential systems. UML/SysML need to be expanded to help the designer cope with the vast landscape of possible systems so as to pick good enough systems and to approach optimality as close as needed. The landscape of possible systems is so vast that good enough systems are hard to find. UML/SysML do not go beyond the representation of the point solution. And they only allow that representation on the level of form. We have to intuit the level of system by looking at the form because we do not have the complementary gASM model of the system to compare to. So even beyond the need to explore the whole design space we need to explore it at all the schematic levels. Thus we get representation as we move down the schematic levels and repetition as we move up the schematic levels but there is actually a leap to the next emergent level that cannot be made by repetition. There is between the repetition and representation within each schema at each dimensional level for that schema a mimicry between these two chains of connection upward and downward through the schematic hierarchy. These chains could be used to inform our models. Probably there are methods at each schematic level as well as diagrammatic conventions associated with them. We ought to be able to leverage the methods and the diagrams that apply at the other schematic levels in our design process if we understood them. At the moment they are only a hypothesis that needs further research.

Full Circle

So in the end we come back to the importance of the Schematic hierarchy and its development for Systems Engineering and for that matter Systems Theory which should be the foundation of Systems Engineering. But ultimately we should be able to talk of Schemas Engineering based on Schemas Theory. But that day is far off in the future. First we must fix all the problems we found when we ventured to go around the ontic/ontological triangle. If we work on those problems then we will create a more robust foundation for understanding the uniqueness of Systems Engineering minimal methods, and methodologies and tools that implement those minimal methods. Ad hoc additions to UML will not solve the essential problem. Systems Engineering has its own emergent organization different from software or hardware and we need to understand that organization. The ontic/ontological triangle helps us do that. It gives a context for the UML/SysML development. Those tools were developed by vendors that were exhausted from the methodology wars in Software Engineering. But by abstracting out the minimal methods and leaving behind the methodology that connects them we are left with a pure representation at the level of form that does not achieve the wholeness of a system. No matter how many different diagrams you add you will not achieve a global view of the system under design. Thus SysML is merely a further complexification of UML. That complexity is demanded by the non-representability and non-computability of the design space into which designers must forage for synergies that will allow emergent properties to appear in actualized implementations of designs. But if we take a broader view given to us by the development of general schemas theory and augmented by the understanding of the connection of schemas to mathesis and logic then we begin to understand the role of UML/SysML and can put them in their proper perspective.

What this paper calls for is a research agenda that looks into the unique requirements of Emergence Engineering based on General Schemas Theory and the necessary co-advances in logic, in model theory, in mathesis, and in representation/repetition theory, not to mention the application of philosophical category theory. It hopes to see this research agenda inform the development of SysML as a superset of UML such that it actually addresses the real problems of designers of Systems and Systems of Systems as well as Meta-systems, but that could also be applied to design on all the schematic levels. SysML is an advance for systems engineers but only a very small advance considering the ever more complex systems that we are expected to build by hand with no tools or even appropriate representations or methods. Software Engineering would like to explain to us that UML solves all our problems and with a few extensions of UML into SysML we will be satisfied. But there is a more fundamental problem that Systems Engineering is an emergent level above Software and Hardware Engineering with its own *sui generis* characteristics that cannot be reduced to the diagramming techniques of Software and Hardware. Rather a new way of representing whole systems needs to be developed that subsumes hardware and software representations. Part of that comes from the duality between gASM models and minimal method representations. Part of it comes from the incorporation of semi-formal methods with logic and math embedded. Part of it comes from the application of the full panoply of the schemas discovered by General Schemas Theory. But exactly what this new representation of the whole system/meta-system will be like is unknown and needs to be the subject of research. I am attempting to engage in that research into the foundations of Systems Engineering representations and methods. But it is a vast field and I invite others to join me in this adventure[7].

Author

A practicing Systems Engineer in a major Aerospace organization and also a student at the Systems Engineering and Evaluation Center (SEEC) at the University of South Australia. See http://holonomic.net and http://archonic.net

[7] See http://holonomic.net