GUREVICH ABSTRACT STATE MACHINES IN THEORY AND PRACTICE

HOW THEIR USE COULD TRANSFORM SYSTEMS ENGINEERING AND SOFTWARE ENGINEERING

Kent D. Palmer, Ph.D.

P.O. Box 1632
Orange CA 92856 USA
714-633-9508
palmer@exo.com

Paper for ASM 2000
COMPLETE ROUGH DRAFT

Keywords: Evolving Algebra, Gurevich Abstract State Machines, Formal Models, Systemic Models, Pattern Models

Why Gurevich Abstract State Machines are not a Formal Method

The Gurevich Abstract State Machine is not considered a formal method by many academics engaged in formal methods research to the chagrin of the proponents of this method. However, I believe that they are right and that those who support and do work on this method are fundamentally wrong about the nature of the method that they are developing. I would like to propose that the Gurevich Abstract State Machine is a ***systemic*** method instead of a formal method. Before we can understand the place of this method within our arsenal of means to help produce systems or software it is necessary to understand clearly what this method actually is and does.

The Gurevich Abstract State Machine came into existence as something called Evolving Algebras. Gurevich attempted to generalize the Turing Machine so that it was not so difficult to encode problems into a Turing Machine format. This method has proved valuable as a means of formalizing various languages. It is beginning to be recognized as an important method for systems engineering and software engineering industrial practice. However, a major barrier to adoption is the fact that its proponents consider it a formal method and because of that it is categorized together with other formal methods which are essentially useless for industrial development except in some very narrow areas such as safety and security. In order to break this log jam it is necessary to dispel the myth that the Gurevich Abstract State Machine Method is formal. Instead we need to recognize it as a ***systemic*** method. But in order to comprehend that it is necessary to get a theoretical perspective on the relation of ***forms*** to ***systems***.

Form and system are two separate templates of understanding[1]. We are familiar with formalisms from logic, geometry and algebra. Essentially formal methods in software and systems engineering use the same logico-mathematical techniques to prove properties about a given model. We can understand this approach better if we consider the meta-levels of every formalism. All specific forms appear at the meta-level zero. Forms as an abstraction occurs at

1 "Meta-systems engineering : A new Approach to Systems Engineering Based on Emergent Meta-Systems and Special Systems Theory" by Kent D. Palmer, Ph.D.

meta-level one. We consider all the mathematical categories such as set, group, lattice, topos (includes logics of various kinds), category as examples of formalisms. When we ask what the meta^2-level of a form is, i.e. a form of a form, then we find that it is the theorems that are proved about the elements of the formalism. When we ask what the meta^3-level of the formalism is then we find that *the form of the form of the form* are the axioms that form the basis of the formalism. There is even a meta-level four which are the anomalies which we cannot tell whether they are inside or outside the formalism like the Godelian statements.

These are the basic meta-level structures of a formalism. Formalisms are best expressed using mathematical category theory as a basis for understanding. Mathematical Category theory establishes how the various elements inter-transform. Those inter-transformations are represented as arrows. Category theory forgets the elements and just treats the arrows. Category theory also establishes the arrows between various categories which are called functors. Something proved in one category may suffice for another category because the structural isomorphisms between categories are understood based on the functors. Category theory uses the same language of mapping arrows (that preserve the associative property in spite of the relaxation of the commutative property in many cases) in order to define the category of CATEGORY and a special category called a Topos that defines various logics. This gives a very powerful means of understanding the basis of formalisms and their relation to logic. Formal Methods normally define models that are open to logico-mathematical interpretation. This logico-mathematical interpretation allows us to reason about the formalized elements and prove properties as theorems based on the posited axioms that underlie the formal model. Examples of such methods are Z, VDM, and Larch. There are myriad such methods developed by academics that have found little use in industry due to the cost of implementing them. Many of the things that are done in systems engineering and software engineering defy rigorous definition required for a formal method.

We can think of a formal method as a scaffolding around the system or software that we intend to build. It defines pre and post conditions on the functions of the software. It normally also defines the data types of the processed data. The most powerful idea of formalism is the idea of uninterpreted function types and data types. This idea allows one to compute results about the relation between functions and data without knowing the exact values. One might get an expression rather than a specific result. The idea of uninterpreted function types and data types allows the formalism to operate at higher levels of abstraction rather than only at the lowest level of abstraction where the programming language defines the algorithm and the data. It allows us to analyze the formal relations within the system and software without knowing everything about it. Other formalisms concentrate on defining distributed or parallel execution such as Hoare's CSP. Normally formalisms specialize in one kind of representational domain. Sometimes various formalisms would have to be used to capture all the aspects of a system or software component.

But Form is not the only template of understanding. Two other templates of understanding at higher and lower levels of abstraction are the system and the pattern. A formal structural system combines all three of these templates of understanding into a single approach toward phenomena that that has been used very successfully by Science. George Klir in his book Architecture of Systems Problem Solving attempts to abstract the formal structural system and give an abstract representation of this very powerful method that combines different levels of comprehension together. The formal level allows us to make proofs. This is the strongest kind of comprehension we can have. But if we cannot prove things then we attempt to explain them with structural explanations at the level of the pattern template of understanding. If we cannot explain then we fall back further to merely describing the phenomena systematically. The systems template of understanding gives us a framework for understanding the gestalts we see out in the world and how they fit together. The pattern level of understanding instead helps us understand the ordering relations between contents of forms. Thus we can see that there is a nesting of the various templates of understanding. The countable forms contain qualitative contents which we categorize and describe as transformations at the structural level. The ordered pattern of content is transformed into another ordered set of content so that we can explain the transformations of content even when we loose the ability to refer to the forms that contain those contents. The forms appear as objects within a system which contains many different kinds of forms perhaps from different categories and describes their relations to each other. Description at the system level is the weakest way of comprehending things which is augmented by explanation at the pattern level that understands transformations between categorized contents which is again augmented by a formalism at the level of form which will sometimes allow proofs in some narrow range of well formed phenonomena that complies to the properties of our algebras or geometries or logics.

Both the system and pattern levels have their own special methods that apply at their level of comprehension. For pattern we consider the work of Grenander who has developed a mathematics of pattern. For system there are also many examples of structuralism in individual sciences. For instance, Jacque Monod develops a structuralism to understand evolutionary theory in Chance and Necessity. Most structuralisms are specific to disciplines like the structuralism of atomic theory that underlies chemical reactions. Underlying atomic theory is the structuralism of fundamental particle theory. Underlying fundamental particle theory is the structuralism of quark theory. Quark theory may be underlain by the structualism of string theory in the future. Structuralism is the fundamental method of science for understanding the world. This is because formalisms have a fundamental weakness when it comes to handling discontinuities in forms, space, or time or other variables. In formalisms there is only one kind of formal element. But in systems there are various kinds of elements. Formal elements are assumed to have homogeneous contents. But in actuality contents come in various kinds and map across transformations in various ways which cannot be captured in any formalism. So the strength of the formalism in terms of proof is balanced by their weakness in terms of describing the interactions between different kinds of contents or forms form various categories. In both cases there are transformations in order. At the level of content we see global patterns of ordering of contents that are not easily captured by the outlines of forms. In categories we see various kinds of orders through the various combinations of ordering properties in sets, lattices, groups, rings etc. Pattern and System need to augment the formalisms in order to understand the discontinuities that exist in the world as we discover it.

The fundamental differences between pattern, form and system are underlined as we take each of these to their various logical meta-levels. The meta-levels of pattern are categories, spectra and singularities. The meta-levels of system are rules, properties and exceptions. See how different the second meta-level is in each case. We can compare categories, theorems, and rules and see how different they are. We can compare spectra, axioms, and properties of elements and we see a similar fundamental difference at the third meta-levels. Similarly at the fourth meta-level we have singularities, anomalies and exceptions. The fourth meta-level raises the temperature of our search if we apply a method like simulated annealing. The more oddities and peculiarities we have the wider our search in the landscape of all possible formal structural systems. The formal structural system has its basis in spectra, axioms and properties of elements. It uses these to articulate categories, theorems and rules. The formal structural system itself is build up from these elements which are further articulated in relation to each other to describe the various different kinds of contents, forms and kinds of categories in a system.

Where we can point to Grenander's unique mathematics of pattern it is difficult to find something at the system level to point at as a proper systemic method. It is my belief that the Gurevich Abstract State Machine plays this role. This is because it is essentially a method that defines things in terms of rules. The properties of things and their relations define system states that are then dynamically modeled using rules to give the diachronic actions of the entire system. Thus where Grenander's method should be called a ***patternism*** or a pattern method, and Category theory and other formalisms like Z, VDM and Larch should be called a Formal Method, then Gurevich's ASM should be called a ***systemism*** because it is a system specific method. If this is true then this method is mis-classified when we call it a formalism and it is no wonder that other formal methods researchers reject it's claim to be a formal method. It is a systemic method and should not be classified with other formal methods at all. This was intentional when Gurevich produced the method because his intent was to abstract the structure of the Turing Machine. The Turing machine is not something formal. It is a model of any computational device. A computational device is a system of diverse elements that work together to achieve the result of computation. Computational devices produce a result across time. Formalisms are generally outside of time and are static representations. Thus formalisms rightly are synchronic slices of what the Turing machine computes. But the Turing machine itself is at a higher level of abstraction which is essentially systemic because various kinds of forms must cooperate to give the result of computation.

Rules are the meta-level two of a system. They are different from theorems. In fact rules give us the basis of proof which ties the statements and lemmas of theorems together into a logical argument. An argument can be thought of as an "***and***ing" of various propositions on the left-hand-side which implies the conclusion on the right-hand-side of the conditional structure. A Theorem is an argument in the form of a proof concerning the relations of the elements of a formal system. Once proven theorems are always true. But rules may enter and leave the system as the system changes phases. Rules are not proven but instead posited which is in keeping with the descriptive nature of the system. The categories of pattern contents and their generators are again of a different order. In order to understand various phenomena and produce design theories all of these levels of understanding are necessary. But each has a specific and different role to perform. There does not seem to be other good examples of the kind of abstraction at the level of system that the Gurevich ASM method provides. It goes beyond what any Formalism can provide because it encompasses the workings of the whole system as a dynamic entity rather than a series of static slices. Yet it is weaker than a formalism because it is essentially descriptive, or its inverse which posits a specification.

The system is a high level abstraction which may be understood more concretely as a social gestalt. The gestalt has a dual which is a flow. A system like the Heisenberg uncertainty principle combines both gestalt and flow as irreconcilable but complementary ways of viewing phenomena. A gestalt is seen as in psychology as a tension between figure and ground. When the figure is submerged and becomes a reference point and the background is raised in our attention flowing across the reference point then the gestalt is transformed into a flow. It is interesting that this duality between gestalt and flow is not recognized in the literature. A system is a series of gestalts synchronically embedded in a flow. We see these gestalts as we look from element to element in order to discern their relations within the system. Many times systems are reified into objects which contain statically other objects. We render these systems dynamic when we think of the changing relations between elements over time as seen in an idealized movie of the interaction of the elements in a system. But in this we forget the idealized observer of the system. When we consider the system as gestalt and flow then we know that the system is always understood only in relation to an observer, in a way similar to that discussed by Guy Jumarie in Subjectivity, Information, Systems. When we retreat from the abstraction of the system then we discover two phenomena that exemplify the system better than most and which supply us with the paradigmatic cases of a system. These are the game and language. Wittgenstein has recognized the importance of these two examples by coining the term "language game" to describe the systemic context within which sentences as elements must be understood within an argument. The second meta-level of language is its grammar while the second meta-level of a game is its rules. We generally speak of the rules of grammar because we understand that grammar gives proscriptions for proper or well formed sentences. It is precisely this level of grammatical rules of a computational which the Gurevich Abstract State Machine specifies. Thus, if we do not understand what a system is because of its abstractness then we can always appeal to either language or games as the basis for our understanding of the nature of the definition of a system that the GASM supplies to us. If we move up to meta-level three we find in games the definition of the properties of the pieces. If we move up to meta-level three we find in language the definition of the phonemes and letters that are the basis for expression. At meta-level four in games there are the exceptions to the rules such as the castling move in chess or the violations of verbal conjugation patterns in language. Pinker[2] has recently argued that rules and exceptions in grammar activate different parts of the brain and thus both are necessary for language. Meta-level three of the language game is always the allowed exceptions to the rules or properties defined at meta-level two. What is fascinating here is that this systemic meta-level three nests into the template of understanding of form because it is the specification of the properties of the thing that allow us to isolate the figure of the form within the gestalt. Thus, there is a very close relation between the template of the system and the template of the form. Similar kinds of nesting exist for the other templates of understanding as well as explained in the paper "Meta-Systems Engineering" by the author.

Each of the meta-levels of the templates of understanding may be generalized across all the templates of understanding in order to yield a picture of the meta-levels of Being which divide into four kinds: Pure, Process, Hyper and Wild. These kinds of Being make up the various meta-levels of our world. In each case the meta-level zero is the being in question whether it be a concrete system, form or pattern. Meta-level one, Pure Being, is always the frozen abstraction of the class of beings described by a particular template.

[2] Steven Pinker, Words and Rules : The Ingredients of Language (Perseus Books, 1999)

Meta-level one corresponds to what Heidegger calls the present-at-hand in Being and Time. Meta-level two is always something that allows for the dynamism of that abstraction. So for instance in forms the dynamism is the proofs that produce theorems from axioms. In systems this dynamism is given by rules which describe how a system will react in the next synchronic slice of a diachronic progression. These rules may be changed by meta-rules in order to create discontinuous action from moment to moment as the ruleset varies. For a pattern it is the categorization process that produces change as we see patterns based on the imposition of various categorical templates. The Process Being level corresponds to what Heidegger calls the ready-to-hand in Being and Time. Meta-level three corresponds to what Heidegger calls ~~Being~~ (crossed out) and what Derrida calls differ**a**nce, i.e. differing and deferring. At that level ambiguity and indecision are introduced. It is interesting that Hyper Being appears at the system level as the definition of the properties of the entities that are allowed into the system being defined by the rules. Gurevich ASM does not have a language for defining the properties of such entities. In terms of the Turing machine we find this language can be anything that is expressible in codes and can appear on the tape. In terms of games it is always the definition of the pieces and players of the game. In terms of language it is the basic phonetic or letter forms that may be used as a basis for speech or writing. This is in contrast to the axioms of the formalism and to the spectra of the pattern level that the categorization has broken up and defined in terms of regularized intervals. Axioms may have undecideable aspects like the parallel postulate that generates various geometries based on our decision as to whether parallel lines cross or not. This indecision point of ambiguity leads to different kinds of formalisms for geometry. Similarly, indecisiveness at the level of the system may admit variations in the kinds of players and pieces and boards that may be used to play the game. We recognize this when ever we see a strange chess board with outlandish pieces. In checkers there is the transformation of a piece into a double piece at the end of the board which can move in any direction which makes the endgame so different from the foregame. This difference in pieces marks a difference in the play of the game in a striking manner. With respect to the spectra of the patterns there are ambiguities between various category systems such as those found in systematics and ecology. Various animals are classified differently based on different category schemes. The same spectra may be broken up in various ways leading to ambiguities in classifications that can have surprising results. All these various ambiguities lead us to have an appreciation of the strangeness of Wild Being as it appears in the highest meta-level of the various templates of understanding. This final meta-level is the fourth which encompasses exceptions, anomalies, and singularities. These are all things that directly violate the property rules, axiom theorems, and spectral categories in various ways that are unexpected and strange. These incursions of Wild Being always lead us to raise the temperature of our search in simulated annealing for more optimal rules, axioms, or categories. Notice that rules and categories exist at meta-level two while axioms exist at meta-level three. Thus the operative level by which we judge what is fundamental in each template of understanding is not the same. Properties of elements of systems, axioms, and spectra share the same level while rules, theorems, and categories function at the same level. But we need to note how different these various expressions of the meta-levels of the templates of understanding are. Rules and Properties of things within a system are very different from the theorems and axioms of formalisms or the categories and spectra of *paternisms*. This fundamental difference is what we have to keep in mind when considering weather GASM is a formalism or not. GASM is not a formalism but the only known *systemism*, it defines the rules of computational systems. It has both the weaknesses and strengths of systemic models. We cannot prove anything with it but only show existentially a working model of the computational system we are describing. It is strong in the sense that it covers a much wider field of phenomena than the formalism, but it is narrower in the sense that we cannot do proofs in it as one might do in a formalism. However it is precisely this which is needed in software engineering and systems engineering for the construction of complex computational systems.

How Gurevich Abstract State Machines relate to Requirements and Architectural Design Methods

If we consider the GASM method to be a *systemism* then it is uniquely qualified to stand between requirements and design in both software engineering and systems engineering industrial practice. Requirements ideally are single linguistic statements of distinction and desire on the part of the customer. Each requirement should be stated as a single aphoristic statement as a sentence standing alone using a shall. Even when we attach notes to these statements to provide context there is still a wide variation of interpretation that is possible given any set of requirements. The Gurevich ASM can serve a good purpose as the interpretation of the set of requirements as a whole. We start with a single rule that describes at a very high level of abstraction the entire system that will satisfy the functional requirements stated in the requirements for the system. Sometimes it is noted that there are three kinds of requirements for a system, functional, non-functional which are qualitative or performance related and constraints. This reminds us of the meta-levels of the system were functional requirements corresponds to rules, non-functional requirements corresponds to the properties of elements or the system as a whole, and the constraints corresponds the exceptions in as much as constraints make it possible to identify those things that violate constraints. Thus, the three major kinds of requirements for a system remind us of the meta-levels of the system itself. It is no surprise then that the rules make possible for us to specify the functionality of the system in a causal context of ***if … then …*** statements. In fact, it can be easily seen that rules in this form encompass all the fundamental viewpoints on the real time system, i.e. agent, function, event and data. Surprisingly rules also have the meta-levels of information within in them which takes data or event and lifts it up to the level of information, then knowledge and wisdom as the combination of experience and knowledge. As we articulate the rules of the GASM to describe the causal nature of the system being described we first expand them to a handful of rules that describe the emergent properties of the system. Then we consider the monitored and controlled variables and describe bridges between them using rules. Finally we begin to express the internal modes and states of the system as a whole until we produce a ground model that can be discussed with the customer in order to hammer out a mutually agreed upon interpretation. Once the ground model has been created and agreed upon then we can use the GASM as a framework to capture our knowledge of the system being developed down to any level of articulation we wish. In producing the GASM causal model of the system we speculate as to the kinds of design elements that might be necessary to create the system and we might include some of these hypothetical design structures as a vehicle for writing rules but these hypothetical design structures do not bind us to any particular design and are considered only temporary postulates which will allow expression of the causal structure of the application under development. That structure will have to be fully designed once the interpretation of the requirement are fully specified.

We note that in order to produce a picture of the properties of the elements that are allowed into the system it is necessary to pursue an object oriented exercise. In this exercise we might wish to have rules relegated to the objects as methods. Thus the top down definition of the emergent properties of the system by rules is complemented with a bottom up definition of the attributes and rules that specify an element within the system. This definition of the objects and their properties and perhaps their dynamic causal behavior allows us to model the third meta-level of the system under construction. Thus we capture the qualitative and performance non-functional requirements of the system in terms of the interaction of its elements rather that in terms of the emergent properties of the system as a whole. Beyond that there is the modeling of the constraints within which the system is bound and the description of exceptions. These may be captured by special rules that are tied to the exceptional cases or that attempt to apply to all the objects within the system under particular circumstances. In this way we see that through using rules in various ways it is possible to describe all three meta-levels of the system. Rules are extremely versatile and this is in fact their unique linguistic capability that may be put to use in describing systems in general or games or language at a more specific level. Phonemes or letters or pieces or board can all be considered as objects with attribute ranges. When we gather together the attributes together into a form and allow that form to have behavior we convert the set of attributes into an object. At the third meta-level of the system we have only a nexus of attributes (what Klir calls the source system) which we transform into variables with specific relations to each other at the level of pattern and group into formal objects. This nexus of attributes can be described as a set of rules operating on spectra given a specific category scheme. In this way we see how closely the templates of understanding of form, pattern and system really are at the higher meta-levels. All the various meta-level models work together to give us a comprehensive basis for understanding that we call the formal structural system. This combination of the templates of understanding gives us a great deal of leverage in comprehending the various aspects of our world in the pursuit we call science.

The author has created a comprehensive picture of the field of possible design methods called the Integral Software Engineering Methodology. That method posits that there are four fundamental viewpoints on every real-time system and that there are a set of minimal methods that bridge between these viewpoints in the design process. What we notice is that these various design methods proliferate the kinds of representations that are needed to perform a comprehensive design and to describe the result. These might include the following minimal design methods: Petri Nets, State Machines, DARTS of Gomma, Use Cases, Virtual Layered Machines, Worldlines and Scenaios of Agha, Dataflows of Yourden and DeMarco and Objects of Simula and Smalltalk fame, Entity Relationship Attribute diagrams of Chen, Interval Logic of Allen, as well as the relations between Data and Event that give us the structure of a Turing Machine. We note that the Gurevich ASM gives us a causal model of Turing computationally in a set of conditionals that unify the viewpoints on design. When we fragment those viewpoints in order to zoom in on the design that is implicit in our ground model then the Turing machine is relegated to merely the relations between elements in spacetime while all the causality is spread out in the interaction of the various elements of the minimal methods. At the level of Architectural Design the design elements implicit and hypothetical in the GASM become explicit and are posited directly in terms of the interaction of the minimal methods.

The relation between the requirements, GASM and architecture is very interesting. We can see that requirements are wholly linguistic sentences that define together the emergent properties of the system under consideration. They use the full expressivity of language in their definition. But this leads to the problem of interpretation of these sentences. GASM allows us to specify a particular interpretation which emphasizes causality and functionality but also expresses agency, event and data. This interpretation can function at all the meta-levels of the system defining functionality, non-functional quality and performance goals, as well as exceptions and constraints. The linguistic requirements can do this as well but lack the uniformity of syntax that the GASM produces. However, the GASM cannot express the various kinds of relations needed to define design because each rule fuses agent, function, event and data perspectives. It allows us to express the entire system as one rule, or a hand full of rules that describe emergent properties, or as bridges between monitors and control variables, or as descriptions of reactions in terms of modes and states. These global views of the system are complemented by descriptions of attribute nexes and their associated rules that can be seen as objects with form and behavior. When we apply global rules to these objects or single some out as exceptions we describe the fourth meta-level of the system. Yet this view given us by GASM is monolithic and can only be broken apart by the separation of the viewpoints which allow the emergence of the minimal methods as alternative forms of relations within the system that address various characteristics of real time systems that cannot be handled by the GASM directly. Boerger says that the GASM may be taken all the way down to the level of code. But this is not a good practice because at various levels distortions and warpages are introduced which should be handled by architectural design and programming languages. One of those warpages is that if the architecture is not made explicit in an architectural design then it cannot be optimized for the application. Minimal methods exist at the level of form rather than system. Each of these minimal methods can be analyzed to show that they are composed of the four fundamental kinds of pattern, i.e. value, sign, structure and process. Value and Sign are continuous and structure and process are discontinuous. Structure embodies discontinuity in space (data) while process embodies discontinuity in time (events). Values are seen in accumulator registers while signs are seen in pointer registers in the CPU. They are embodiments of what Heidegger calls the present-at-hand and ready-to-hand respectively and what Merleau-Ponty calls pointing and grasping. The combination of CPU cycles and memory locations give us the plenum in spacetime of discontinuities that these mechanisms use to produce illusory continuity. For a deeper analysis of this see the authors paper on "Software Ontology"[3]. The relation of the four viewpoints on real-time systems and the minimal methods to systems theory can be found in the author's paper on Software Engineering Design Methods and General Systems Theory[4]. The major point of this section is that when GASM is used to mediate between requirements and design, at either the software engineering level or the systems engineering level then we can see a progression of reification in which the fragmented aphorisms of the requirements are converted to the single conditional syntax and unified in a construct that combines the four viewpoints with the various meta-levels of data, information, knowledge and wisdom. But this description at the systemic level

[3] See second chapter of author’s Wild Software Meta-systems at http://server.snni.com:80/~palmer/apeiron.htm

[4] See first chapter of author’s Wild Software Meta-systems at http://server.snni.com:80/~palmer/apeiron.htm

needs further elaboration at the level of form where we describe the interaction of the various minimal methods from the various separated viewpoints. Once an architecture has been elaborated then we can apply the normal formal methods like Z, VDM and Larch to erect a scaffolding around the architecture and prove various properties about it. But we can also prove some properties at the systemic level once we have a representation in terms of rules. Being has four aspects: identity, presence, reality, and truth. The relations between these aspects of Being define the six fundamental properties of every system which are verification, validation, coherence, consistency, well-formedness (clarity) and completeness. These properties of the system may be reasoned about at the level of the GASM directly. Formalisms themselves exist as systems when they are used to produce a set of statements. Thus formalisms to the extent they are considered systems of statements may express these properties. Other rules may be added to the GASM to specifically embody other properties than these fundamental ones. The point is that every formalism needs for the architecture to be specified in order to operate on it in terms of logic. GASM alone can describe the emergent properties of the system as a whole with little or no articulation of an implicit architecture. Thus, when we move down from the level of the system to that of form when we create the architectural design that is when the other formal methods are most useful.

GASM allows the Software Engineer or Systems Engineer to describe directly the essence of the system to be created in terms of its inherent causality, in a way that brings out its functional, non-functional, and constraint requirements and interprets them concretely. Thus, GASM has a unique role to play which it cannot perform as long as it is considered just another formal method, as useless as all the others for the industrial designer. But instead we must consider the uniqueness of the GASM method as deriving as a model of a Turing computational system, which is fundamentally descriptive rather than honed to provide proofs. Designers need such a dynamic descriptive technique to mediate between the scattered field of requirements and the myriad elements that work together to produce a design. GASM allows a moment of unity between these different kinds of fragmentation and forms a bridge between the meta-system or general economy of aphoristic requirements statements and the formal composition of the design by describing the restricted economy of the application being specified as the dynamic rules that govern the causal behavior of the system.

Applying GASM in an industrial setting

GASM has been applied to a large scale software development project of a real time nature by the author. In that exercise many interesting lessons were learned that should guide our application of GASM to other projects of an industrial scale. Some lessons from this experience applying GASM will be mentioned in this section.

The good thing about GASM is that you can describe how to apply it in one sentence: Describe the system in rules. This makes for rapid comprehension of the method. The idea that GASM is not embodied in a particular formal language needs to be emphasized always because it is necessary that the person learning the method creates their own linguistic embodiment of the method in order to assure their understanding of it. Formal languages that embody GASM should be de-emphasized as that makes it seem like all other formal methods that emphasize learning specialized logical languages.

The GASM interpretation may be considered an appendix to the SRS at the

Software Engineering Level or the System Specification when used at the Systems Engineering Level. This appendix serves to specify the interpretation of the natural language requirements in the document as a prelude to either system design or software architectural design.

The GASM may be developed in parallel with a dataflow model in which case the two should be reconciled before entering detailed design. The dataflow model should remain essential or logical and should not embody design decisions.

GASM is a natural bridge between Systems Engineering and Software Engineering. The best case is where there is a GASM model at the System Level and also at the CSCI level for each CSCI. These models should be connected to each other through an IRS or ICD at the system level. The IRS/ICD is the best basis for organizing the GASM models at each level. They specify the inputs and outputs at the Software and System levels respectively. Every input should be associated with a rule which chains with other rules until all the outputs are produced. Through double bookkeeping of outputs it is possible to preserve the output sections of these documents and show which rule they emerge from in the nesting of rules under the input sections. Internal rules need to be placed in another section of the model. By using the IRS/ICD as the framework for modeling there is a natural place for the GASM model to fit into the documentation. SRS and IRS are produced together. The GASM model should be an appendix to the SRS but based on the IRS so that it actually mediates between these two traditional documents. The same may be said for the System Segment Spec and the Interface Connection Document.

An inverse GASM model may be produced to embody the test cases for the system. This is probably the best use of the GASM model because it specifies through this inversion the test conditions at a very early stage in the development. This inverse model may provide a view of the meta-system of the system being constructed.

We may produce an architectural model of the system early by distributing the rules of the GASM model among agents in a hierarchy of processors and tasks that make up the system under construction. It may mean that certain of these agents pass on the messages that are finally processed by another agent. Distribution of the GASM model with an agent hierarchy gives a sketch of the requirements for the architecture that needs to be defined.

The most difficult thing about using the GASM method is beginning to think causally in terms of rules. It is best to first attempt to describe the whole system in terms of one rule, or a very few rules that capture it as something unified at the highest possible level of abstraction. Then from this it is best to advance to a set of rules that embody the bridges between the monitored and controlled variables of the system. After that one might embody the modes of the system which might occur in degraded or other special system configuration circumstances. Finally the highest level states of the system might be embodied. These highest level states can be considered the ground model which can be discussed with the customer in order to align expectations. The ground model does not embody any internal structures produced by derived requirements. As we continue refine the GASM model we introduce derived requirements and speculative design structures increasing the resolution of our model of the system under construction. We may continue this refinement process indefinitely but practically it should at least be continued until detailed design. Thus, we keep this model going all through Requirements Engineering and Architectural Design phases of new development. We consider the GASM model to be an interface between the SRS and IRS on the one hand and between Software Requirements Engineering and Software Architectural Design on the other hand. Or if we are using it at the System level then it is an interface between the System Requirements Document (SRD) and the Interface Connection Document (ICD) and between Systems Requirements Engineering and Systems Architectural Design.

GASM as a method is the same whether it is applied at the level of the entire system which includes both hardware and software or at the level of the software system as an independent whole. In both cases it serves as a bridge between requirements and architectural design. This bridge does not exist in current practice and thus it is difficult to reconcile the requirements and the design in many cases. Requirements are normally open to many interpretations by the various stakeholders. The GASM model nails down a particular interpretation early and allows it to be agreed upon by all the parties involved because it makes the specified interpretation visible as an appendix to the requirements document. It serves to connect all the functionality causally and thus produce a constrain on the space of possible designs which helps the Architect be sure that all the functionality and its causal linkages are embodied by the software design.

Class room exercises show that students can quickly pick up this method and it is easy to relate to requirements documents. In general doing a GASM model uncovers mistakes in the requirements document that need to be repaired as the requirements development proceeds. GASM gives a basis for the mental simulation of the system that answers to the requirements proposed by the designer and the requirements analyst either as a team or as one person doing both jobs. GASM is an excellent language for communication between these two job functions if they are being done by two people instead of one.

GASM gives us something to count. We can keep track of introduced, changed and deleted rules as the GASM model evolves. GASM because it executes gives us a criteria for when we are done with the requirements process. We are done when a complete ruleset is built which executes at the appropriate level of abstraction. It gives us intermediate mile stones because it has various levels of refinement. Its execution may be automated either by current Rule Based expert systems technology or by building programs using *If…Then…* statements. Thus the GASM model may be turned into a simulation of the whole system which may be tested within a simulation test-bed to verify the behavior of the emergent system prior to its design and implementation. It is because GASM models can capture the emergent behavior of a proposed system without recourse to the specification of the parts of that system beyond some speculative sketches that GASM is an excellent model at the ***systemic*** level rather than at the lower formal level necessary.

The GASM model connects not only requirements to design but also connects design to integration and requirements to test. The inverse GASM model provides a definition of the test environment that is necessary. The inverse GASM model at the CSCI provides the test environment for the CSCI and the GASM model at the System level provides the test environment for the System level tests. Also the integration of these various GASM models indicates the difficulties that will be encountered in the integration activities.

The maxim “build it twice” which is seen by many developers as the only way to get the optimal design of a system is followed when we add GASM models to our development process. The first build is the GASM model which allows us to work out the causal structures of the software under construction. Then we build the software itself based on what we learn from the executable and mental simulations of the GASM model and anti-models.

Conclusion

GASM is not a formalism but is instead a ***systemism***. This means that it functions not on the level of the formal template of understanding but on the level of a systemic template of understanding. These two templates have completely different properties and thus need to be distinguished. These differences become apparent if we ask ourselves what are the meta-levels of the *form* and the *system* respectively.

GASM is an excellent bridge between requirements and architectural design at both the Software Engineering level of abstraction and the Systems Engineering level of abstraction. When we use this bridging method on our projects we find that it confers many benefits and makes our software and systems development processes more efficacious, i.e. more efficient and effective.

GASM deserves to be piloted in industry so that these beneficial qualities of the method may be experienced by other practitioners. At this point there is a gap between requirements and architectural design which needs to be bridged in order to produce a smooth transition between these necessary phases of development. The application to these upstream phases has direct implications for both test and integration. The inverse GASM model produces an early view of the necessary test environment and the integration of software and hardware level GASM models into a system level GASM model gives our first indications of the difficulties that will be encountered in integration.

The promise of the GASM method is the transformation of both Software and Systems Engineering practice due to the fact that this method unifies what are now separate phases with separate models and methods into a synthetic and coherent overall structure of methods and processes. We notice that the GASM model at the software level fills the gap between the SRS and IRS, between requirements engineering and architectural design. We notice that it also serves as a bridge from architectural design to integration and from requirements engineering and test. We notice that we could ideally take our refinement down to the implementation level. In other words the ruleset serves as a glue to connect all the different non-routine parts of software development, i.e. requirements engineering, architectural design, integration, test together and then to ideally connect these with the routine software development processes as well, i.e. detailed design, implementation and module test. This glue like or bridging function we see in the GASM method with respect to the other processes and methods of software development comes from the fact that it is in fact a systemic method instead of a formal method. The GASM embodies the emergent properties of the software system to be constructed and serves to create a process and method level infrastructure between the now various and disparate methods and processes we find in current software development practices. GASM not only embodies the system to be constructed but also turns the development process into an integrated, coherent and synergistic system as well. By introducing GASM into the software development process we transform the independent and autonomous development processes with their associated methods into a whole which are now independent and separate. In other words, now these various phases have only an implicit coordination that we see in the resulting end product. By introducing GASM this coordination becomes explicit and is embodied in a specific model which has many unifying and bridging uses within the development process itself. GASM modeling represents explicitly the system level glue that holds together the fragmented requirements and the articulated design at the representational level and holds together the various software development processes at the method and process level. GASM represents explicitly the now hidden synergetic effects of the emergent properties of the system in a way that can be seen and manipulated directly by the developers. The same thing could be said at the Systems Engineering level of abstraction.

GASM's transformative potential comes from this rendering explicit of the emergent systemic characteristics at the representational, methodical and process levels within the development process at both the software engineering and the systems engineering levels.

Acknowledgements

I would like to thank Egon Boerger for teaching me the Gurevich Abstract State Machine Method. I would like to thank Philipp Kutter for soliciting this paper for ASM 2000. I would like to thank my students at the University California, Irvine Extension for their work learning and applying the method from which I have learned so much.

About the Author

Kent Palmer is a Senior Systems Engineer at a major Aerospace Systems Company. He has a Ph.D. in Sociology concentrating on Philosophy of Science from the London School of Economics and a B.Sc. in Sociology from the University of Kansas. His dissertation was on The Structure of Theoretical Systems in Relation to Emergence[5] and concerned how new things come into existence within the Western Philosophical and Scientific worldview. He has written extensively on the roots of the Western Worldview in his electronic book The Fragmentation of Being and the Path Beyond the Void[6]. He has at least seventeen years experience[7] in Software Engineering and Systems Engineering disciplines at major aerospace companies based in Orange County CA. He served several years as the chairman of a Software Engineering Process Group and now is engaged in Systems Engineering Process improvement based on EIA 731 and CMMI. He has presented a tutorial on "Advanced Process Architectures[8]" which concerned engineering wide process improvement including both software and systems engineering. Besides process experience he has recently been a software team lead on a Satellite Payload project and a systems engineer on a Satellite Ground System project. He has also engaged in independent research in Systems Theory which has resulted in a book of working papers called Reflexive Autopoietic Systems Theory[9]. A new introduction to this work now exists called *Reflexive Autopoietic Dissipative Special Systems Theory*[10]. He has given a tutorial[11] on Meta-systems engineering to the INCOSE Principles working group. He has written a series on *Software Engineering Foundations* which are contained in the book Wild Software Meta-systems[12]. He now teaches a course in "Software Systems Requirements and Design Methodologies" at University California Irvine Extension.

[end of document]

[5] http://server.snni.com:80/~palmer/disab.html
[6] http://server.snni.com:80/~palmer/fbpath.htm
[7] http://server.snni.com:80/~palmer/resume.html
[8] http://server.snni.com:80/~palmer/advanced.htm
[9] http://server.snni.com:80/~palmer/refauto2.htm
[10] http://server.snni.com:80/~palmer/autopoiesis.html
[11] http://dialog.net:85/homepage/incosewg/index.htm
[12] http://server.snni.com:80/~palmer/wsms.htm