SOFTWARE ENGINEERING FOUNDATIONS

A Paradigm for Understanding Software Design Methods

Software Systems Meta-methodology

Kent D. Palmer, Ph.D.
Software Engineering Technologist
PO Box 4402 Garden Grove CA 92642
palmer@netcom.com

The foundations of the Software Engineering discipline[1] have yet to be articulated definitively. Like all engineering disciplines, little thought is given to the theoretical foundations of the discipline beyond borrowing from the residue of scientific theory and results. The first part of this essay suggested that this may be inappropriate with regard to the engineering of software, because it may have a different ontological foundation from other kinds of things we are more used to encountering in our scientific and technical endeavors. Because of this, a radical paradigm for understanding software engineering was ventured. Whether or not this paradigm which explains many of the difficulties in creating large software systems is ultimately proven correct, it is still necessary to understand the central features of the software discipline.

[1] A definition of this discipline may be found in Peter Freeman’s Software Perspectives See Bib#. See also Bib#.

These core features taken together make up the essence of software practice directed at producing and controlling software entities. The kernel of this set of practices concerns the nature of software design methods. Design methods are taken to be the heart of the software engineering practice. It is through these methods that the software entity comes into being in the software development process through a series of essential transformations. So moving from the nature of the software entity, we must next consider the nature of the software design methods. At this point, no general theory of software design methods exists. What does exist is a plethora of specific methods proposed by experts vying for predominance. What is necessary is a comprehensive examination of the field of methods through a kind of meta-methodological study of software methods. George Klir defines meta-methodology as follows.

> In order to manage the complexity involved in the solution process, systems problems can rarely be handled without any simplifying assumptions. However, simplifying assumptions can be introduced in each problem in many different ways. Each set of assumptions reduces, in a particular manner, the range of possible solutions

and , at the same time, reduces the complexity of the solution process.

Given a particular systems problem, a set of assumptions regarding its solutions is referred to as a methodological paradigm. When a problem is solved within a particular methodological paradigm, the solution does not contain any features inconsistent with the paradigm.

It is reasonable to view a paradigm which represents a proper subset of assumptions of another paradigm as a generalization of the latter. Given the set of all assumptions which are

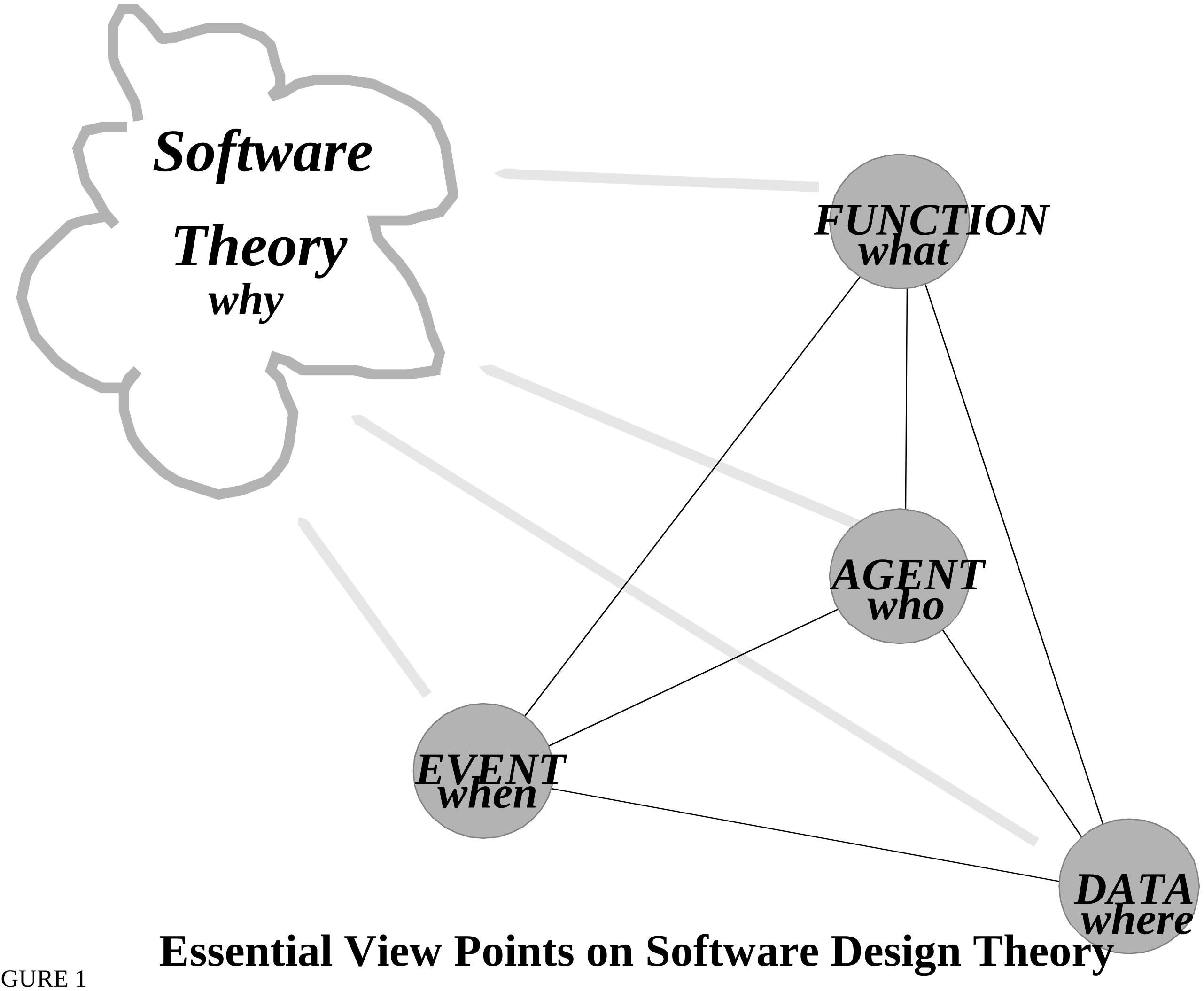


FIGURE 1 **Essential View Points on Software Design Theory**

considered for a problem type, their relation "paradigm A is more general than paradigm B" ... forms a partial ordering among all meaningful paradigms associated with the problem type. ...

Paradigm generalization is a current trend stimulated primarily by the advances in computer technology. Any generalization of a paradigm extends the set of possible solutions to the problem and makes it possible in many cases to reach a better solution. At the same time, however, it usually requires a solution procedure with greater complexity. The study of the relationship between possible methodological paradigms and classes of systems problems is a subject of systems meta-methodology. This is an important new area of research in which little has been accomplished as yet. The central issue of systems meta-methodology is to determine those paradigms, for various classes of problems and the current state of computer technology, which represent the best compromise between the two conflicting criteria -- The quality of the solution and the complexity of the solution procedure. The main difficulty in this investigation is that there are usually many alternative solution procedures which can be developed for a given problem under the same methodological paradigm.

Another issue of systems meta-methodology is the determination and characterization of clusters of systems paradigms that usefully complement each other and may thus be effectively used in parallel for dealing with the same problem. Together, they may give the investigator much better insight than any one of them could provide alone.

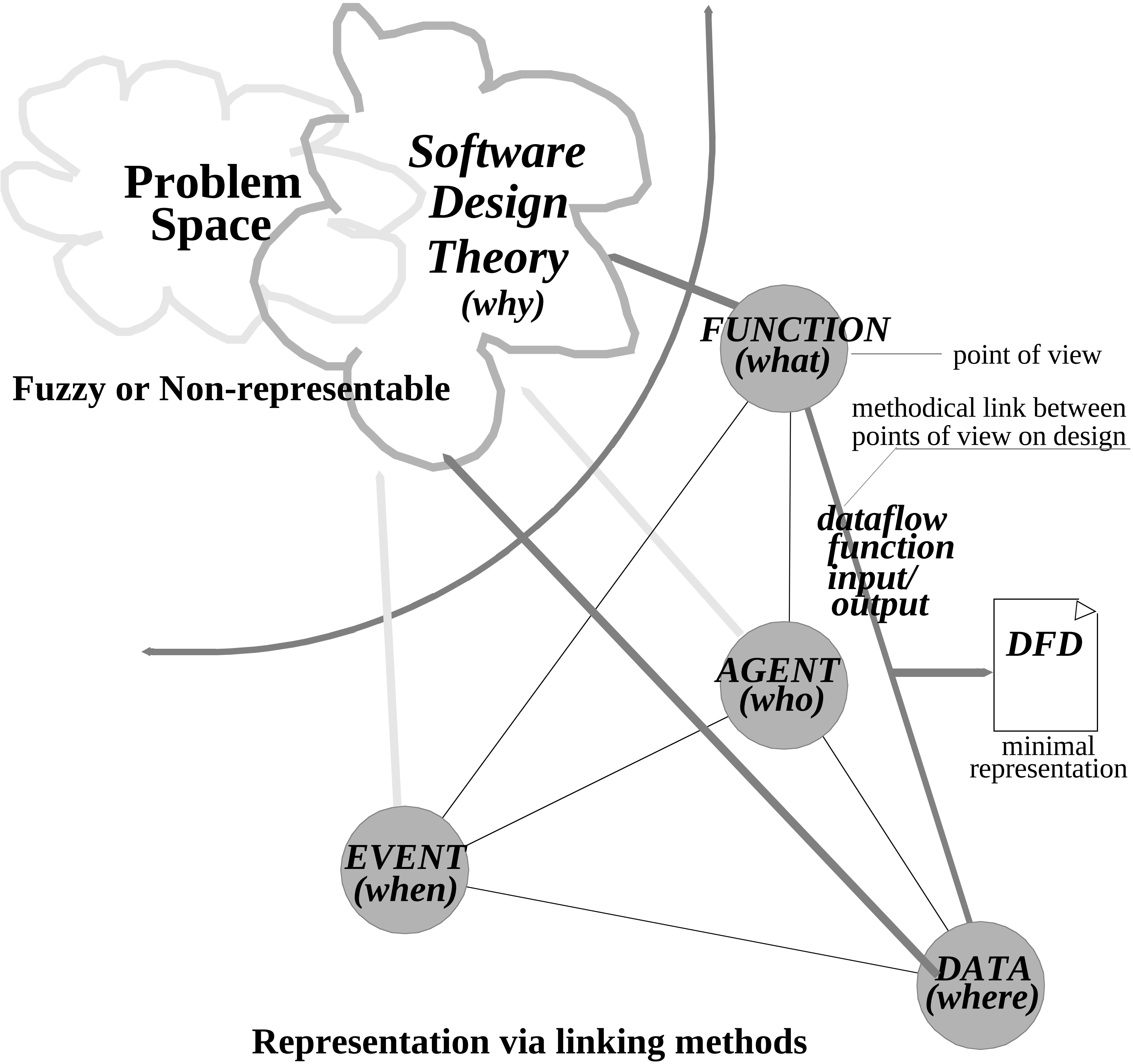


FIGURE 2 **Representation via linking methods between different points of view**

> Every mathematical theory that has some meaning in terms of a systems problem-solving framework ... is actually a methodological paradigm. It is associated with a problem type and represents a local frame within which methods can be developed for solving particular problems of this type. One of the roles of systems meta-methodology is to compile relevant mathematical theories and identify their place in the overall problem space. Another of its roles is to propose new meaningful paradigms; the ultimate goal is to characterize and order all possible paradigms for each problem type. Since the recognition of a new paradigm is an impetus for developing a new mathematical theory, comprehensive investigations in systems meta-methodology will undoubtedly be a tremendous stimulus for basic mathematical research of great pragmatic significance. Mathematics is thus a contributor to systems problem solving as well as a beneficiary of the later.[2]

Software systems meta-methodology concerns the relation of software design methods for building large real-time software systems to each other. It specifically refers to the definition of the ground that allows different methods to be related to each other and compared. This ground for comparison has not as yet been the subject of study or debate within the software engineering community. This is due to the nascent state of the software methods themselves. There are a plethora of new software methods being proposed by

[2] George Klir in ARCHITECTURE OF SYSTEMS PROBLEM SOLVING pages 18 -19 (See Bib#)

experts in the field. Many of these are mentioned in the bibliography of this paper.[3] Surveys of software methods already exist.[4] Also, a few taxonomies of methods and enabling tools, such as that produced by the Software Engineering Institute (SEI), [5] have

[3] For instance Hatley/Pirbahi (Bib#), Neilson/Shumate(Bib#), Ward/Mellor(Bib#), Shaler/Mellor(Bib#), etc.

[4] MCC Survey (See?)

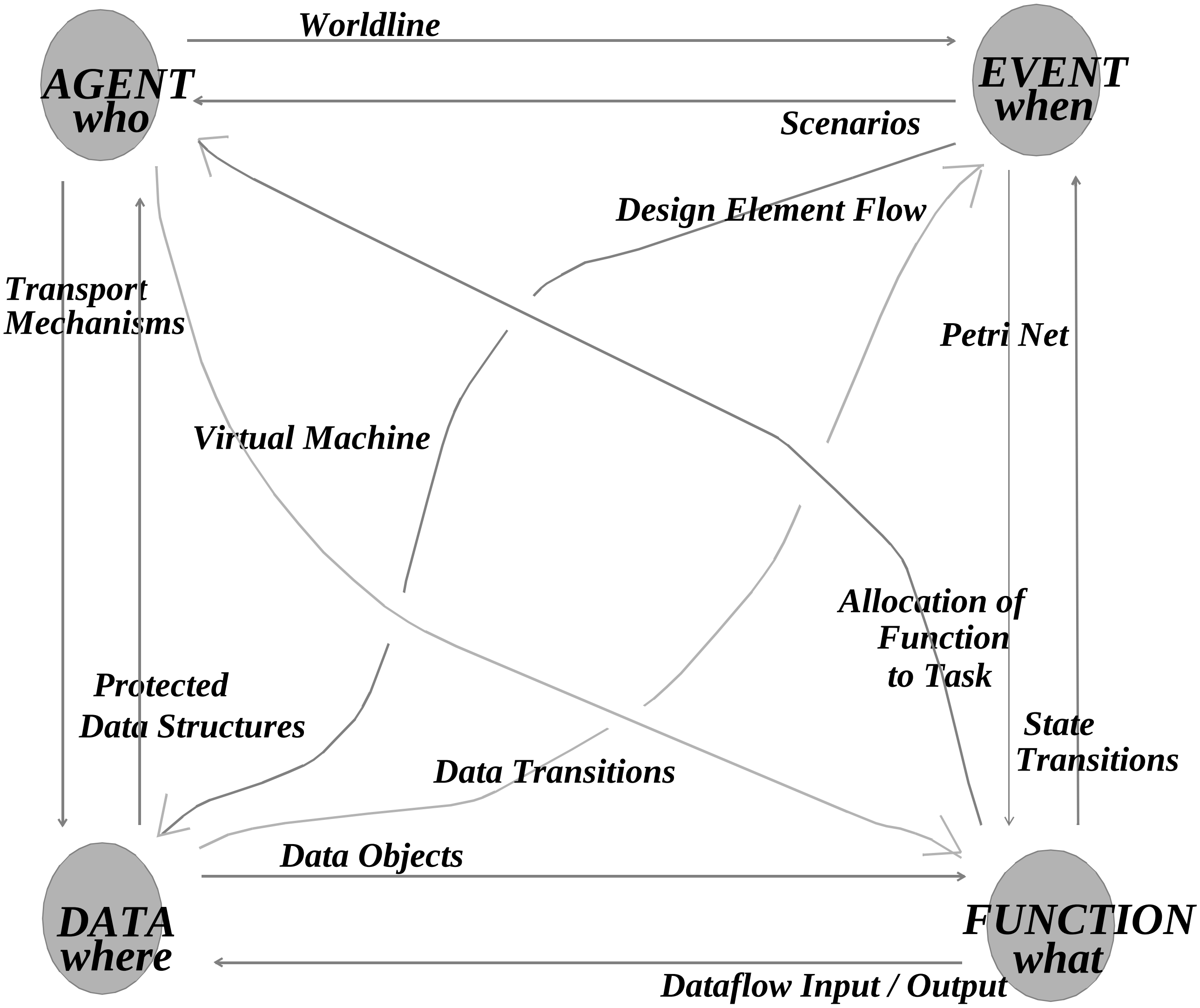


FIGURE 3 *View Points Combine to Give Minimal Design Methods*

been created. The efforts of surveying and cataloging various software methods is very important for the growth of software engineering design knowledge. However, this paper will seek to extend our appreciation of software methods in a completely different direction. Here the emphasis will be upon establishing a software engineering paradigm which will make it possible to define the grounds against which any software method can be understood and compared reasonably to other, both similar and dissimilar, software methods. A key part of the establishment of this paradigm was developed in the first section of this paper. There a foundation was established to approach the definition of the unique aspects of software. Building upon that foundation, this section will attempt an analysis of the grounds for software methods. This analysis will proceed from existing methods, but will focus on the creation of a hypothesis as to what the total field of software methods is like overall. The concentration on the field or ground that supports all possible methods is what distinguishes this study as meta-methodological rather than methodological.

[5] SEI Taxonomy (See?)

Meta-methodology also concerns the relation of software methods to the rest of science and scientific method in general. In fact, this relation will be our starting point for extending the paradigm laid out in the first section of this essay. Software engineering has a special relation to scientific methods which has already been explored. A corollary of this special relation is the relation between software engineering and general systems theory. When we speak of general systems theory, we will be referring to the work of George Klir as it appears in Architecture of Systems Problem Solving [6] (ASPS) and his numerous articles. Klir has created an excellent abstract representation of what has been called formal-structural systems. In this study, we will take this representation to be definitive of what is meant by formal-structural systems even though many other representations exist in specific fields, like Chomsky's Transformational Grammar. Klir has done us the service of expressing the essence of all the formal-structural systems in a single concrete representation. He is to be commended on the scope and detail of his generic rendering of what separately appears in many disciplines

[6] See Bib#

with various contents. Understanding how formal-structural systems work, regardless of the details of content contributed by the various disciplines, is an arduous task which is much simplified with an abstract model as a point, of departure. This essay will use Klir's representation as its starting point and any questions as to what is meant by a formal-structural system should be referred to his book which merits serious study.

In fact, Klir's ASPS has a special significance for software engineers. It should become the central reference for software engineering's concept of what constitutes a 'system.' We produce 'systems,' yet it is difficult to get good definitions of what constitutes a 'system' and what it means to produce one. Definitions of 'system' tend to be vacuous. Like the many fundamental concepts, 'system' suffers from a lack of rigor, and most definitions of it are so vague as to be almost meaningless. When we define a 'system,' it is really its relation to formal and structural underpinnings that make it meaningful. The full articulation of the formal-structural underpinnings of systems gives real meaning to this all encompassing term.[7] Software engineers build *'software* *systems'*. By knowing what software *is*, i.e. the manifestation of the singularity [8] of pure immanence [9] in the differing/deferring [10] of automated writing, <u>and</u> what a system *is*, i.e. the fully articulated formal-structural relation between elements, then it becomes possible to take a deep look at what '*software systems'* <u>really</u> *are*. This deep look is the purpose of software systems meta-methodology. Software methods are abstractions of software systems which we can manipulate and know that those manipulations are as good as manipulating the written code itself which may not yet exist. Our software design methods are pictures of the essentials of software systems. The better our picture of those essentials, then the better will we be able to design software systems. Meta-methodology concerns our getting the best possible picture of the essentials of software systems. We already know that in the very term 'software systems' there is a clash between the essential non-representability of software theory and the representable nature of formal-structural systems. Formal-structural systems are precisely a means of representing

[7] See Bib#; See Also Bib#.

[8] Singularity is used in the same sense here as in physics where it is a point within the area coverred by a theory where the theroy breaks down. See Bib#.

[9] Immanance is the opposite of Transcendence. What is Immanent never appears in any form. Similar to the concept of the Unconscious in Psychology only applied to Ontology. See Bib#.

[10] See Bib# and Bib#.

things that cannot be captured by formal systems alone. In fact, anything that involves time must be represented with structural additions to formal systems. Formal-structural systems are a means of making things that cannot be fully made present-at-hand appear as if they could be rendered fully present-at-hand. Formal-structural representations have proved themselves in physics to be very powerful means of representing transforming objects in the world. Thus, in 'software systems' there is a confrontation between a powerful form of representation and something that is inherently non-representable. This clash is an interesting phenomenon by itself.

We need to know what formal-structural system are in order to understand their relation to software. Klir gives a good representation with which to work. Yet, beyond that there is a special relation between software and the general systems science that Klir is attempting to found. Software representations are the data for Klir's general systems science which has empirical as well as theoretical aspects. Abstract representations of systems are represented as computer programs and tested. Those tests are meant to reveal inherent properties of the general systems architectures which have been abstracted from similar concrete images in various disciplines. This brings an interesting point to the foreground. General systems science itself has an intrinsic relation to software. It seeks to be more than an empty abstraction by discovering characteristics of systems by manipulating software representations of systems architectures. So on the empirical side of general systems theory, it too must confront the essential nature of software as effected by the manifestation of pure immanence. While general systems theory seeks to present a purely present-at-hand picture of what constitutes a formal-structural system, it is also dealing with the non-representable nature of software theory. The empirical discoveries of general systems science about systems architectures is in some sense an exploration of the same issues that are driving software engineering. In fact, software engineering has an intrinsic relation to general systems science which is similar to its relation to computer science. From the discipline of computer science it seeks the knowledge about how software works in a small scale on various kinds of hardware. From general systems

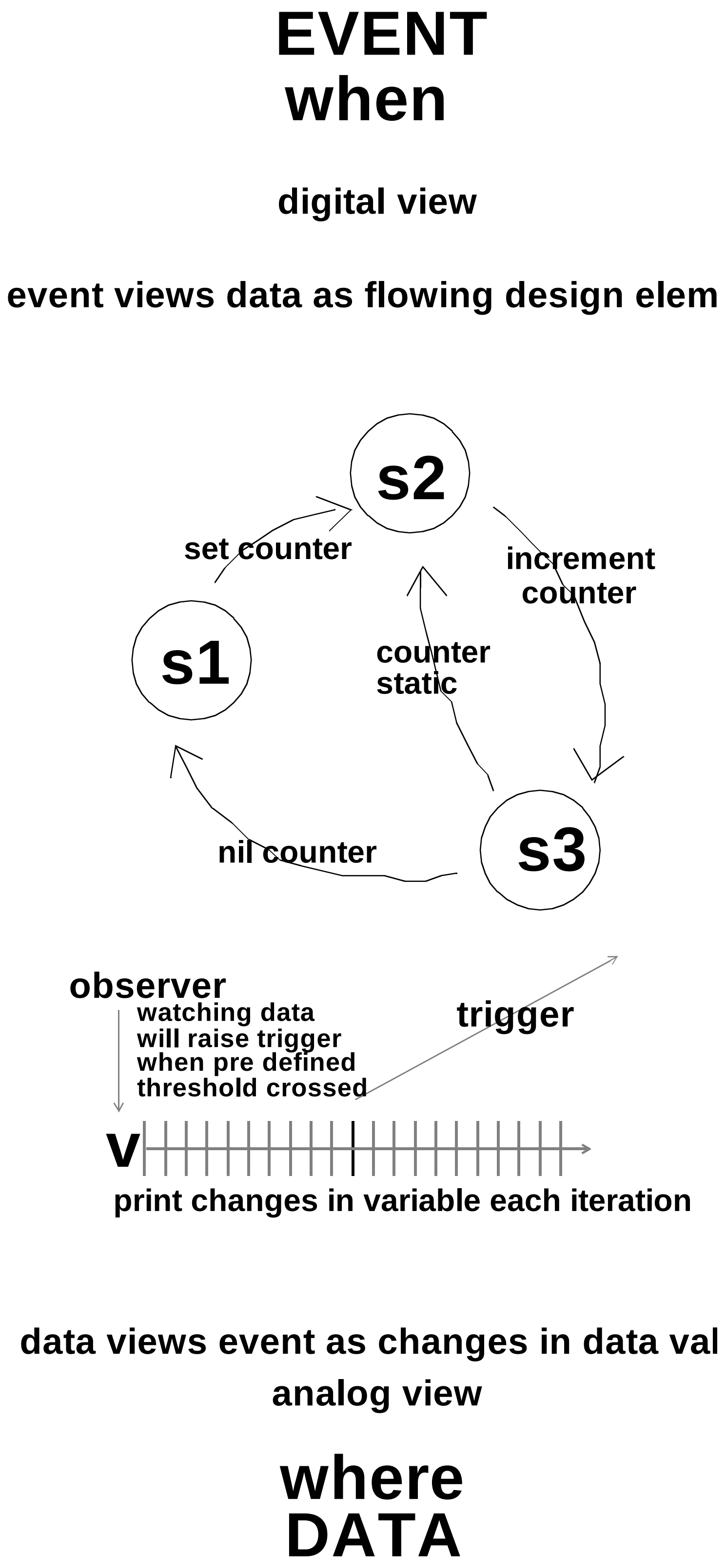


FIGURE 5

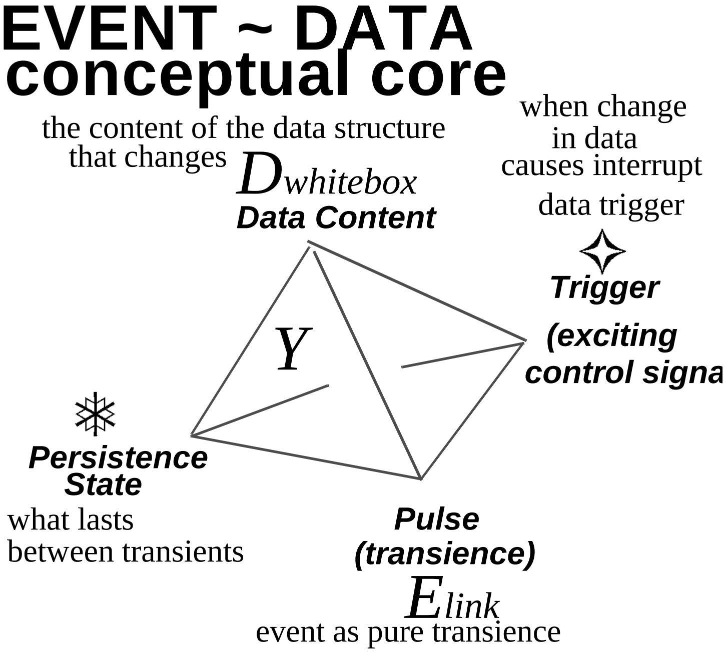


FIGURE 6

The minimal methods mentioned here are not well developed in the literature. Event/Data relations are basically of two types. One may be looking at variables modified by a system and making sure they are altered in the correct way by the executing program. Or one may be looking at the data structures that make up the mechanism of the software itself and seeing whether these data structures, like pointers and buffers, are used correctly. The view that design elements flow through a systems state is a major improvement over the conventional concept of design elements as a static clockwork mechanism. Each of these two views corresponds to the internal relations between Data and Event viewpoints. The fact that methods do not exist for this bridge is very interesting. At first, I could not even think of examples of these method bridges. It took some time to realize that this is because we look on our systems from the point of view of persistence of data structures, not their change. The primitive nature of these methods shows that we are still applying Being1 as the criteria for the systems we build, and the function perspective inherent in Being2 has not impacted our design approaches as yet.

The concept core shows how these two ill-defined methods might be seen as addressing a single set of problems. We are concerned with data as content (interior view) which varies according to transient system changes. Transiencies are Events viewed as links. The persistent state is the length of time things remain the same, which is different from the coordinating state. Remaining the same depends on the sensitivity threshold applied. Significant changes trigger interrupts.

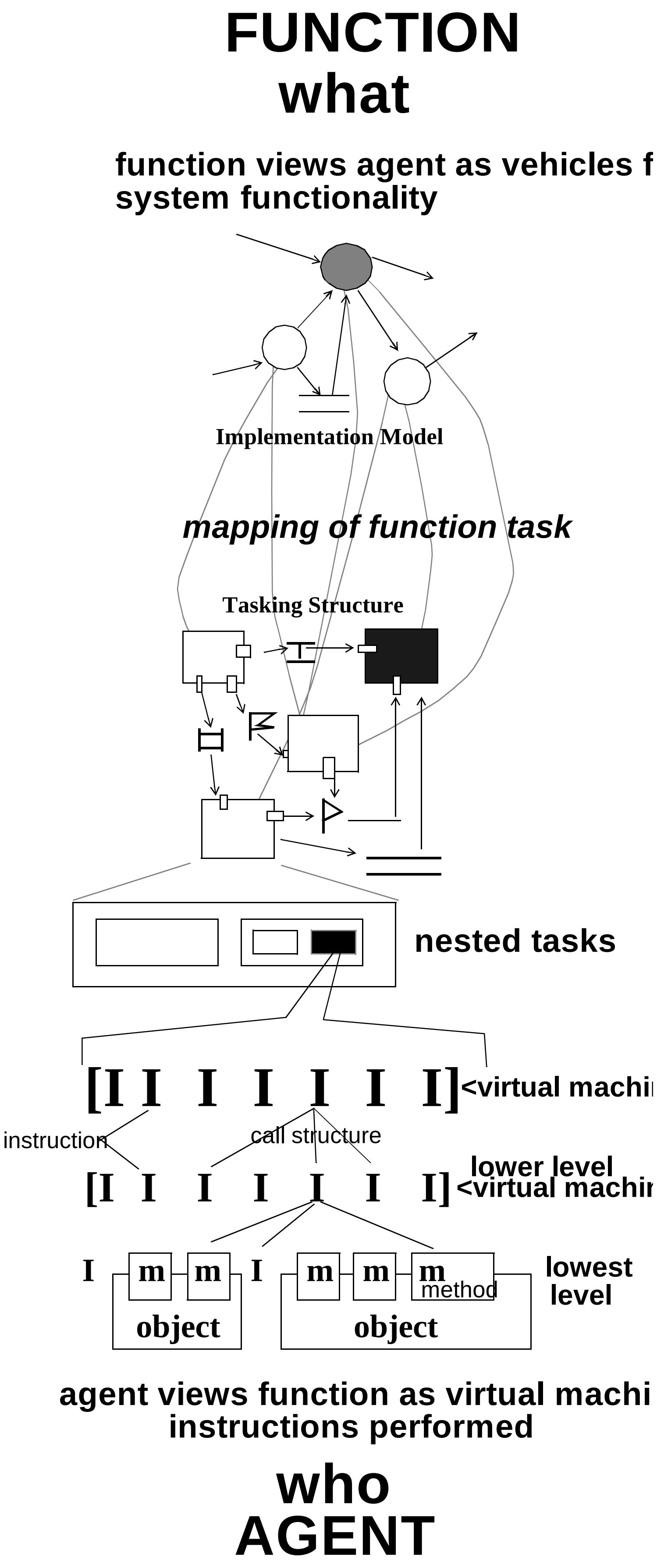


FIGURE 7

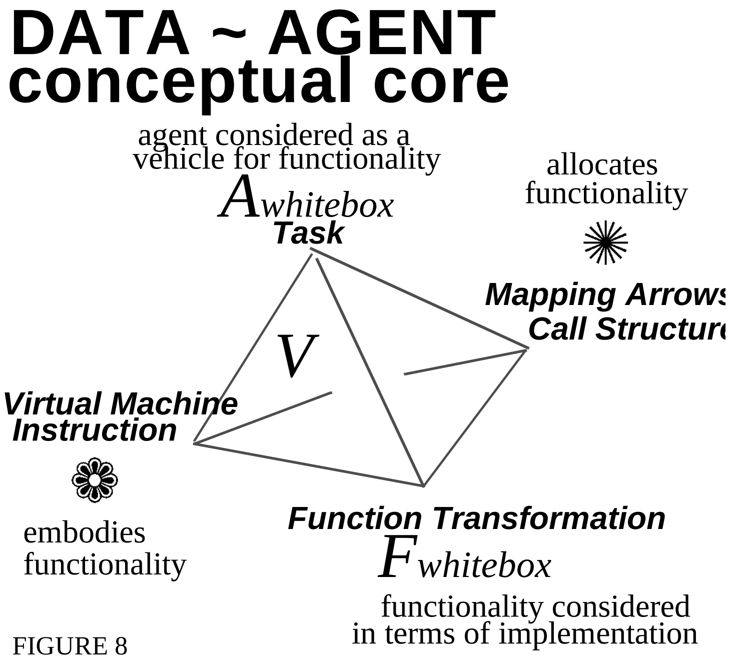


FIGURE 8

The methods presented here are further articulated by Ward and Mellor (function allocation from and implementation mode.) and Neilson and Shumante (OOD/VLM methodology). The two minimal methods here do not combine into a simple two-way method bridge. Two separate methods are referred to in attempting to describe the relationship between Agent/ Function. From the point of view of function, the main thing is that all the system functionality is allocated to the tasks so that everything that needs to be done gets implemented. The implementation model records the progressive incorporation of constraints into the essential model. However, the tasking model developed in the proceeding bridge is used here for the mapping of functions onto tasking agents that carry out these tasks. The tasking structure may consist of nested tasks. But ultimately, each task is composed of a virtual machine which is, in turn, composed of lower level virtual machines. From the task's point of view, it is the execution of virtual machine instructions that causes functional processes to be done. Thus, agents see function in terms of these procedures and functions that embody different aspects of system functionality.

The task here is seen as a white box which we are looking inside of, either to place functionality by mapping, or by seeing what functions are performed within the task. In terms of functions, we are concerned with the type of transformation that is to be performed, either in terms of a requirements statement or in terms of the actual performance vehicle. The virtual machine and the mapping arrows actualize the relations between these two concepts. This bridge is the most complex all.

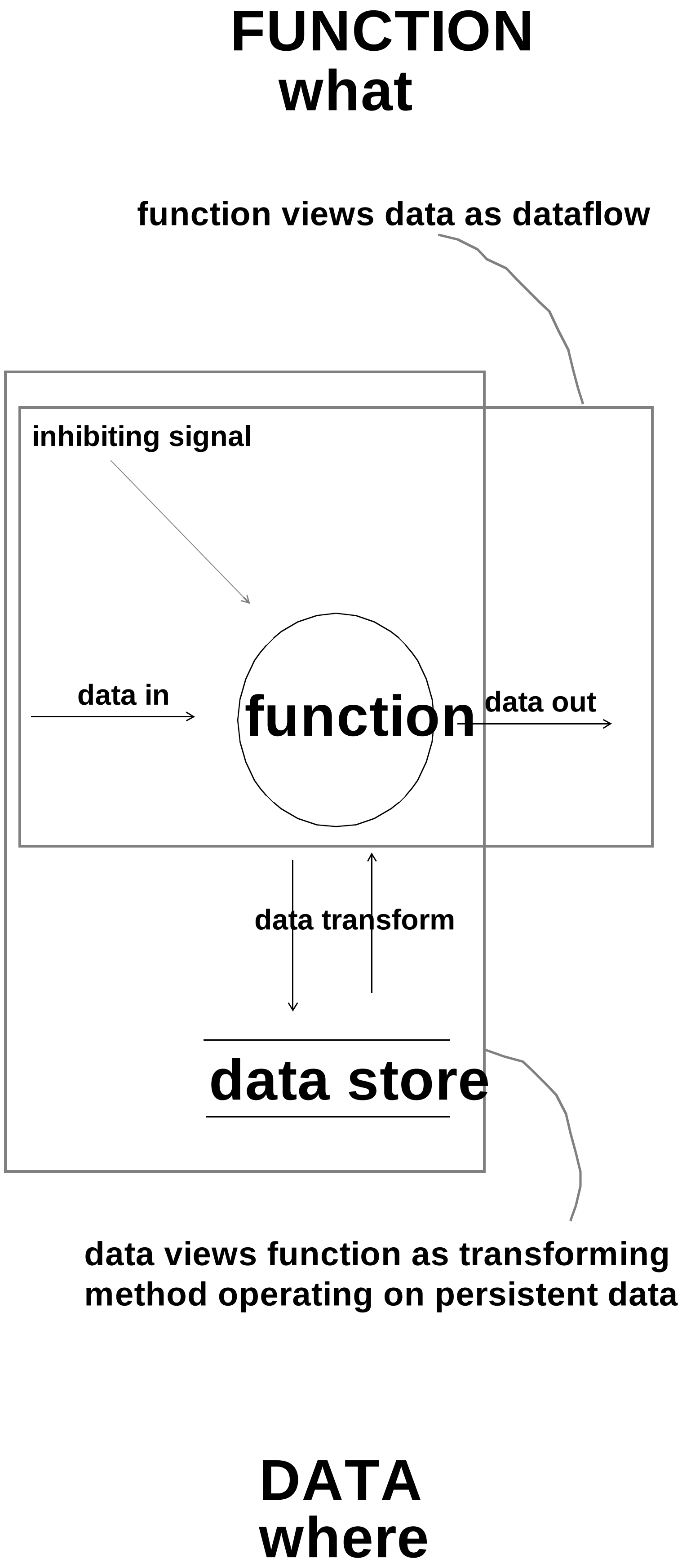

FUNCTION
what
function views data as dataflow
inhibiting signal
data in
function
data out
data transform
data store
data views function as transforming method operating on persistent data
DATA
where

FIGURE 9

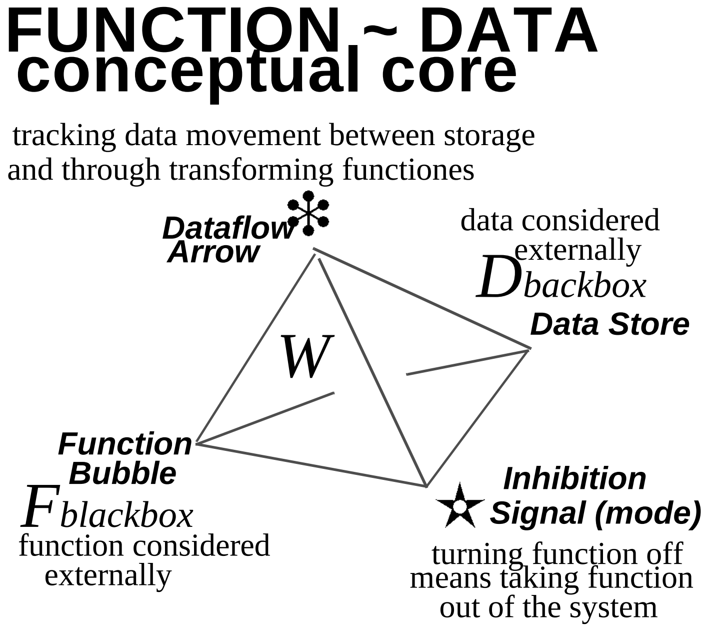

FUNCTION ~ DATA
conceptual core
tracking data movement between storage and through transforming functiones
Dataflow Arrow
data considered externally
Dbackbox
Data Store
W
Function Bubble
Fblackbox
function considered externally
Inhibition Signal (mode)
turning function off means taking function out of the system

FIGURE 10

The Dataflow is further articulated by Hatley and Pirbhai in terms of its relation to control models. The key concept that they have added is the idea of inhibition control signals to express system modality. This control is not adequate for real-time design, although it may suffice for the development of the essential requirements model. The Dataflow technique is very successful and shows the power of constructing two-way method bridges. The dataflow is used in requirements analysis and specification phase to build the essential model of the system. It is then used in architectural design phase to elaborate the essential model into the implementation model by introducing constraints. The conceptual core is isomorphic to the major elements in the representational scheme.

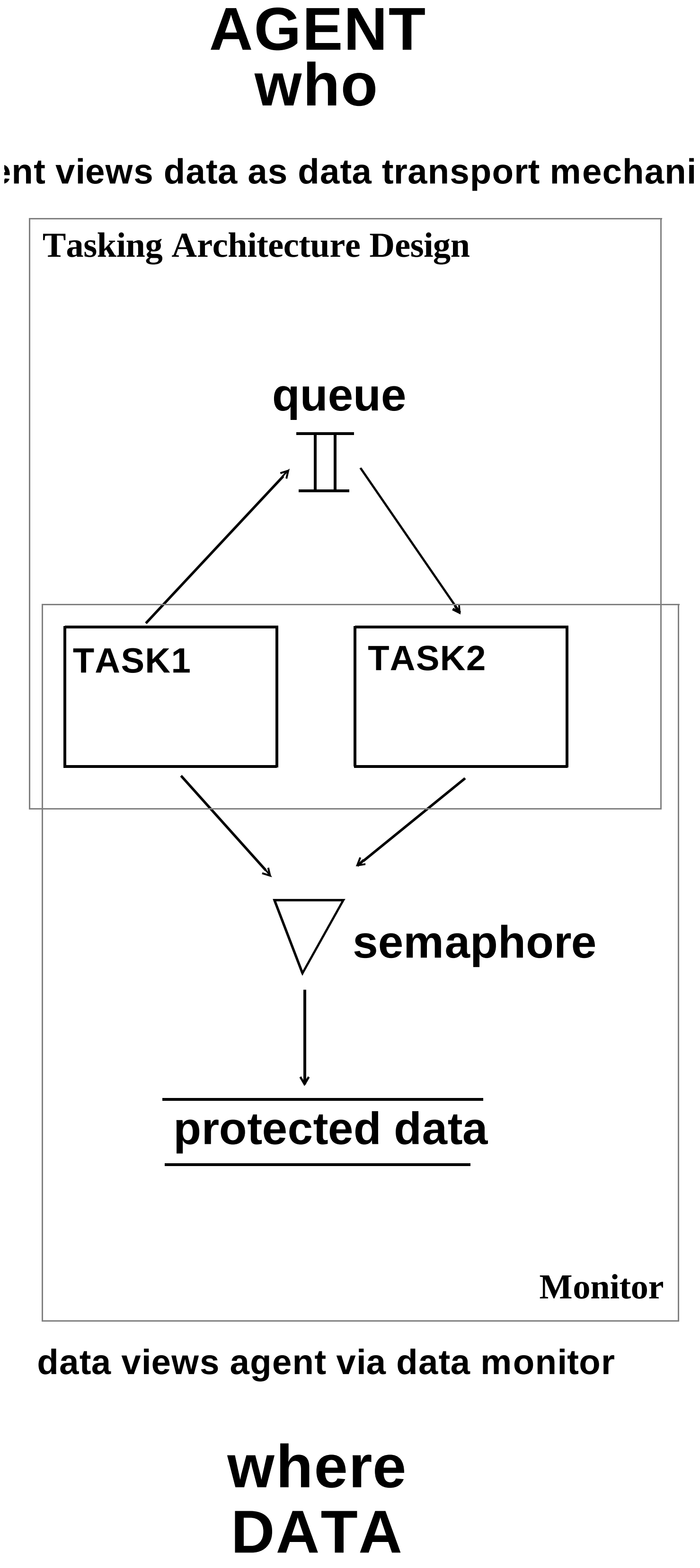

AGENT
who
ent views data as data transport mechani
Tasking Architecture Design
queue
TASK1
TASK2
semaphore
protected data
Monitor
data views agent via data monitor
where
DATA

FIGURE 11

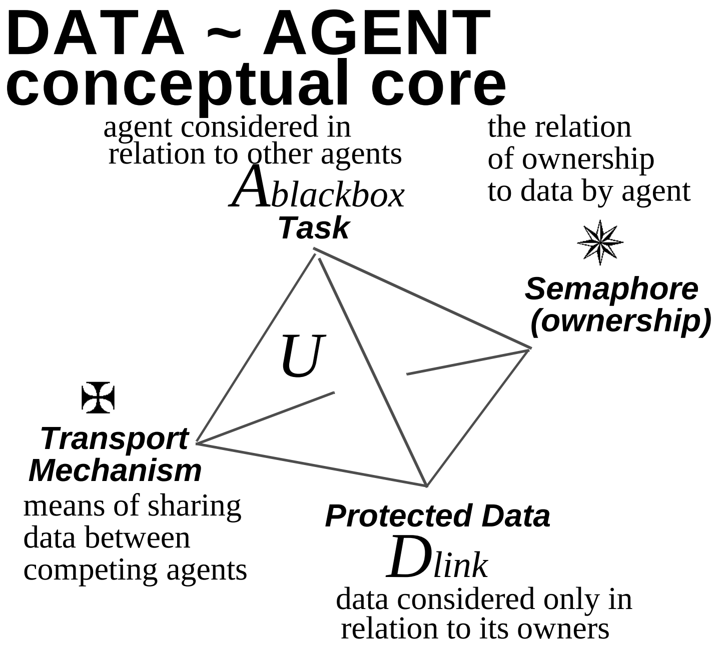

DATA ~ AGENT
conceptual core
agent considered in relation to other agents
Ablackbox
Task
the relation of ownership to data by agent
Semaphore (ownership)
U
Transport Mechanism
means of sharing data between competing agents
Protected Data
Dlink
data considered only in relation to its owners

FIGURE 12

The methods articulated here are further developed by Hoare (Monitors) and Gomaa (Tasking Architecture Design). The two minimal methods combine easily into an overall architectural model similar to that of the dataflow diagram. However, here the emphasis is upon architectural issues of tasking rather than functional considerations.

The conceptual core is derived from the basic concepts displayed by this model.

Task here is treated as a blackbox because we are not concerned with what it does. Tasks are related to Protected data through their ownership shown by possession of semaphore tokens. Protected data is a liking aspect of data because protection is the inverse concept of ownership. What is owned must be protected by the owner. Tasks communicate via transport mechanisms. Transport mechanisms assure date integrity in a multitasking environment.

In this case the derivation of the conceptual core is straight forward because the methodical model has only four elements which are isomorphic to the concepts in the core.

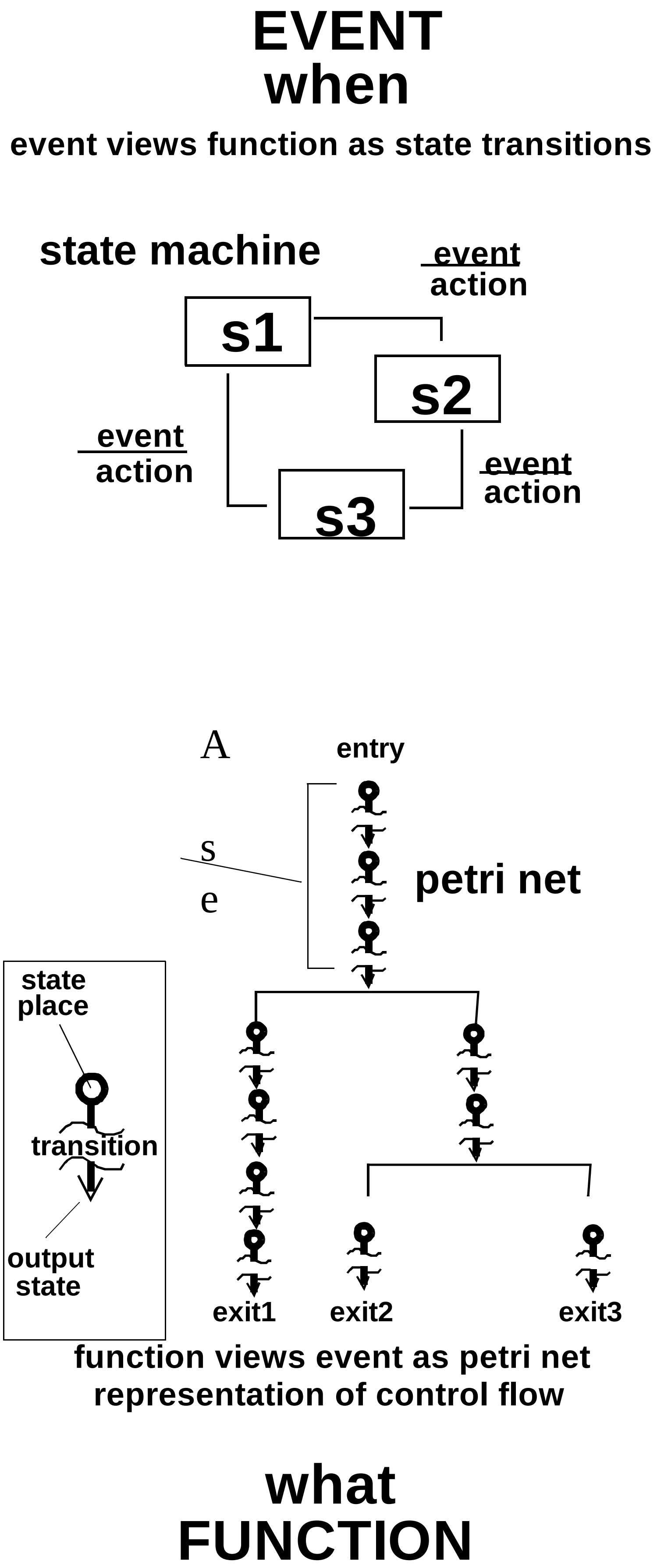


FIGURE 13

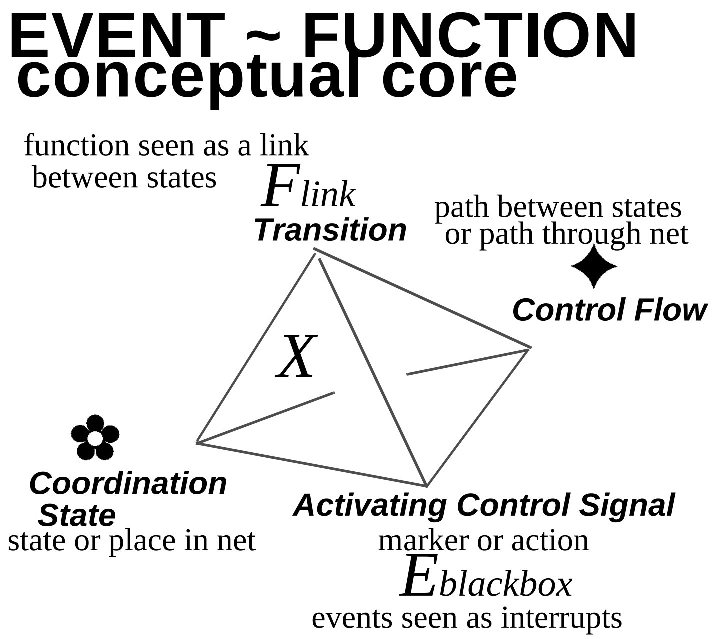


FIGURE 14

The state transition method is well explained by Ward and Mellor and by Hatley and Pirbhai. The use of perti nets for modeling control is not as well known. Its advantage is that dynamic models which pass tokens through the net can be used to track control flows throughout a complex system with multiple agents. The transition may be seen as the execution of an instruction in a virtual machine. The petri net may be reduced to form the minimal machine by collapsing all sequences of instructions into a single transitional block. In this case, it is through the petri net reduction that the transition between the petri net and the state machine is achieved. State machines invoke actions which may be seen as implementing functionality.

The concept core is abstracted from these two methods. The transition (seen as the transition in the petri net and the action called by the state machine) is considered as a linking aspect of function, that is function considered as transitions between states only. The event here is seen as the activating control signal (seen either as the event in the state machine or as the marker in the petri net). As an activating control signal (pure interrupt), the Event is displayed as a black box.

The relations between these are captured by the ideas of control flow and coordination state. The control flow shows up as the path between states in the state model, and as the path between transitions in the petri net. The coordination state shows up as the places in the petri net and the states in the state machine. Coordination states are differentiated from persistence states. Coordination states signify logical states that control the virtual machines and allow agents to coordinate.

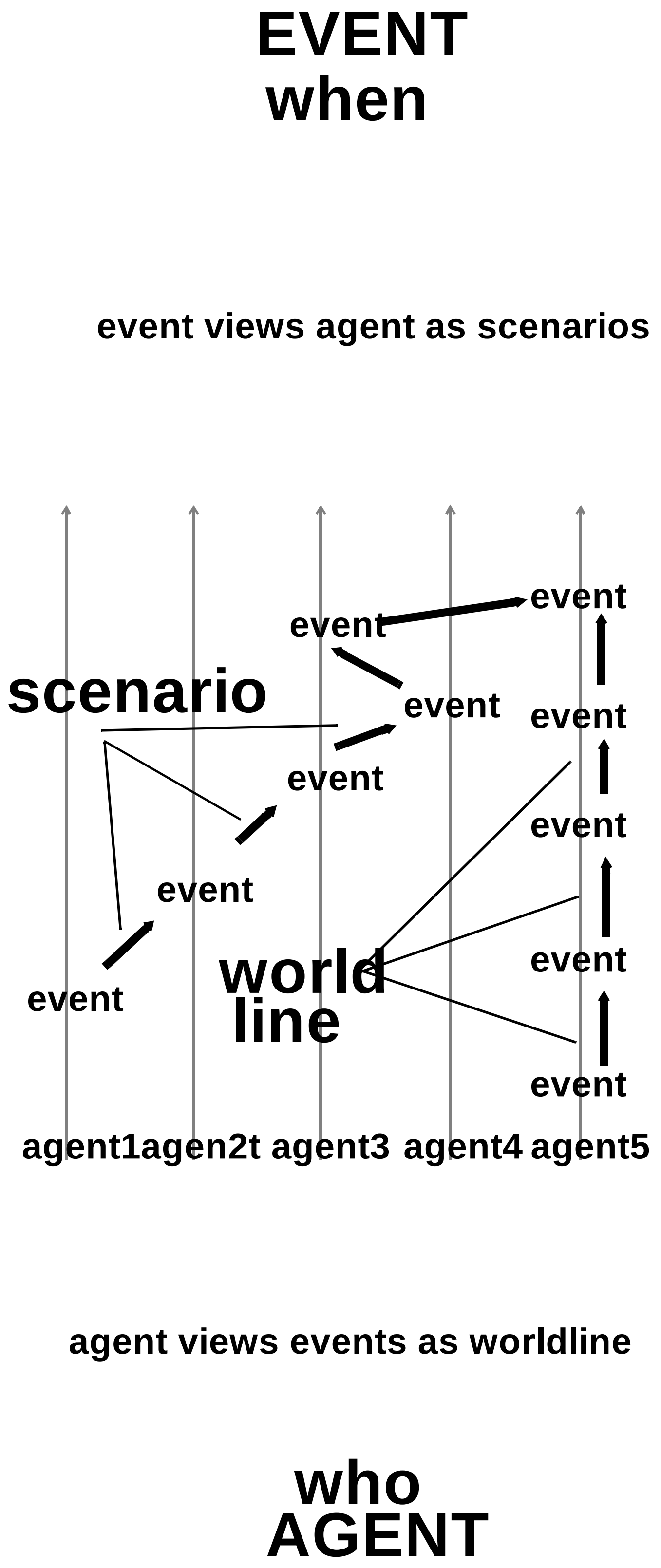


FIGURE 15

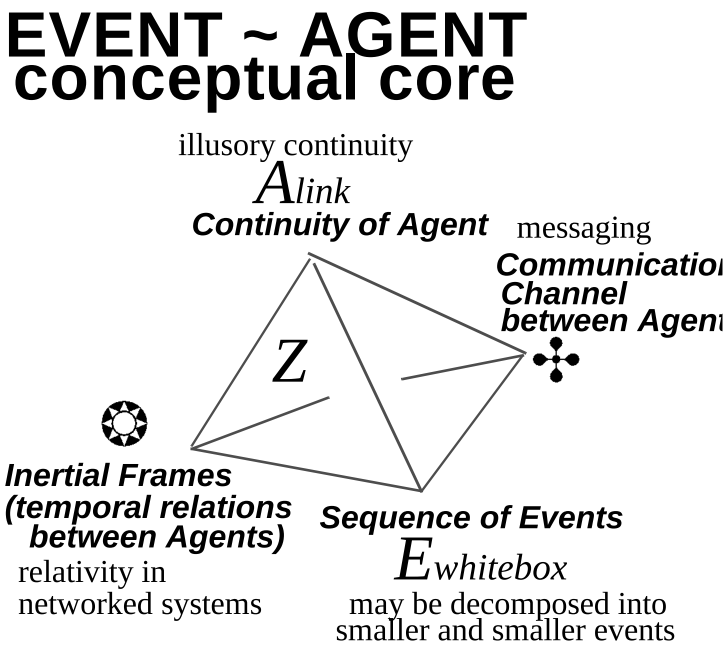


FIGURE 16

The worldline method is elaborated by Agha in the book ACTORS. The scenario method is widely used for mental simulation, but no satisfactory description of it as a methodology is known. These two methods form a two-way bridge in an atheistically pleasing way. The world line is a means of understanding the relativistic relation between actors. The scenario is the major means of knowing whether the system being designed will work properly.

The concept core expresses the inherent ideas behind these two methods. The agent has illusory continuity, and this sees the agent in its linking aspect. The sequence of events can be looked at with different thresholds of refinement. One event may be made up of many sub-events so that the Event here is seen as a white box. The concept that agents may communicate with each other, and the fact that their communication is structured relativistically, are the two other concepts that serve to bring out the relations between the continuity of the agent and the decomposition of events.

These methods are very important to the development of networked systems of any size. Now they are not widely known or appreciated. The analysis of indeterminate relativistic systems will be of prime importance in the near future. One tool that models this method bridge is VERDI, which is derived from RADDLE, developed by Vincent Shen at MCC consortium in Austin, Texas.

Three conceptual cores define a viewpoint. Each conceptual core contains an image (V) of the viewpoint being defined either in whitebox, blackbox or linking aspect. Each conceptual core contains an image (n,o,p) of the other viewpoint that it is linked to by the method bridge. Each conceptual core contains two connecting concepts (GiKebC) related to control, communication, coherence, or cohesion. Three conceptual cores define each viewpoint in a

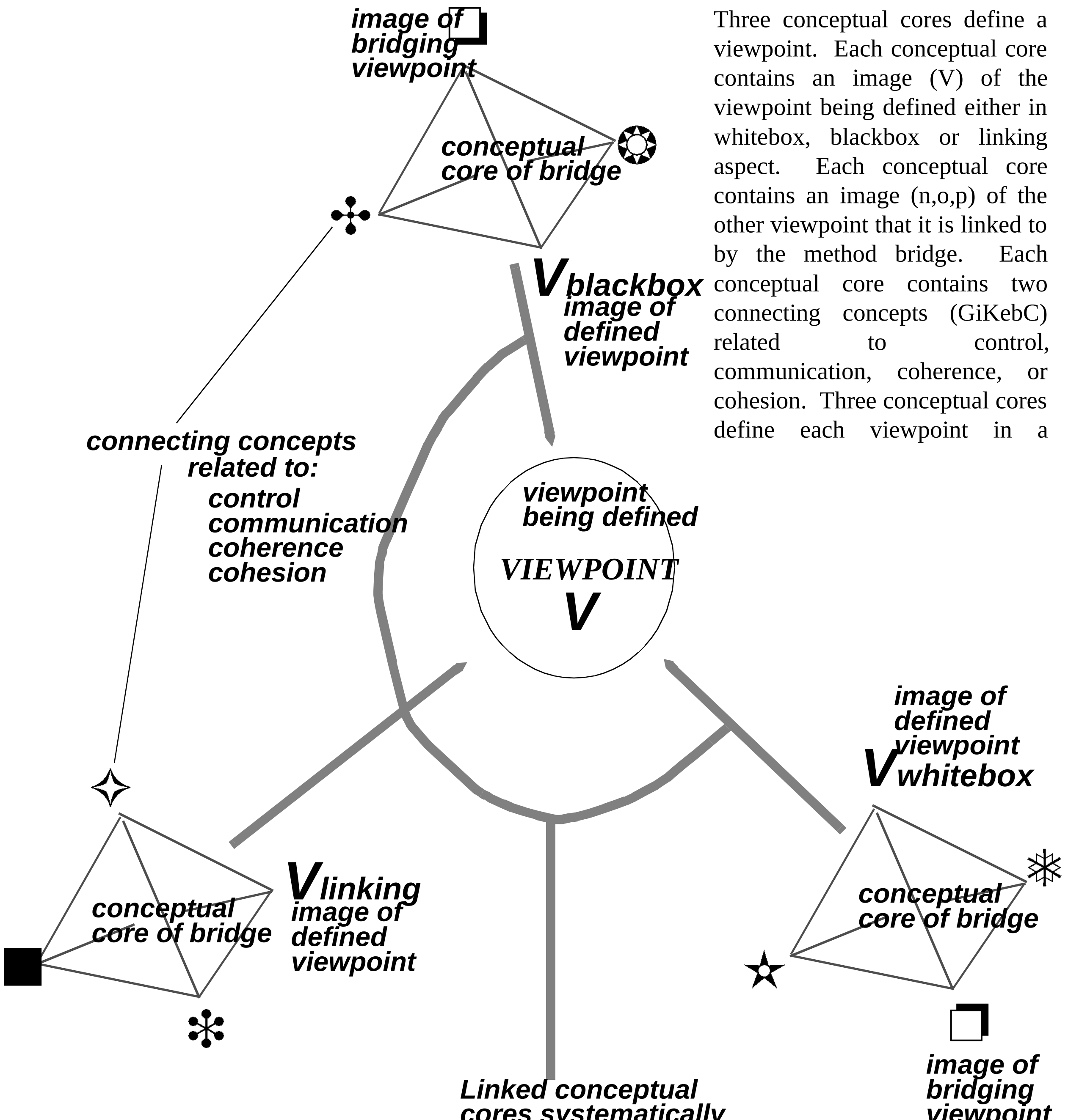


image of bridging viewpoint
conceptual core of bridge
V blackbox
image of defined viewpoint
connecting concepts related to:
control
communication
coherence
cohesion
viewpoint being defined
VIEWPOINT
V
image of defined viewpoint
V whitebox
V linking
image of defined viewpoint
conceptual core of bridge
conceptual core of bridge
image of bridging viewpoint
Linked conceptual cores systematically define a particular viewpoint on system design

DEFINITION OF VIEWPOINT USING METHOD CORES

FIGURE 17

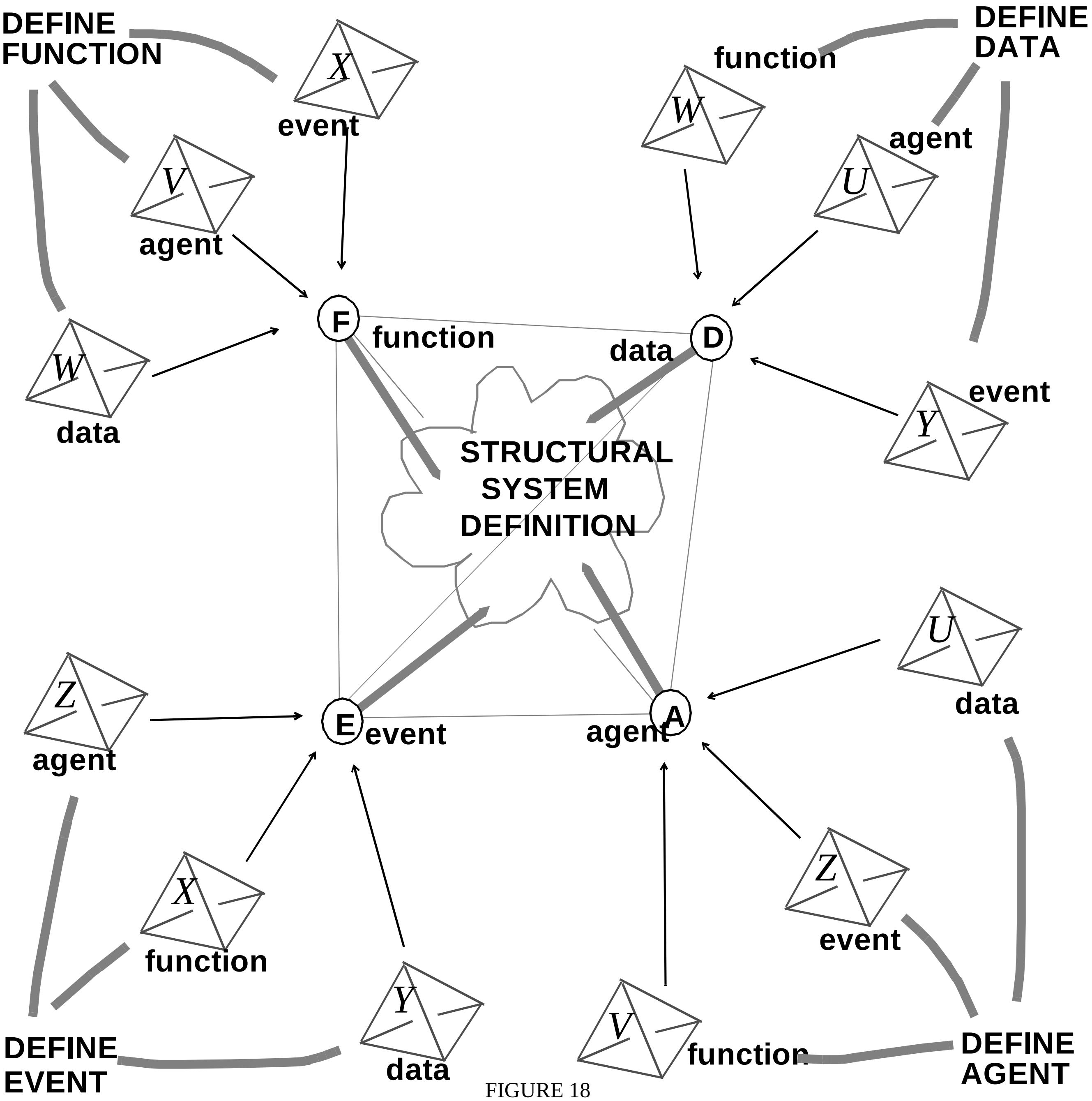
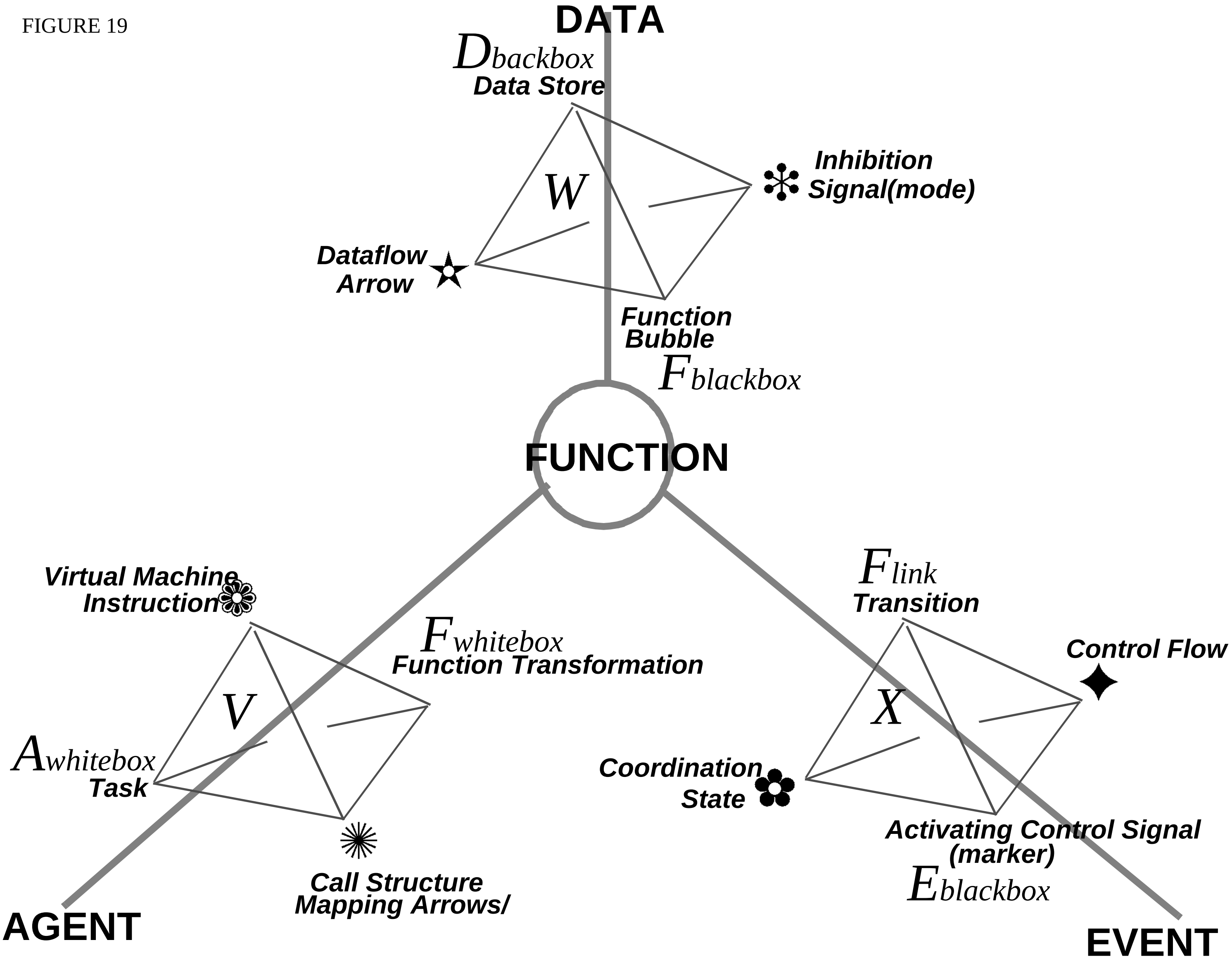

DEFINE FUNCTION
X
event
V
agent
W
data
function
DEFINE DATA
W
agent
U
event
Y
F
function
data
D
STRUCTURAL SYSTEM DEFINITION
U
data
Z
agent
E
event
agent
A
X
function
Y
data
Z
event
V
function
DEFINE EVENT
DEFINE AGENT

FIGURE 18

FIGURE 19

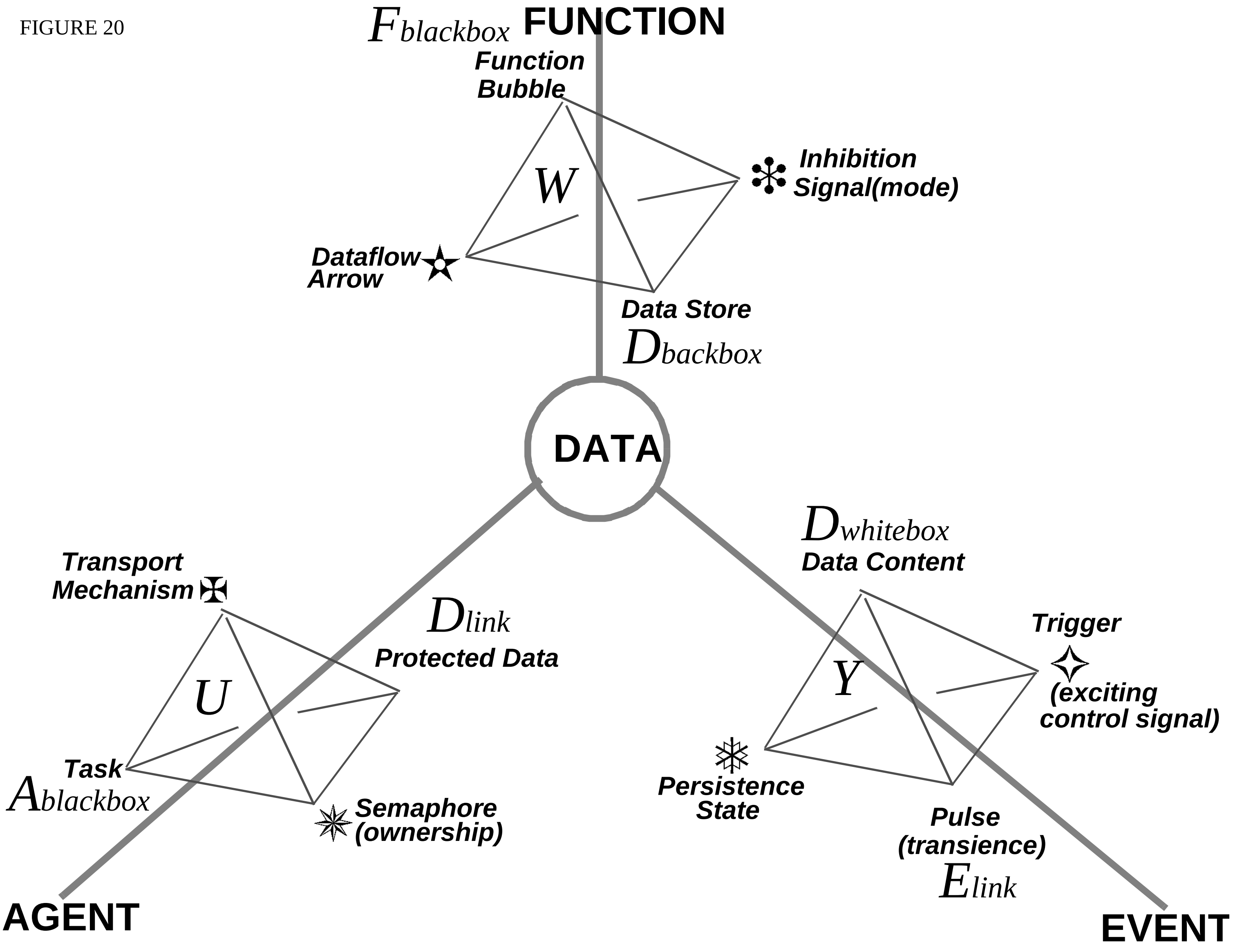


The conceptual cores of the method bridges defined in the previous six diagrams will undoubtedly change as our understanding of the application of structural systems to design improves. They are wholy derived from an analysis of existent methods empirically present in design circles at this time. The analysis of these methods is difficult due to their different states of development. From what has been seen, sometimes the derivation is clear and obvious, while at other times it is unclear. This is usually related to whether the bridge is two-way or not.

Once these conceptual cores have been derived, it is possible to show how they can be used to define the particular viewpoints of system design. In the above diagram, the function viewpoint is being defined by moving systematically between method bridges that revolve around the function perspective. By revolving around a single design perspective iteratively, a better and better picture of the system from that point of view is represented. In revolving around the perspective, it is the concepts of the bridge conceptual cores that are being used to refine the system. In each revolution, we could use a different one of theses key ideas as the guiding theme of our analysis and representation construction.

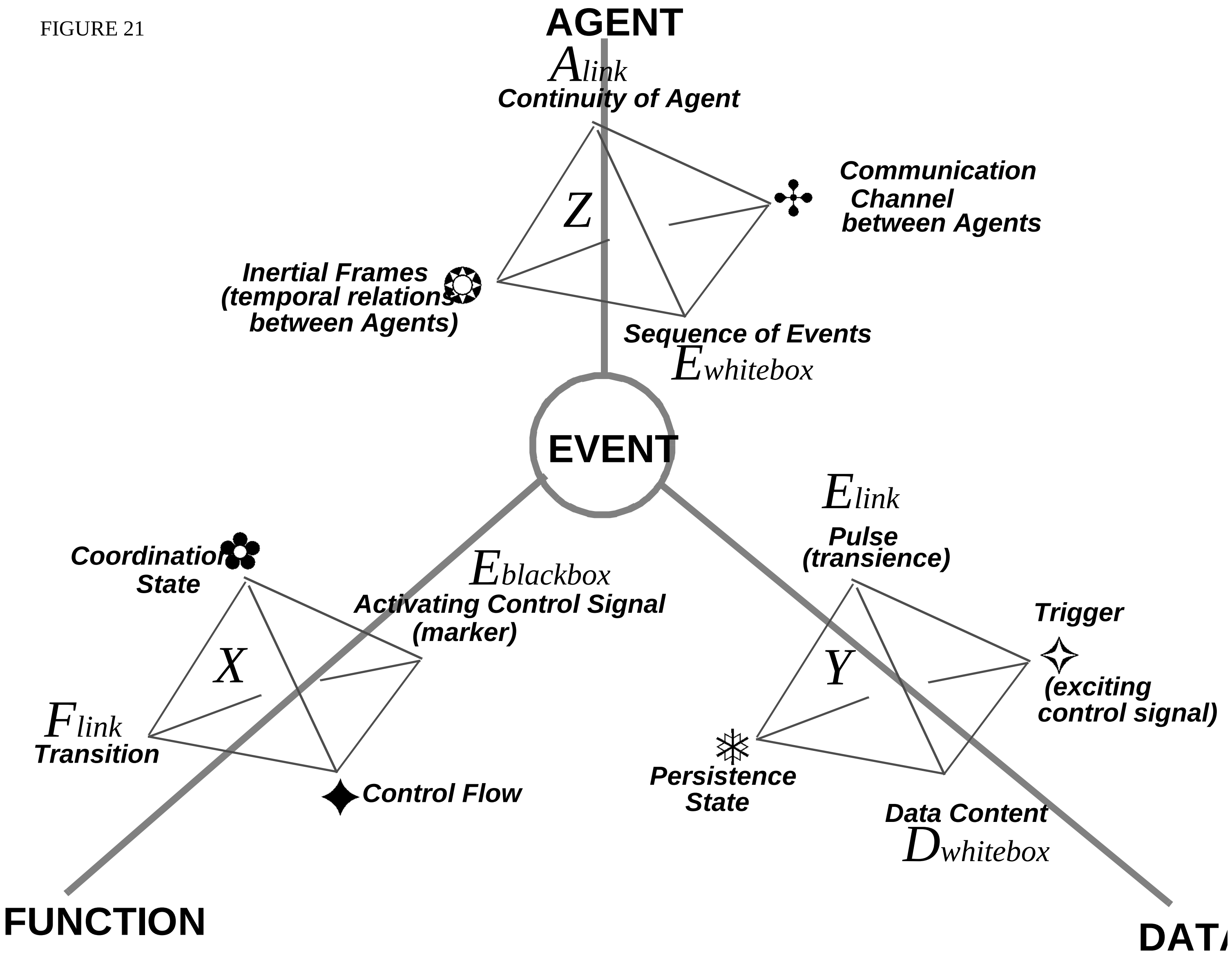
Notice that from each method bridge the Function perspective is treated differently. It is treated in turn as blackbox, whitebox and link. This variation of the treatment is precisely what allows us to articulate the system from the given viewpoint. In this step-by-step articulation the functional decomposition is developed.

FIGURE 20



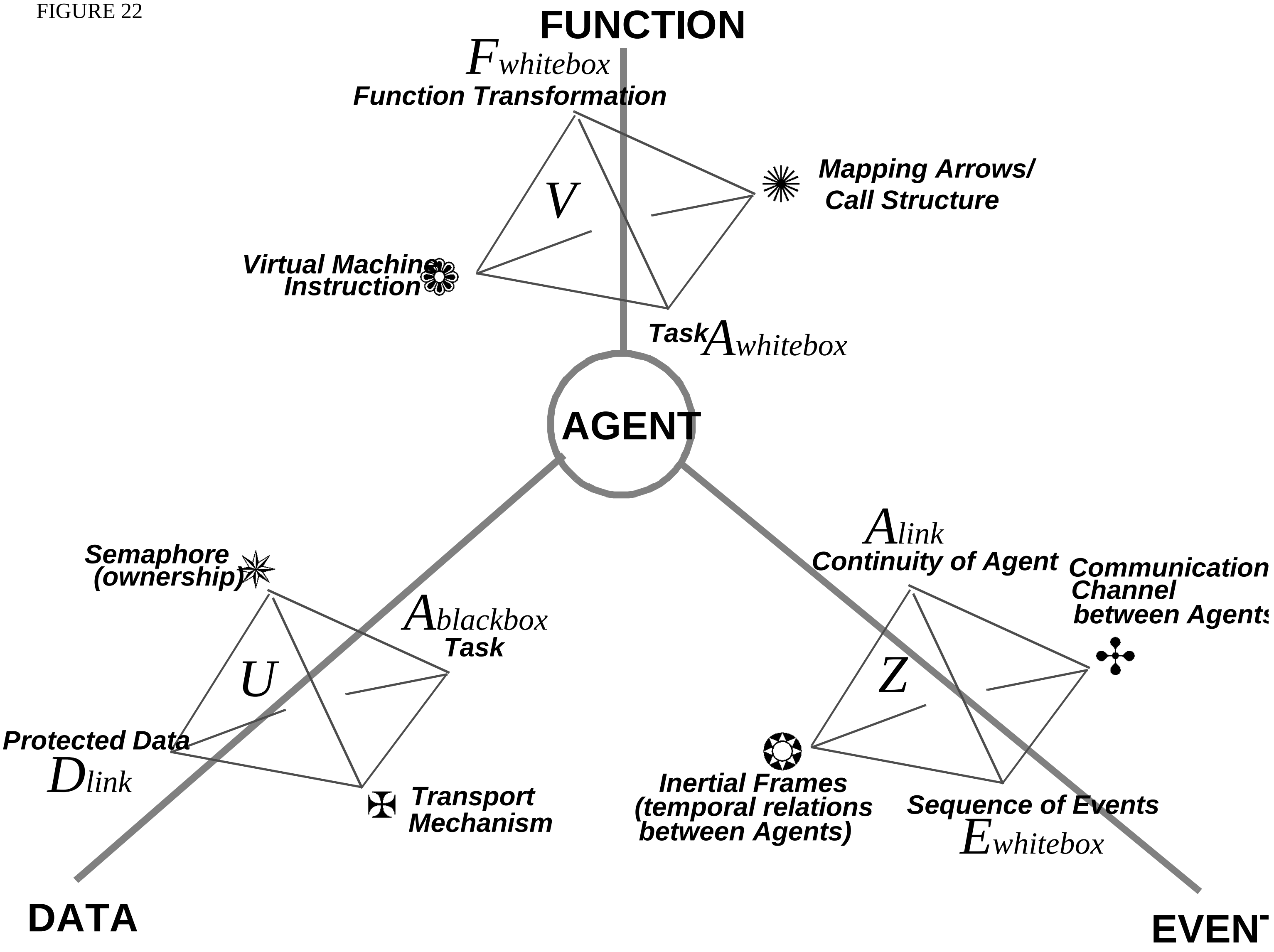
The same process of refinement is being carried out there with respect to another perspective, i.e. that of data. Method cores should be looked at in their relation to each other as they relate to a particular perspective. One is only in one perspective at a time, looking at objects created from other perspectives. Perhaps different methods should be created that link all the methods that surround a particular view- point. It is this kind of analysis that needs to be carried out beyond the refinement of the idea of the method cores themselves. All the ideas that go into describing a real-time embedded software system form a formal-structural system that relate Being^1 and Being^2 concepts to describe something rooted in Being^3. This description must always be partial, but that does not mean we cannot make descriptions. Our descriptions need to isolate the discontinuities across which our formal representations cannot cross. Here, this is done by positing that those discontinuities are related to the shift between perspectives on the software design theory. Thus, the set of conceptual cores around a particular perspective are able to form a formal-structural system which does not have discontinuities. What we are actually doing is venturing out from a particular perspective just enough to use its surrounding and thus defining methods, but we are avoiding actually crossing the bridge. This means of using the methods will result in the high definition of the software system from a particular viewpoint, and will also cause sharply defined discontinuities between a given viewpoint and another viewpoint to appear. The bridging methods will allow us to cross these discontinuities gracefully. The redundancy of method viewpoints surrounding the different viewpoints (for instance U is part of the surrounding system for both Data and Agent viewpoints) is what allows the movement across the discontinuities to be made without total disruption of the design. In other words, something is the same when you get to the other side. Speeding up the movement through the perspectives, give the illusion of encompassing the whole theory.

FIGURE 21



Really seeing the whole theory is impossible because of the hidden presence of pure immanence. However the faster and faster movement between the viewpoints will give the impression of seeing the whole system. Our focus should not be upon glimpsing the whole system, but instead upon the discontinuities that become apparent as we define the system from each perspective. It is out of all these discontinuities that all of our really difficult problems in system design come. These wicked problems express immanence as it constrains our freedom to design. The set of method cores are, in fact, the very expression of our transcendence. Here we are conceptualizing the structural system that allows us to define the software theory abstractly. This abstract representation is our means of transcending the particular systems with which we are confronted, and allows us to view them in relation to the generic software system, i.e. the pure theory. The definition of method cores is thus the expression of the purely theoretical software theory. Because of this, the development of the method cores should allow us to develop our software design metrics. Currently, metrics for code representations are becoming fairly well developed. The most popular of these are those of Halstead and McCabe. We are still, though, in a quandary of how to apply metrics to designs. It is easy to see that if we could agree upon a set of method cores, then the concepts of those cores should be what is measured in order to get quantifiable design metrics that describe the software theory considered purely abstractly as fully as possible. I suggest it is those metrics related to method cores that should be related to quality measurement.

FIGURE 22



We are now ready to transition from the relation between method cores and their associated perspectives, to the concept of the infrastructure of the formal-structural system. Note that in the case of each method core, two of its components is related to the representation of perspectives. The other two concepts have the function of relating the two represented perspectives. The question then becomes whether it is possible to formulate any relations between these mediating concepts. There are twelve mediating concepts in all, and what will be shown here is that these form an infra-structure that reveals the deep structure of the methodical structural system. The infrastructure will be represented according to the form Buckminster Fuller calls the vector equilibrium. This is an important synergistic theoretical structure. Because of the empirical origin of our mediating concepts, we do not achieve a perfectly symmetrical theoretical structure. But this is exactly the point because by considering these theoretical structures, each mediating concept is related to all the others in ways that would not be achieved otherwise. It is the consideration of the lack of symmetry of the theoretical structure in relation to the pure geometry of thought that should give us some insight into what is ultimately wrong with this way of thinking about methods. Perhaps method cores should really be five-fold instead of four-fold. When one constructs a theoretical structure, it is exactly the anomalies that are interesting. The point is that there is enough coherence to this explanation that it is worth entertaining until someone can explain more adequately the relationship of software engineering methods to each other.

So the next figure shows all the mediating concepts, from the conceptual core going together to form the infrastructure of the formal-structural system by which the software design theory is represented.

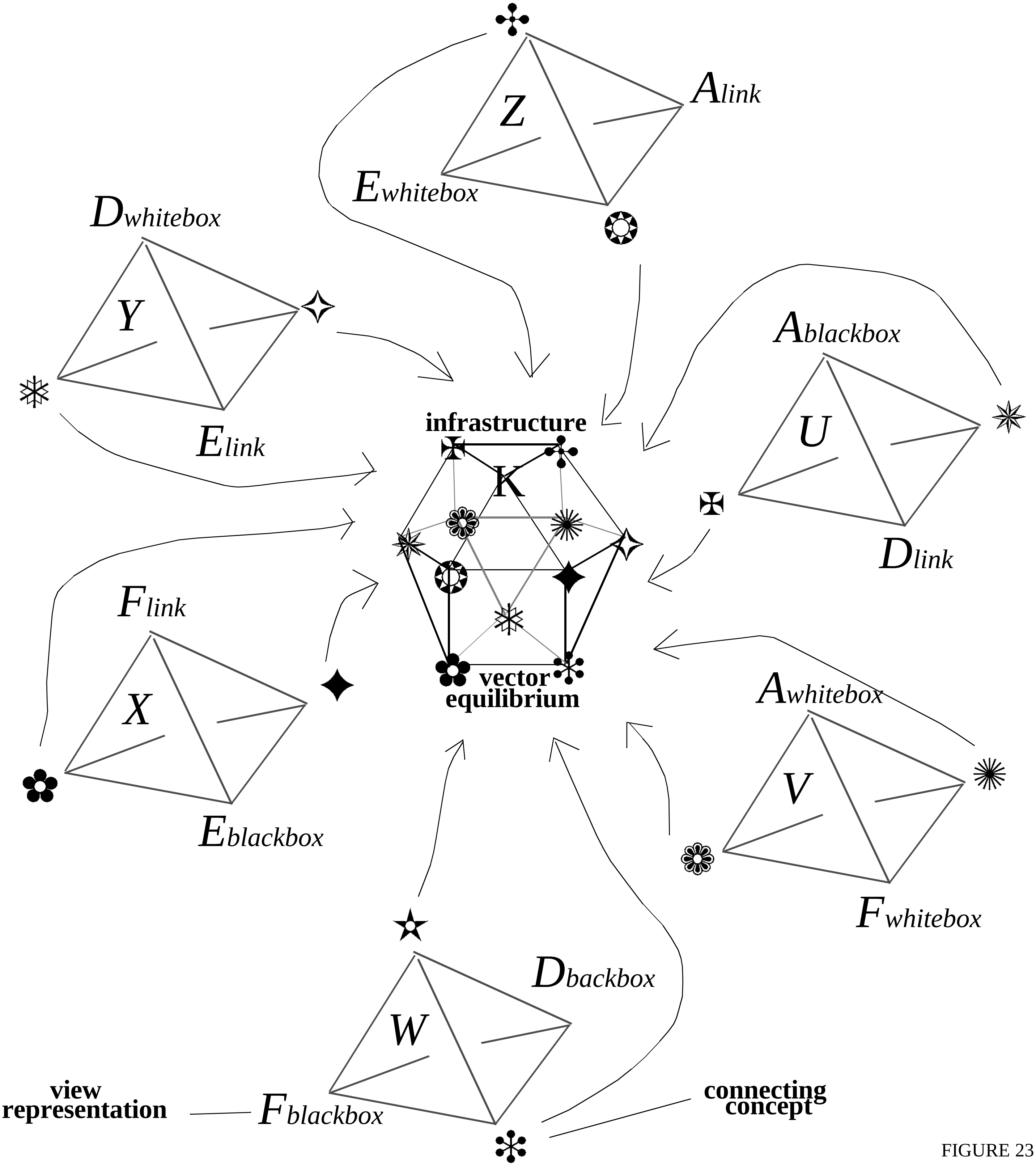

Alink
Z
Ewhitebox
Dwhitebox
Y
Elink
infrastructure
K
Ablackbox
U
Dlink
Flink
X
vector
equilibrium
Awhitebox
V
Eblackbox
Fwhitebox
Dbackbox
W
view
representation
Fblackbox
connecting
concept

FIGURE 23

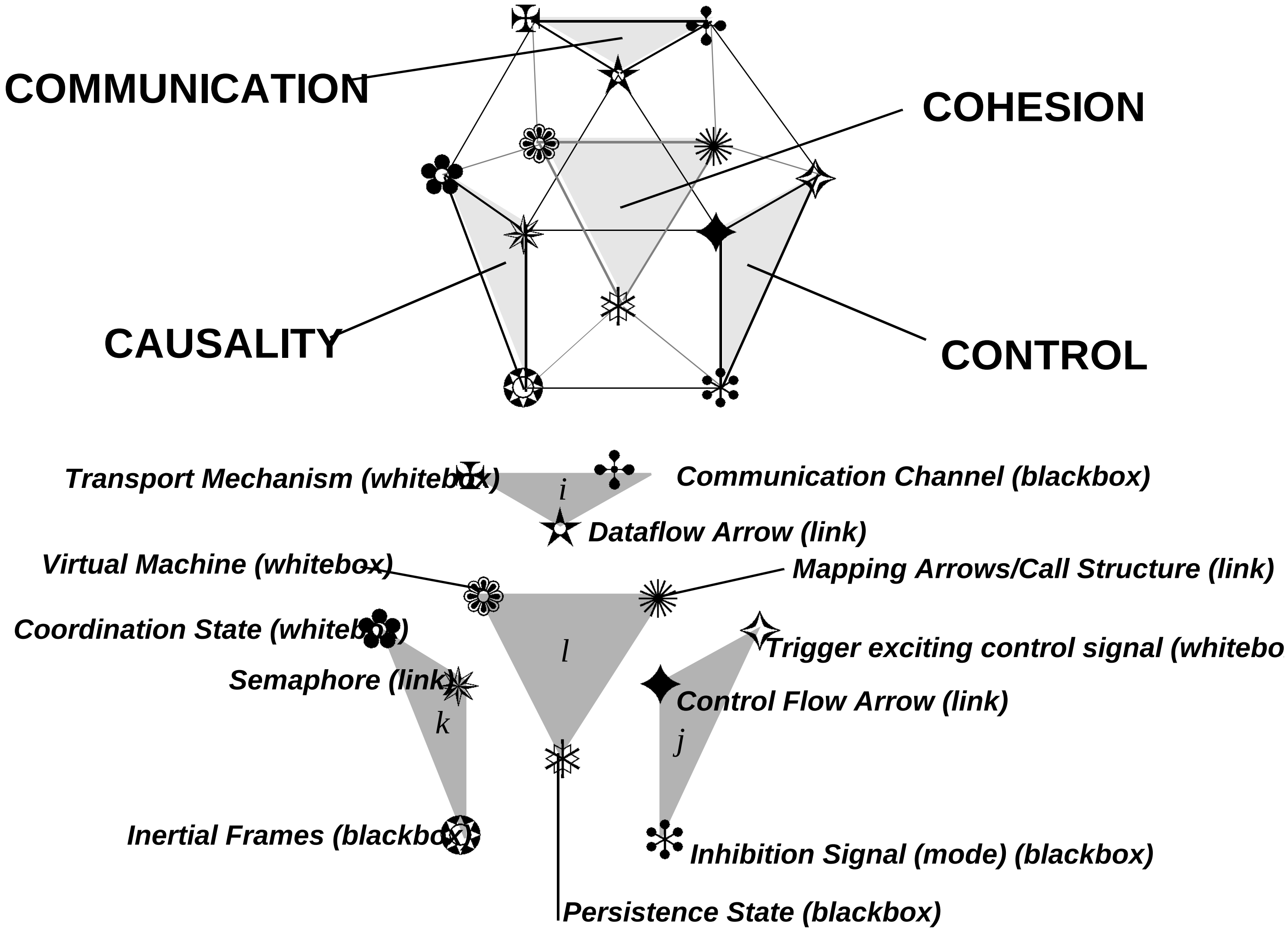

COMMUNICATION
COHESION
CAUSALITY
CONTROL
Transport Mechanism (whitebox)
Communication Channel (blackbox)
i
Dataflow Arrow (link)
Virtual Machine (whitebox)
Mapping Arrows/Call Structure (link)
Coordination State (whitebox)
l
Trigger exciting control signal (whitebo
Semaphore (link)
Control Flow Arrow (link)
k
j
Inertial Frames (blackbox)
Inhibition Signal (mode) (blackbox)
Persistence State (blackbox)

FIGURE 24

If each of these core-mediating concepts are related, using the same schema of as the representations of the perspectives (whitebox, link, blackbox), then analysis reveals four major complexes. These complexes may be related to several deep structural ideas with which all software systems must deal. These deep structural ideas are Communication(i), Control(j), Causality(k), and Cohesion(l). What is interesting is that each of these ideas concerns the relation of immanence to transcendence in some basic way. Control represents the transcendence of master over slave, and what are automata but pure slaves? Causality is a traditional way of expressing transcendencies. The cause is transcendent in time to the caused. Communication expresses our ability to transcend the barriers between entities. Cohesion is the expression of the transcendental unity of the system. This reduction two four basic transcendental themes at the infra-structural level serves to show that the description of the software theory as being rooted in Being[3] is accurate. This minimal system of deep structural ideas that are all representations of transcendence in various guises forms a core around the kernel of pure immanence that lies at the heart of the software design theory. It is this kernel of pure immanence that we must constantly deal in our software engineering practice. If we do not recognize what we are dealing with, it only makes the job more difficult. If instead, we learn to recognize the different ontological categories that effect our work and learn to use the work-arounds given to us by the formal-structural system, we can more easily understand the source of those difficulties. There is no 'silver bullet' but not because we are dealing with a werewolf. There is no silver bullet because we are dealing with our innermost relation to technology as 'controlling nature.' It is part of ourselves that we cannot see. It is a blind spot that grows darker the more we try to bring light there. We must learn to deal with it indirectly, and ultimately accept our own limitations.

science it seeks to understand the structural properties of general systems architectures, also on a small scale. The point is that software engineering uses this knowledge to build large-scale software systems. In these large--scale systems, quantitative changes in the dimensions of the systems cause qualitative changes in the nature of the systems. The facets of the software engineering discipline are responses to these unexpected large-scale effects. As general systems science explores larger and larger systems architectures, it becomes involved in building larger and larger software representations, and thus needs software engineering discipline. Also, as software engineering focuses more and more on design issues, it needs to know what general systems theory can tell it about general systems architectures.

The fact is, that software engineering draws knowledge from four disciplines. Besides general systems science and computer science, it also draws from management science and social science. Software engineering should emphasize it roots in social and management sciences more than it does at the present time. Software engineering concerns teams of people working together to produce the software system. It concerns the management of this development process which involves both general systems architecture and computer science knowledge, as well as application specific knowledge. By treating software engineering as if it arises directly from computer science alone, many of the key features of software engineering are distorted. Software engineering is seen as merely ‘programming in the large.’ Instead, software engineering is a mainly social phenomenon of human beings confronting the task of building large-scale software systems. In that task, there is a need for computer science specific knowledge and application specific knowledge. But it is the understanding of the general systems architecture which is key to making the whole system work properly. Beyond the general systems architecture, a large system is merely a large collection of small programs. This is why architectural design and the recently appearing myriad of software design methods are of interest. These methods are of interest not just because they provide the abstractions by which we can structure very large software systems. They are also of intrinsic interest because they provide the means of building very large architectural representations general systems science can experiment. Also, they are methods by which social and business systems can be described so that automated simulations of them can be constructed. They are a means of describing and controlling very large systems and their computer representations. Computer representations are a way of automating and controlling the described systems themselves. Software methods thus take on a key role of connecting social systems, business systems, and computer systems by describing at high levels of abstraction how these systems work in a way which can be automated. This allows a direct connection between the descriptions of systems, the simulation of systems, and the automation of systems, which has never before been possible.

Probably one of the most interesting aspects of software methods are their application to describing the software development process which is a social and managerial system. Traditionally these methods have been used to model the manual systems replaced by automations before modeling the changes that would be made in the system when it is automated. However, the modeling of the software production system itself shows the true versatility of these methods. What we are dealing with here is not something special to software engineering, but a general purpose system description language which can be used to describe any system. This description is the first step in the process of automation or simulation of that system. Since any object system could be described using software methods, they are a general purpose tool for understanding systems and mapping between system representations. For this reason, software engineering design methods have the same claim to generality that general systems science has. It is a universal discipline which can be applied to the description and modeling of any large-scale system. It is the means of creating mappings between specific images of systems in individual disciplines and their abstract counterparts. It is also the means of mapping between abstract representations of different kinds of systems. Finally, it is the means of mapping from an abstract representation of a system to its software simulation or architectural equivalent. Software engineering provides the glue that allows general systems science to create meaningful connections between systems and their representations and simulations. Software engineering has a much broader role in science than has been hitherto recognized. It is this broader role that causes it to have a structure that exemplifies many of the problems of scientific theory focused on by philosophy of science. Scientific theories are now reaching the stage where they are becoming complex enough that it is necessary to create computer representations of them. Scientific theories have structures that are merely specific instances of general systems architectures. Thus, scientific theories need to be engineered into specific systems architectures so that scientists can explore their ramifications through simulations and artificial experiments. In order to create these simulations, software engineering methods are necessary to translate from the structure and specific facts of the scientific theory into the software representation of those facts. In this way, software engineering methods will become more and more necessary to scientists as a means of structuring their representations of their theories and the simulations in software performed using those representations. Software engineering is a crucial component of future scientific explorations in many diverse fields. As simulations get larger and larger scientists themselves will start using the language of software methods to design the representations of their theories. Thus, it should not surprise us that many of the strange aspects of scientific theories are manifested in software theories. Software theories are the representations of scientific theories that can be tested in an artificial universe of a simulation. Software theories are the concrete representation of scientific theories in a dynamic form. This dynamic form is related to the nature of automated writing. Automated writing exhibits, as one of its side effects, the differing/deferring of differ*a*nce which is the advent of pure immanence within the field of the structural system.

This concept of the embedding of a singularity within the structural system which results in the fragmentation of perspectives was elaborated in the first part of this paper. This is a key idea which is not recognized by Klir in ASPS. This is because Klir sees himself as operating at the level of Being^{2} and does not explicitly recognize any higher metalevel of Being. Thus, Klir believes his representation of the structural system is purely present-at-hand, even though the purpose of creating a formal-structural system is to handle ready-to-hand phenomena such as discontinuities in the temporal unfolding of formal systems. If Klir's presentation of the formal-structural system had itself been at the level of the ready-to-hand modality, then it would have been more philosophical in nature. Alan Blum in his book Theorizing ,[11] and John O'Malley in Sociology of Meaning ,[12] have developed these kinds of self-conscious theoretical representations in sociology. They are more elegant because they say what they themselves do. It would be preferable to operate with these kinds of self-conscious representations in our explaination of software methods, but that would cause this text to become even more unreadable than it already is now to most readers. That kind of self-consciousness is intellectually more honest but leads to great difficulties in communication. Therefore, we will accept Klir's naivete and will practice that same simplified approach that assumes that the representation of the theory can be at a different metalevel than the action of the

[11] See Bib#.

[12] See Bib#.

theory. This is probably a false assumption.

Instead of attempting to make this presentation theoretically self-conscious in the way Blum or O’Malley have, we will posit the idea of the fragmentation of perspectives on the formal-structural system. This is equivalent to turning the formal-structural system inside out as demonstrated in the previous section of this essay. We will engage in working out the implications of this program as related to Klir’s representation of the formal-structural system. This intellectual trick will make it appear that we are still operating in present-at-hand mode in our theorizing, while in effect, we have represented both ready-to-hand and in-hand modalities in the more complex theory. The ready-to-hand is represented in the fragmentation of the software level of the formal-structural system, while the in-hand is represented by the hypothesized presence of a singularity hidden by the multiple perspectives. The alternative to this formulation is to enter into O’Malley’s universe of discourse which is posed within the ready-to-hand mode, and introduce a single extra theoretical element to represent the singularity. Another alternative would be to construct a theory which was selfconsciously operating within the in-hand modality. This would probably take the form of poetic dialectically cancelling aphorisms. Perhaps Frederich Nietzsche or Theodore Adorno were attempting this in their philosophical writings. We will not be so bold.

◆ ◆ ◆

The software engineering profession is composed of many roles. One of those roles is the software engineering technologist. The software engineering technologist has four basic realms of specialization. These realms concern software engineering *practices, processes, methods,* and *environments*. The software engineering technologist is first and foremost a practicing software engineer. But as a software engineer, his focus has been shifted from the product to the processes of software production that result in the product. This shift in focus is from the present-at-hand software artifact (the ends) to the ready-to-hand means of production. This realm of specialization has been assigned as the responsibility of the Software Engineering Process Group (SEPG) as defined by Watts Humphrey of SEI.[13] The

[13] See Bib#. See also SEPG Handbook Bib# & Bib#.

SEI believes by defining processes and then by measuring the outputs of these processes, we will be able to control and improve upon our software development processes. One means of controlling software processes is to proceduralize the entire development cycle. Certain software processes are crucial, such as requirements analysis and specification, or architectural and detailed design. For these crucial processes we may develop software methods to guide us which go beyond rote procedures. These methods provide detailed guidance in how to solve the problems to which a particular process is addressed. Software methods are very important. They provide a language for communication concerning the problem and solution spaces. They provide a means of capturing requirements or design concepts. Examples of some more prominent software methods are as follows:

o HATLEY / PIRBHAI [14]

Real-time Structured Analysis

o WARD / MELLOR [15]

Real-time Structured Analysis & Design

o SHALER / MELLOR [16]

Object Oriented Analysis & Recursive Design

[14] See Bib#.

[15] See Bib#.

[16] See Bib#.

o SPC / GOMAA ADARTS [17]

Ada Design Approach for Real-time Software

o NEILSON / SHUMATE [18]

Virtual Layered Machine /Object Oriented Design for Large Real-time Ada Systems

o CONSTANTINE / WASSERMAN [19]

Object-oriented Structured Design

The software engineering technologist must be conversant with as many of the major software methods as possible. Because these methods cover crucial software production processes, they represent the best expertise in how to solve the important problems in software development. Different methods may be appropriate for different software projects. The software technologist should be able to act as a consultant concerning the tailoring of methods to the needs of projects and also the selection of particular methods. Beyond this comes the necessity to train staff in the use of adopted methods. It is at this point that software systems meta-methodology becomes important. As it now stands, software methods, in general, are fairly new. Many are being proposed each year. Thus, the software practitioner, whose main concern is building

[17] See Bib#.

[18] See Bib#.

[19] See Bib#.

the software at hand, cannot be expected to keep up with the dizzying expansion of the field of software methods. The practitioner wants a tool box of techniques that will work for him. He wants to adopt a method which will facilitate the use of these techniques in a coherent way. The software technologist thus needs some overview of what is available so that he can help the practitioner select the appropriate techniques and methods. This overview can only be achieved by the development of software meta-methodology. Software meta-methodology gives a means of comparing methods with each other, and attempts to give an overview of the general field from which all software methods arise. Without a view of the field from which these methods arise, there is no way to adequately compare and discriminate between these myriad methods. Software meta-methodology is a theoretical discipline with very practical results. By establishing software systems meta-methodology, the software engineering technologist and the practitioner alike can establish the coverage of a particular method and its relation to other methods with different coverages. By coverage is meant how much of the field of possible software methods does a particular method cover.

The final realm of specialization for the software engineering technologist is enabling technology. The software engineering technologist should be part of a Software Engineering Technology Group which is concerned with enabling software processes and methods in a software engineering environment. This involves Computer Aided Software Engineering (CASE) and Computer Aided Cooperative Work (CACW). Implementing software engineering environments automates the tasks of software production. Up until recently, software production was a mostly manual process, except for editing and compilation. Now with many new software tools coming on the market, there is a great diversity of vendor-supported tools. These tools sometimes have their own unique methods associated with them, and also call for special software processes. Thus, building software engineering environments in the current environment is not trivial. It usually means integrating vendor tools with home grown tools within a single environment.

This particular essay grew out of a very practical situation concerning software methods and practice. Our Software Technology Evolution Project (STEP) had, in 1986, just implemented a simple software engineering environment on SUN workstations using the IDE[20] Software-through-Pictures™ CASE tool and Frame Technologies FrameMaker™ document processor. Upon completion of this project, I began working on a project that was using this environment. Immediately I noticed something was strange in the way our engineers were using the tools we had given them. I was appalled to see that they had introduced their own ad hoc method rather than following that method supported by the tool. At this point, they had little training in methods or in the use of the tool. However, I wondered why they had changed the Hatley/Pirbhai[21] method. As I began studying their modifications to the method, which were still in the range of what was allowed by the graphical editor but were not allowed by the design analysis tools, an interesting picture became clear. All of the changes made by the software engineers were

[20] IDE = Interactive Development Environments of San Francisco CA founded by Tony Wasserman.

[21] See Bib#

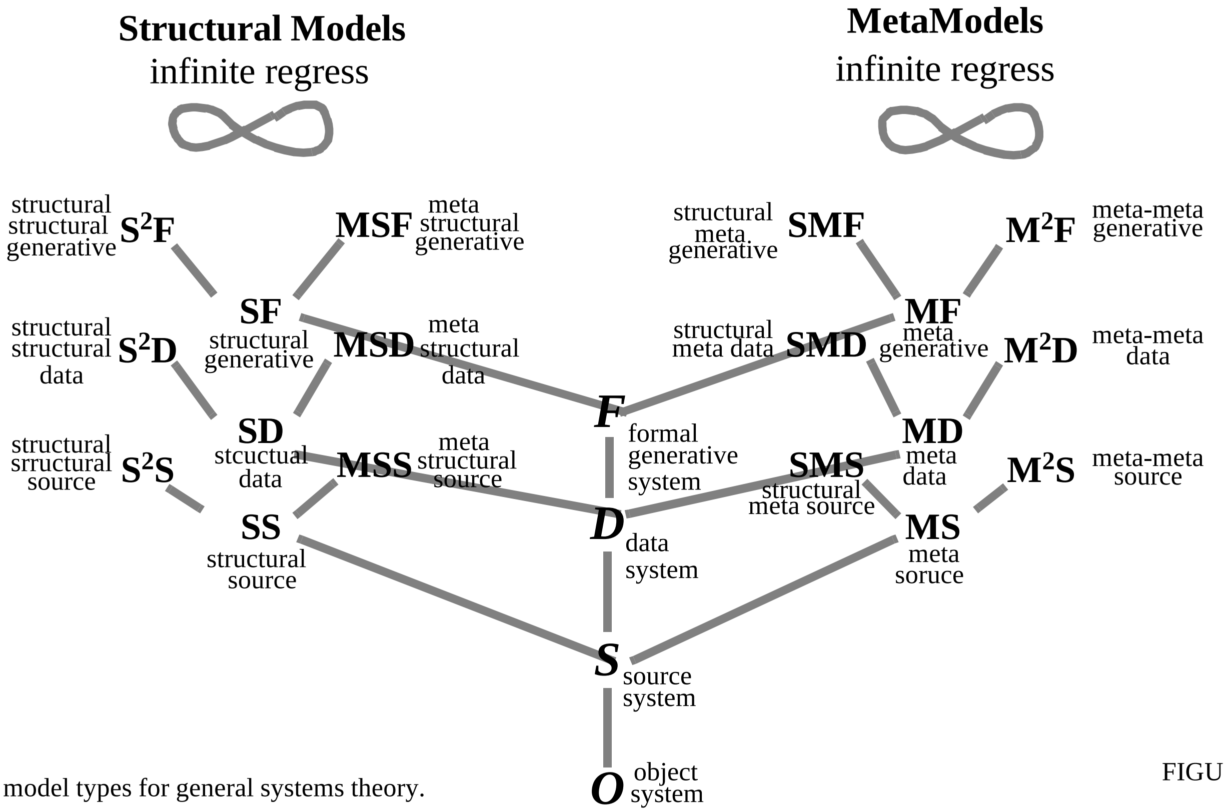
attempts to introduce tasking constructs into their design representations. On further study, it became clear that the popular tools all supported methods that did not express tasking. It took some research to find tools like DURRA by Mario Barbacci,[22] Task Builder™ by Ready Systems ,[23] and StateMate™ by iLogics[24] which did support some sort of tasking constructs. Since that time, IDE has introduced OOSD,[25] and SPC has introduced ADARTS.[26] Vincent Shen of MCC has also introduced RADDLE and its visual editor VERDI.[27]

What is interesting is that since most software methods were introduced to support information system development, the original methods, like Yourdon/DeMarco's Structued Analysis[28] and Yourdon/Constantine's Structured Design[29] did not deal with time. Hatley/Pribhai and Ward/Mellor introduced Real-time Structured Design at about the same time in which state diagrams were used to deal with the timing aspects of real-time systems. It was these methods which were implemented by

[22] See Bib#
[23] See ???
[24] See Bib#
[25] See Bib#
[26] See Bib#
[27] See Bib#
[28] See Bib#
[29] See ???

most of the early vendors producing commercial CASE tools. Yet these tools did not deal with one crucial problem in software engineering of real-time systems. That is the problem of tasking. Research showed that Gomaa had provided the most straightforward solution to this problem, and our shop began using DARTS diagrams produced in the IDE picture editor. Other solutions were too expensive, and this proved to be an excellent interim solution. The realization that software architectural design required more than just temporal ordering being added to the basic Yourdon/DeMarco/Constantine methods, made it necessary to think through the relation between these methods. Paul Ward[30] provided an insight into this relation between software methods when he observed that different methods ordered the same techniques in different sequences, and by doing so, produced very different views of the same system. The best example of this is the difference between the Hatley/Pirbhai and the Shaler/Mellor methods. These use the exact same representation techniques, but use them in different orders.

[30] See Bib#



Klir’s set of model types for general systems theory.

FIGURE 30

HATLEY/PIRBHAI
1) Dataflow Diagram
2) State Diagram
3) Entity Relation Diagram (optional)

SHALER/MELLOR
1) Entity Relation Diagram
2) State Diagram
3) Dataflow Diagram

By applying both of these methods to the same problem, I saw that the kind of system that would be designed using these two different methodological approaches was very different. I felt that this was an interesting phenomenon and wondered what would happen when one added the DARTS tasking structure method of Gomaa to the set of techniques to be used. The question was, where should it be added and what would occur if it were added at different points in both suggested sequences. These ruminations suggested that there was a field of knowledge here which was worthy of exploration in its own right. What was needed was a general approach to the whole field of software methods, a meta-methodological study, which would give guidance on how to connect techniques into method sequences in order to get full coverage of the software engineering design solution space. The answer seemed to be to introduce the concept of software systems meta-methodology along the lines that Klir suggested.

In order to get a clear picture of the field of software methods, a good starting point seemed to be the abstraction from each technique used by the major methods (Hatley/Pirbhai and Shaler/Mellor), and add that of Gomaa (DARTS). This led to the conceptualization of the four perspectives on software theory.

These developed through a process of further abstraction into what has been dubbed the four fundamental software engineering perspectives.

The Four Fundamental Software Engineering Perspectives

WHO => AGENT view \ W software

WHAT => FUNCTION view \ H design

WHEN => EVENT view / Y theory

WHERE => DATA view / ?(the idea)

With each of these perspectives is associated a minimal method that is used to represent that perspective by itself:

DATA viewpoint: Answers the question --Where is it in memory? Is modeled by entity-attribute-relation diagram. Data in a computer system is basically a manifestation of the configuration of memory locations and thus answers the question ‘where?’ Data is modeled in its pure state using Chen’s entity-relationship diagrams[31] extended from the

[31] See Bib#.

concept of Space as represented by computer memory.

FUNCTION viewpoint: Answers the question -- What is to be done? Is modeled by hierarchical functional decomposition. Function models what happens in a system in terms of data transformations. This is related to the operators in the formal language which define what can be done within the system. Extended from the concept of Grasping as Being^2 represented by the accumulator in the CPU, grasping allows manipulation which causes transformations in data which are called functions in Dataflow terminology. Function means transformation by the application of operators to operands.

EVENT viewpoint: Answers the question -- When is it to happen? Is modeled by timing diagrams that represent the temporal relations between signals, extended from the concept of Time as represented by the cycles of the CPU. It is also depicted through the use of interval and temporal logics.

AGENT viewpoint: Answers the question -- Who does it? Is modeled by tasking a hierarchy which shows nested independent threads of control within the system. The ‘actor’ paradigm developed by Hewitt[32] and Agha[33] is a development of this paradigm in a pure form, extended from the concept of Pointing as Being[1] as represented by the index registers in the CPU. Pointing is observing, so the agent is an independent automated ‘daemon’ or observer and actor.

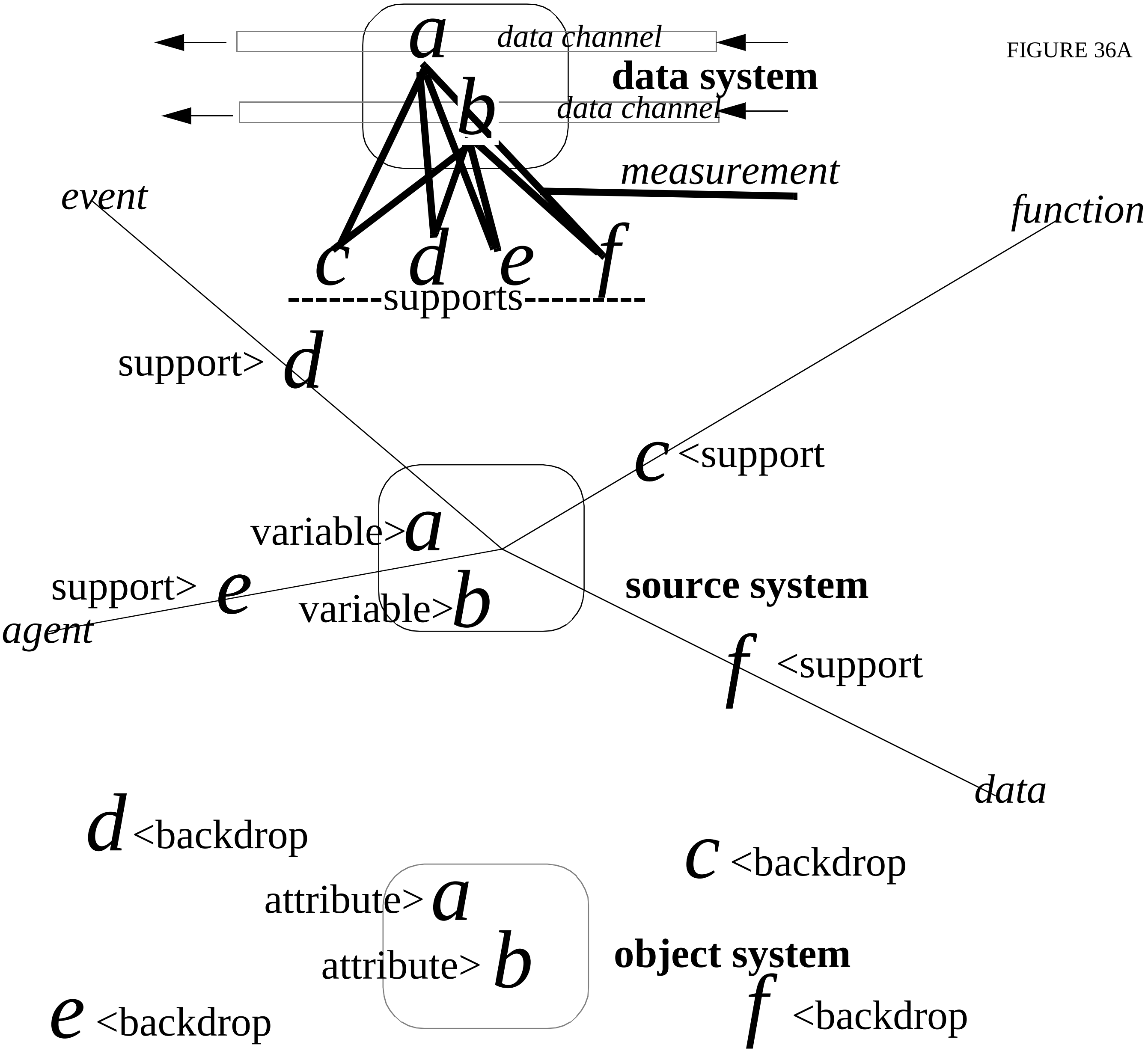
Later it was realized that these four perspectives could be seen as extensions from the ontological categories developed in the first part of this essay. But the main idea was that by taking these perspectives, it was possible to take fundamentally different views of the software system under development. Combining this with the concept of Software Theory developed by Peter Nauer, gave a means of understanding software representations as being tied to these perspectives. All of the representations produced by software methods attempt to approximate the software theory. The software theory is the <u>why</u> of the system under design. There are so many reasons why a system is

[32] See ???
[33] See Bib#

exactly the way it is that they could never all be written down. Instead of asking ‘why’ through representations, we ask questions of reduced scope like: Who? What? When? and Where? If one adds the assumption that only one perspective can be attained by the designer at a time, then one immediately has a reasonable answer to the question why software theory cannot ever be completely represented. One also immediately introduces the hiding place for the singularity which hides within the software formal-structural system.

Once the four perspectives have been delineated, it is possible to get a handle on the interrelation between software methods. The techniques like dataflow and finite automata, which are strung together in ordinary software methods, can be seen as minimal methods. A minimal method is a means of representing the software system from a single perspective or as a means of transitioning from one perspective to another. Perspectives are envisioned as looking at each other. Perspective *A* looks at Perspective *B* from its own viewpoint. This is what causes the minimal methods to be different. The minimal methods, when defined, tell us the absolute minimum conceptual

FIGURE 36A


a
data channel
data system
b
data channel
measurement
event
function
c
d
e
f
-------supports---------
support> d
c <support
variable> a
support> e
variable> b
agent
source system
f <support
data
d <backdrop
c <backdrop
attribute> a
attribute> b
object system
e <backdrop
f <backdrop

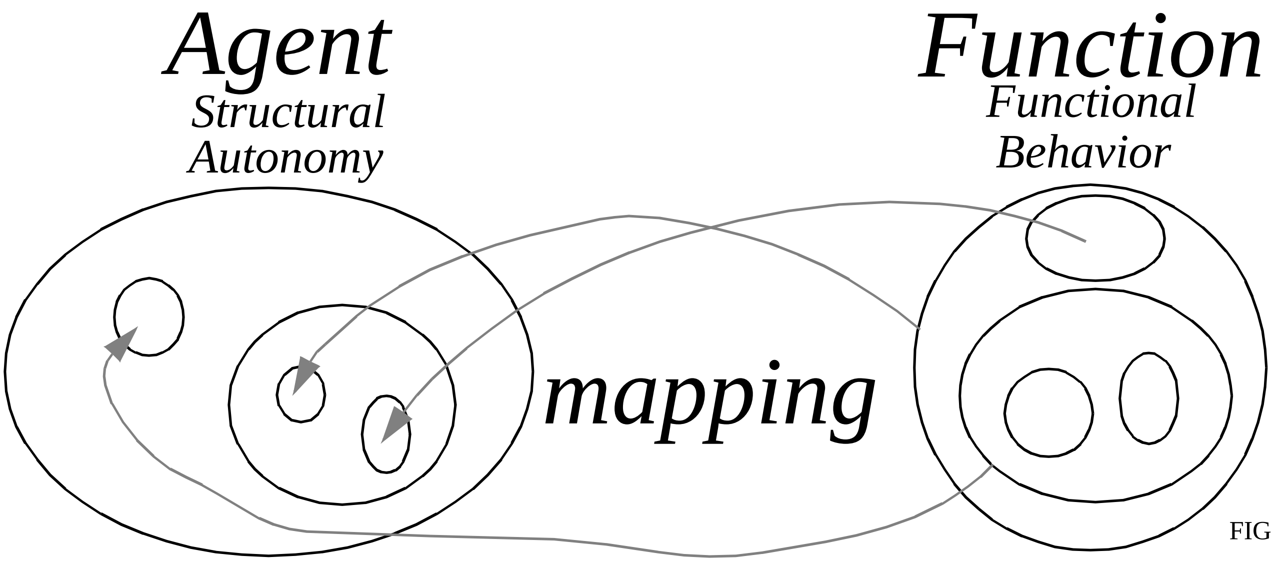


FIGURE 37A

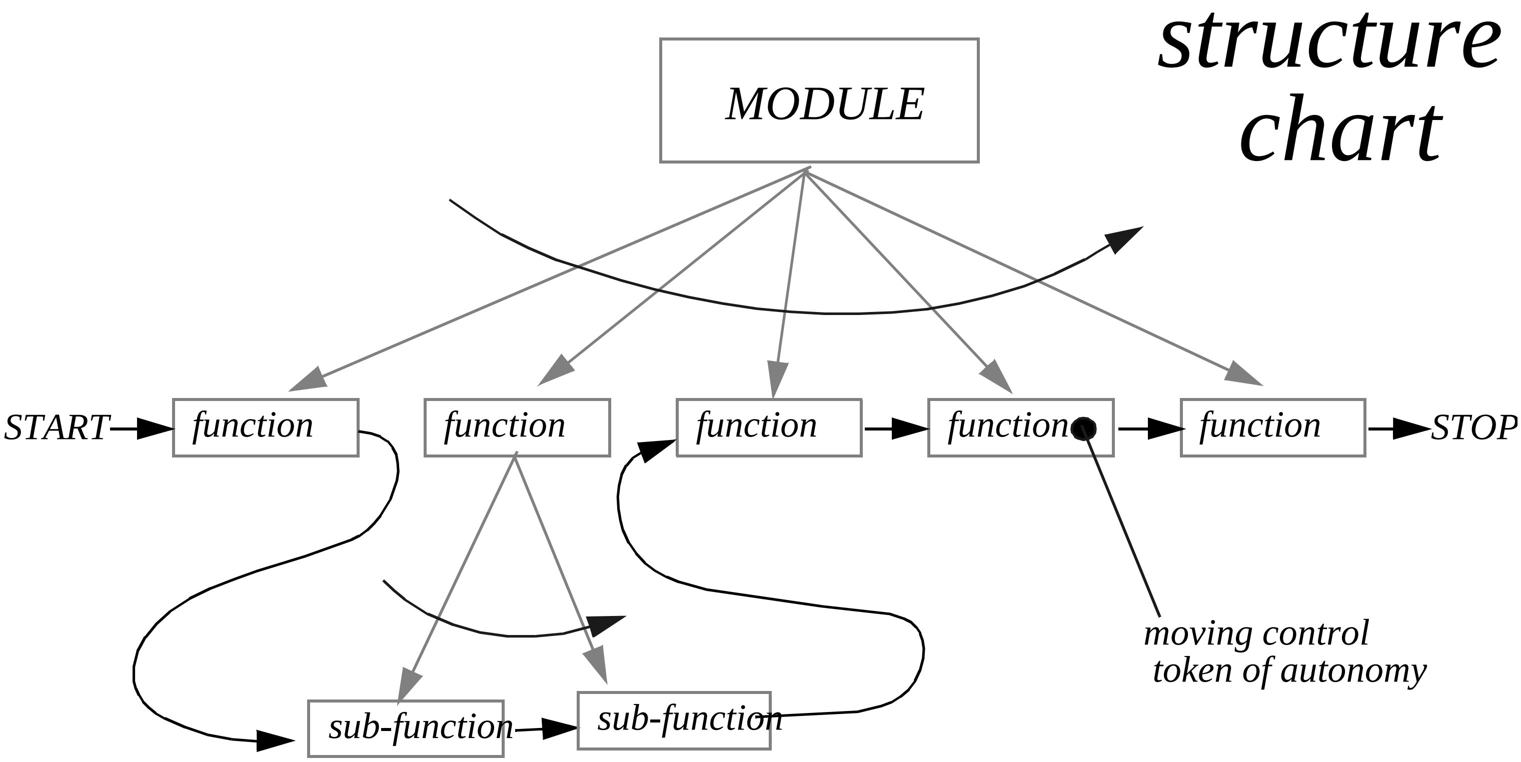


FIGURE 37B

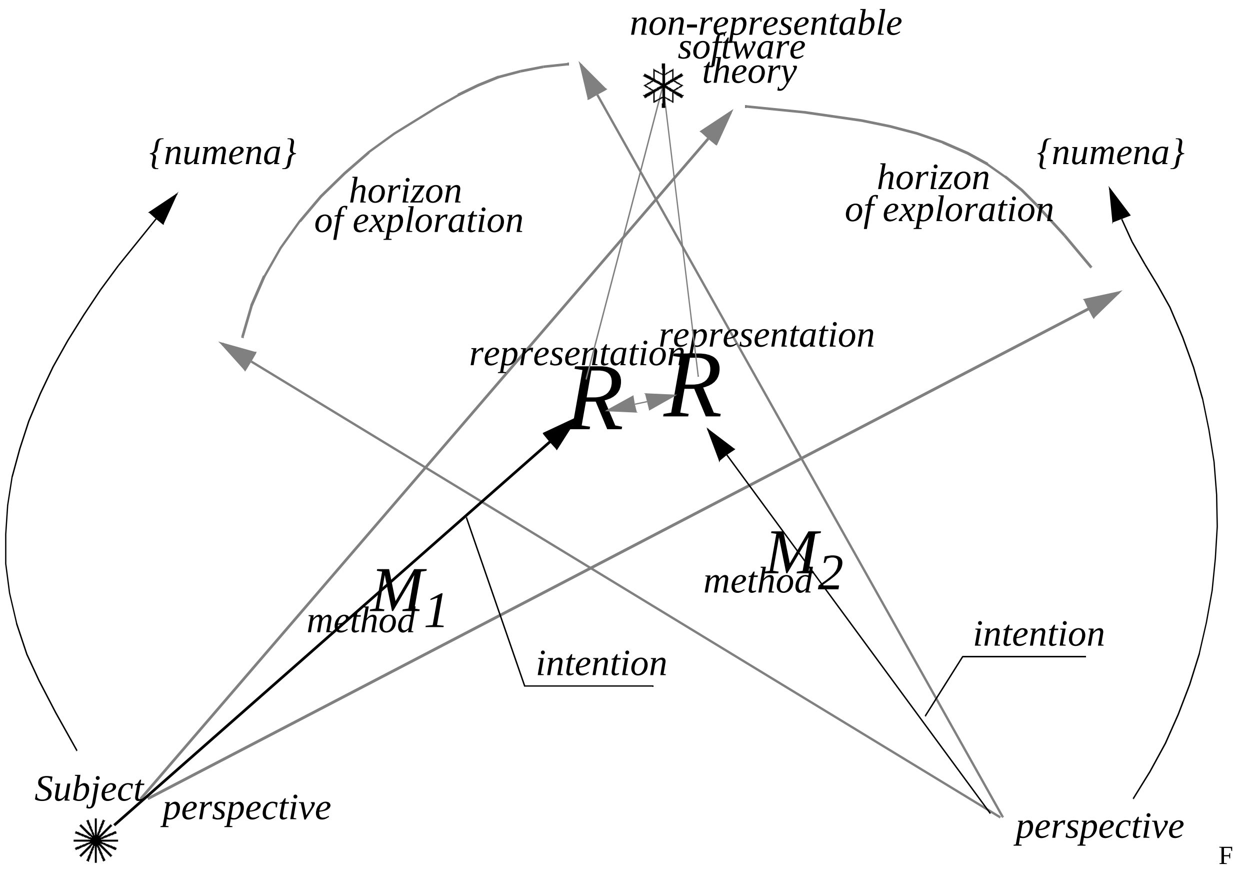

non-representable
software
theory
{numena}
{numena}
horizon
of exploration
horizon
of exploration
representation
representation
R
R
M_1
method
M_2
method
intention
intention
Subject
perspective
perspective

FIGURE 38

baggage needed to transition from one viewpoint to another. With the concept of minimal method we no longer need to take the methods given us for granted. We can analyze them to see what is absolutely necessary in our methods. Finally, we achieve a means of analyzing proposed methods by relating them to the field of minimal methods as a whole.

This leads to another interesting point. When the dataflow diagram technique is analyzed, it becomes apparent that it is really composed of two minimal methods -- one addressing data flowing between functions, and the other addressing the transformation of data in stores by functions. Thus, we see in the dataflow a merger of two minimal methods. Data looks at Function as transforming methods, while Function looks at Data as flowing streams between functions. The great success of the dataflow diagram may be attributed to the fact that it represents a two-way bridge between the Function and Data viewpoints. We see in this method an excellent example of how methods should be constructed to represent, in the simplest manner possible, the means of transitioning back and forth between design perspectives.

This kind of study leads inevitably to an attempt to isolate all the minimal methods connecting the different design perspectives. This is not a simple task. A first cut was made in an earlier paper given at CASE88 workshop.[34] The version presented here is a further refinement. It is necessary to state that this version is still tentative. The process of attempting to isolate the minimal methods connecting all the design perspectives has demonstrated the ad hoc nature of the methods designers are using. Singling out those techniques that can serve as good one-way bridges between perspectives, is a matter of intuition and analysis of the relations between existing techniques used by different popular methods. This kind of analysis is necessary in order to create a concept of the field of design methods. The field connects all the minimal methods via the fundamental design perspectives.

Viewpoint Pairs and Minimal Bridging Methods

FUNCTION > AGENT
Minimal Allocation / Mapping of Function to Task

Looking at Agents from the Function's

[34] See Bib#

point of view. Function views tasks as vehicles for system functionality. This method establishes the mapping from function to task, attempting to establish the logical coherence of the tasks by grouping logically-related functions as much as possible, given performance requirements. See Mellor and Ward, Vol. III of Structured Development for Real Time Systems[35] for an explanation of how Functional Allocation is done.

AGENT > FUNCTION
Minimal Virtual Machine

Looking at Function from the Agent's point of view. Agent views Function as Virtual Machine Instructions performed. This method attempts to establish the space/time coherence of the logical functions within the tasks. Space/time coherence allows the system of tasks to function within the performance specifications. Space/time coherence may conflict with logical Coherence. Space/time coherence is established through the construction of a

[35] See Bib#

virtual machine which executes instructions whose outcome displays the required system functionality. See Neilson and Shumate's description of Object-Oriented Design/Virtual Layered Machine Methodology in Designing Large Real-Time Systems with Ada[36] for an explanation of how to build virtual machines.

FUNCTION > EVENT

Minimal State Transition / Function Activation Table

[36] See Bib#

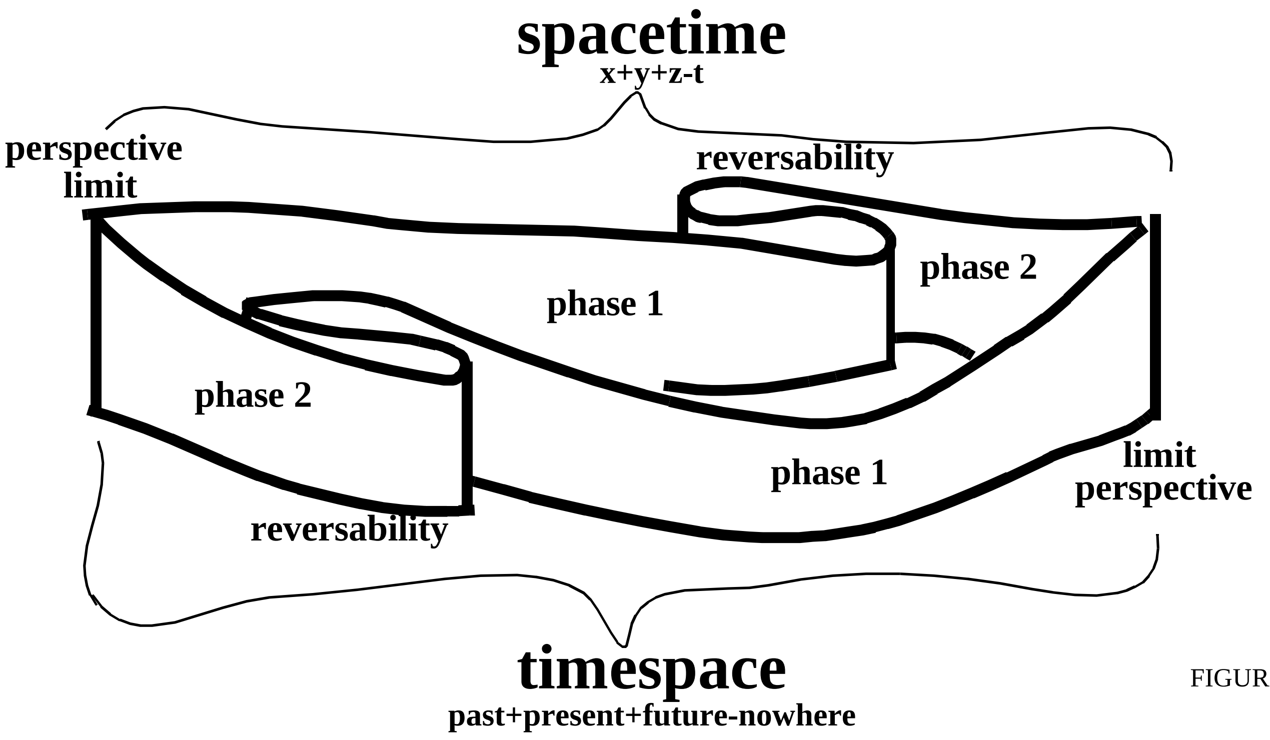


FIGURE 41

Looking at Event from the Function's point of view. Function views Events as state transitions. State Transition Diagrams activate functions on specific event occurrences given the current state of the system. See Hatley and Pirbhai or Ward and Mellor or Shaler

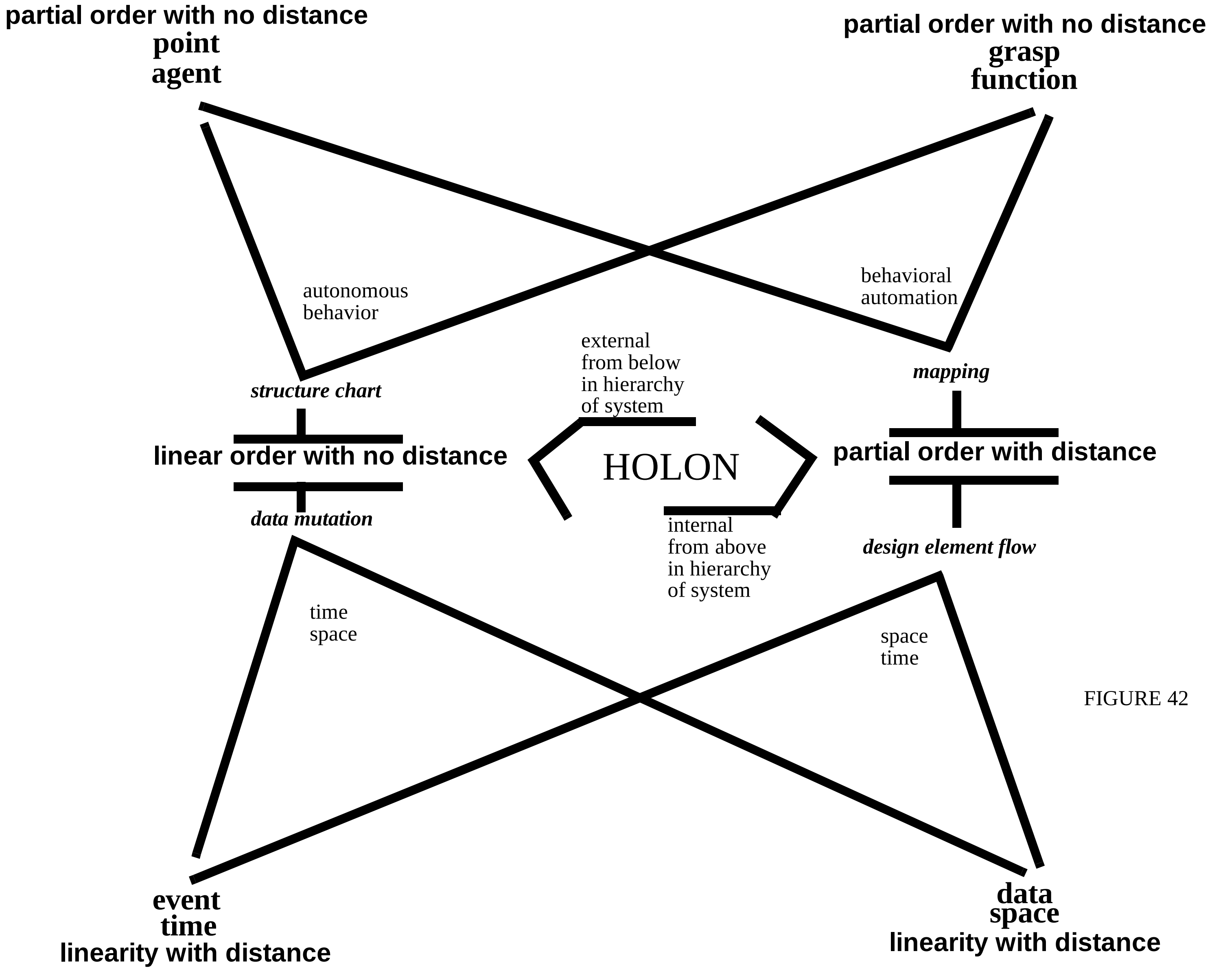


FIGURE 42

and Mellor(Project Technology) for the use of state machines in real-time analysis and design.

EVENT > FUNCTION

Minimal Petri Net

Looking at Function from the Event's point of view. Function views Events as Petri Net representations of the flow of control in the execution of virtual machine instructions. Petri Nets establish the firing order of functions. Functions are considered events which move markers through the petri network. See W. Resig Petri Nets: An Introduction.[37]

FUNCTION > DATA

Minimal Input/Output on Dataflow

Looking at Data form the Function's point of view. Function views Data as dataflow. Function sees data in terms of inputs and outputs. The dataflow diagram represents this method very well. See DeMarco Structured Analysis and System Specification[38] for ex-

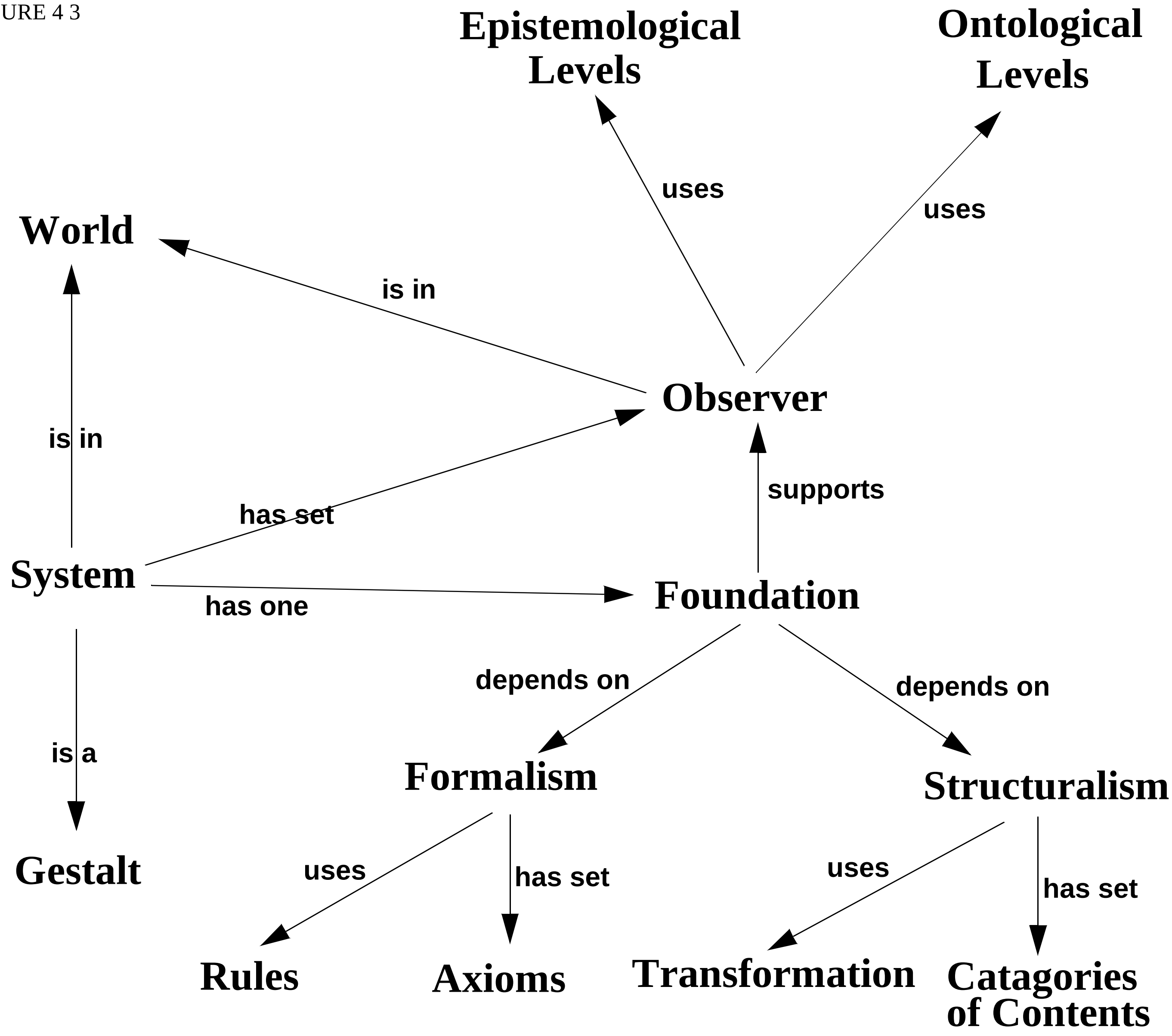
[37] See Bib#

[38] See Bib#

planation of the dataflow diagram.

DATA > FUNCTION
Minimal Data Objects (Encapsulation, Access Methods)
Looking at Function from the Data’s

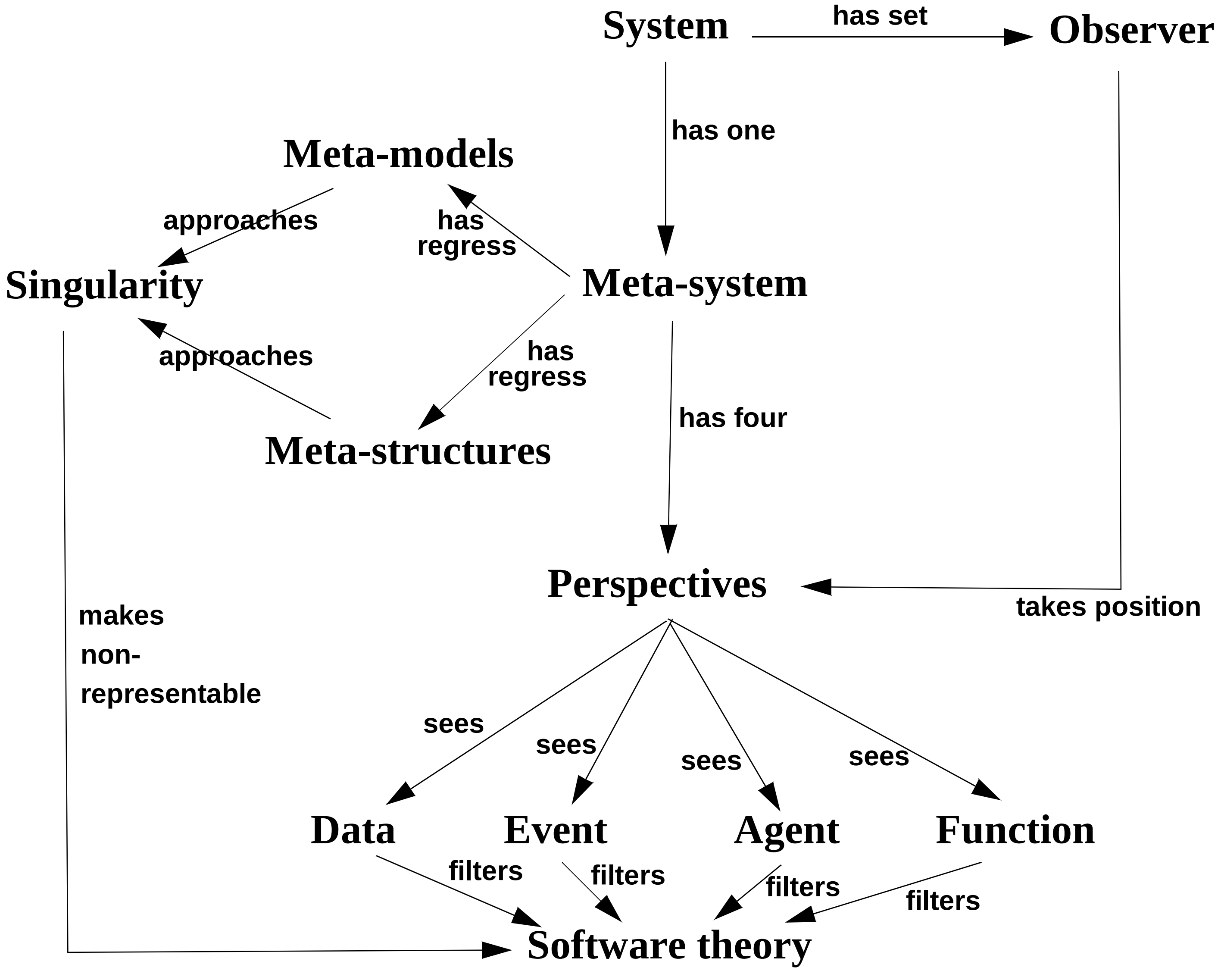
FIGURE 4 3



point of view. Data views Function as a transforming 'method' operating on Data. Stress on Data Encapsulation, such as that by Guttag and Liskov,[39]

[39] See Bib#

FIGURE 4 4



provides a method of organizing functions with respect to data access.

AGENT > EVENT
Minimal World-line

Looking at the Event from the Agent’s point of view. Agent views Event as a Worldline (i.e. a track through space-time). Following one agent through time and watching what messages arrive in what order, can be represented by a worldline diagram. See Agha on Actors[40] for an explanation of the use of this method.

EVENT > AGENT
Minimal Scenarios

Looking at the Agent from the Event’s point of view. Event views Agents as Scenarios. Scenarios show the sequence of agent’s actions triggered by a given event. It follows the flow of agent’s actions given a specific initial situation. Scenarios are the normal means of doing mental simulations to test a design. This is a standard method which is used in software engineer-

[40] See Bib#

ing but little documented in the literature.

AGENT > DATA

Minimal Transport Mechanisms

Looking at the Data form the Agent's point of view. Agent views Data as data transport mechanisms such as

FIGURE 4 5

Meta-system

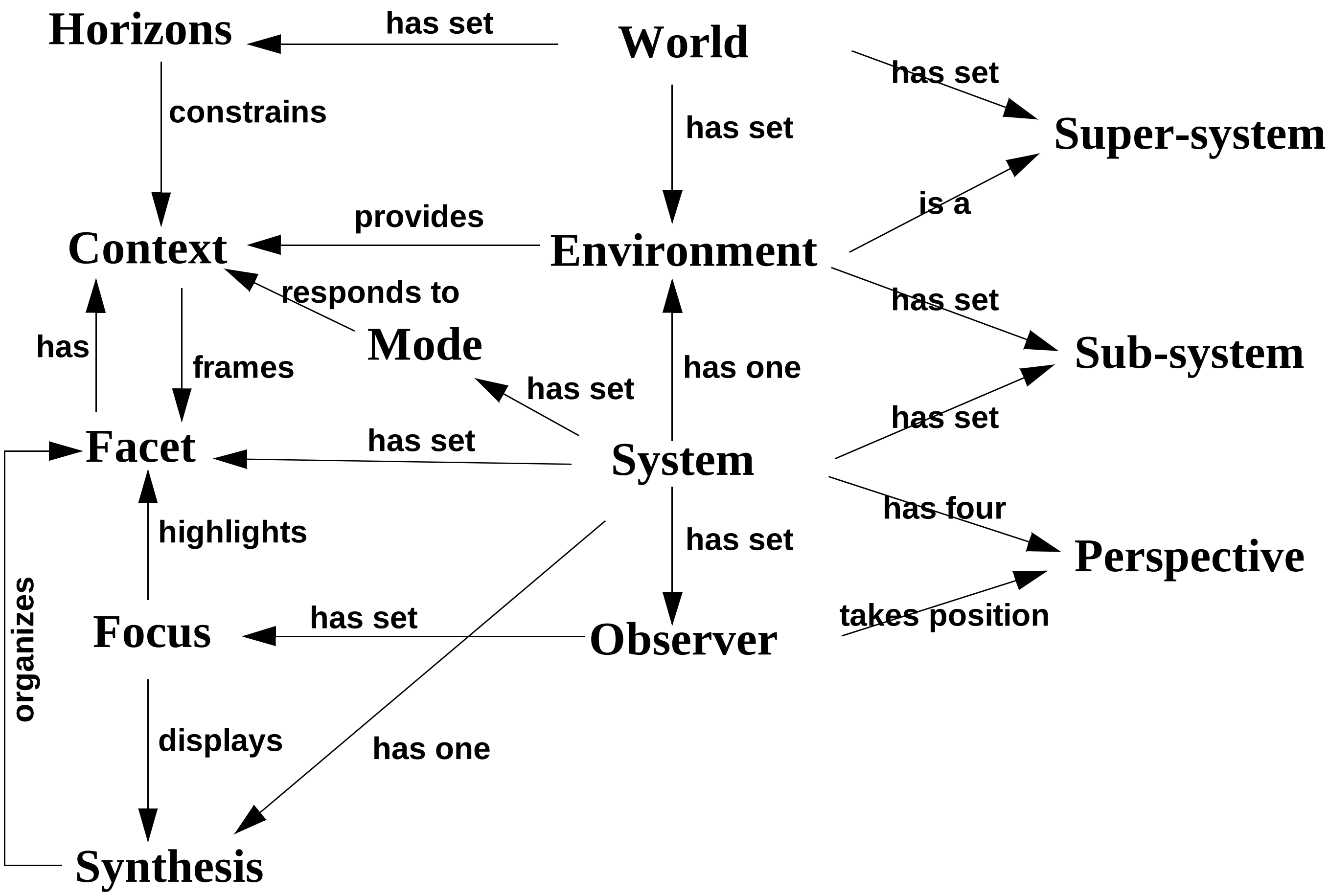
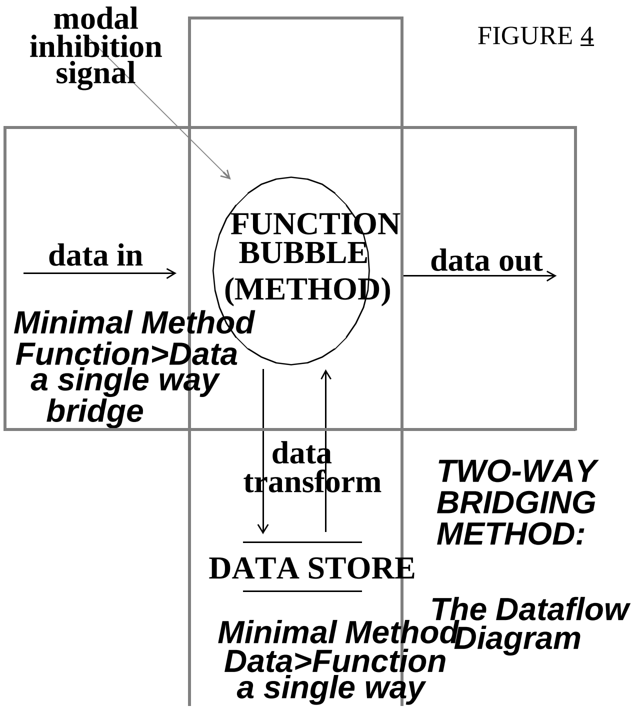
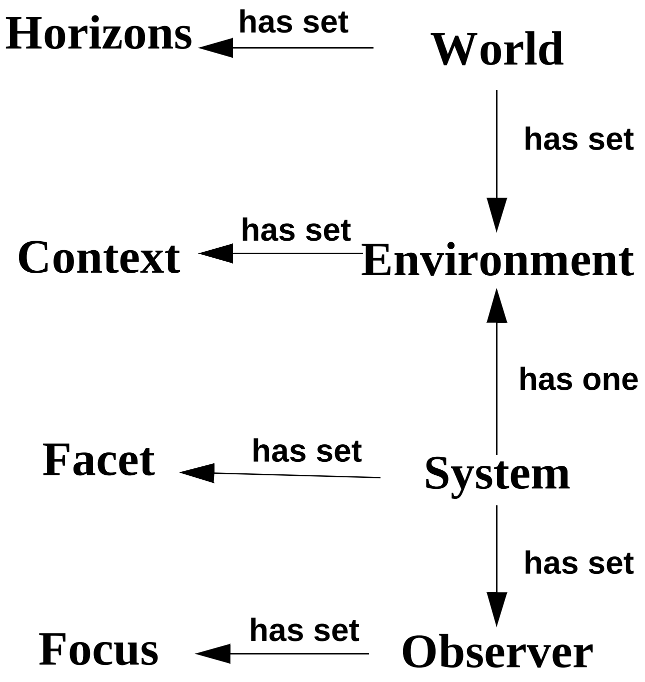


FIGURE 4



agent	function	event	data
peripheral	external	interupt	datum
network	context bubble	sheaf	class
processor	transform	signal	entity
task	function	event	instance

FIGURE 4 6



Queues, Mailboxes, Semaphores, Pipes, Etc. Agent looks at data in terms of the transport mechanisms that which allow communication between concurrent tasks. See Gomma "Software Development of Real-time Systems"[41] for an explanation of a design method that displays transport mechanisms.

DATA > AGENT
Minimal Data Structures

[41] See Bib#

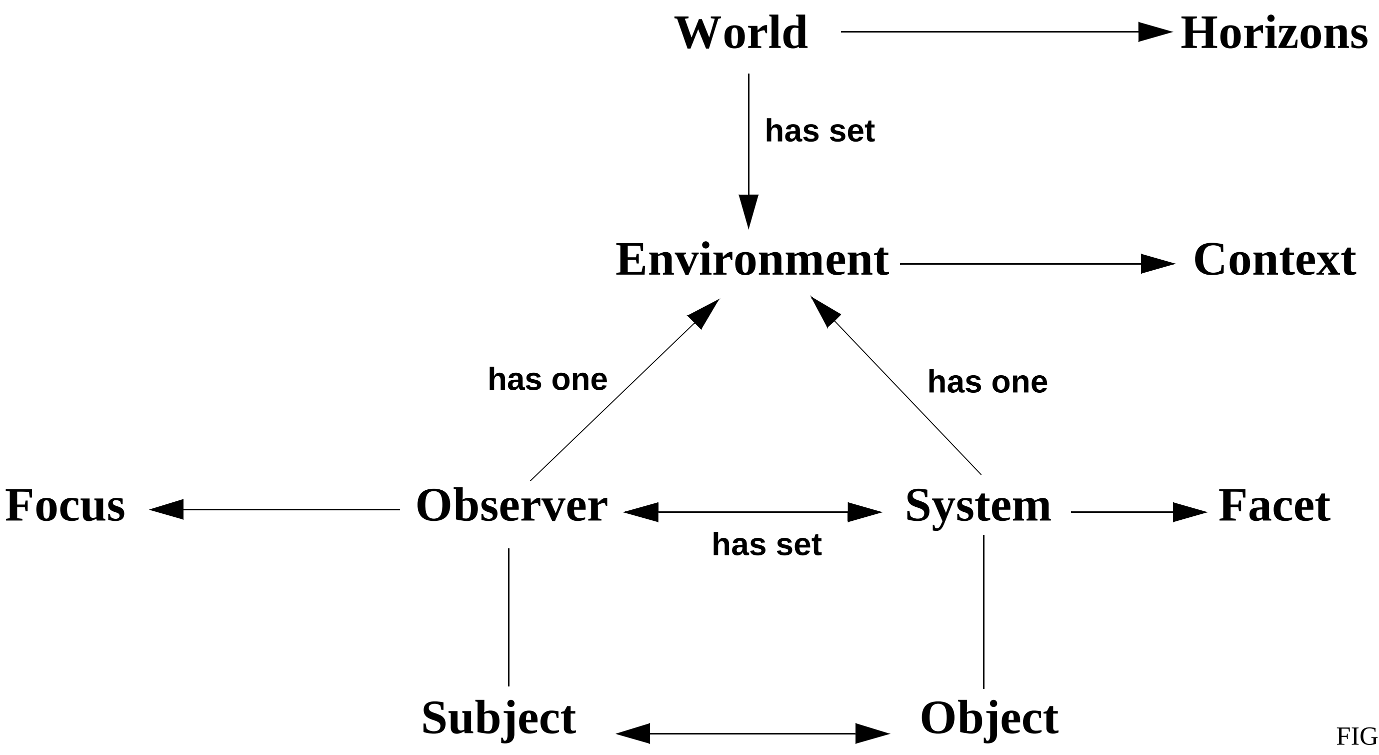


FIGURE 4 7

Looking at the Agent from the Data's point of view. Data views Agent via a Data Monitor. The data structures used by agents to support their activities are represented by this method. Normally they are guarded by semaphores or Hoare's "Monitors."[42]

EVENT > DATA

Minimal Design Element Flow

Looking at the Data from the Event's point of view. Event views Data as flowing design elements. Event sees data as persistent. The most persistent elements of a system are the design elements that are operands created in the design process. Normally, these are considered a static clock work mechanism. However, in real time systems, they can usefully be considered as flowing through the states of the system with a life cycle of their own.

DATA > EVENT

Minimal Data Transitions

Looking at the Event from the Data's

[42] See Bib#

> Point of view. Data views Events as changes in data values. Looking at printouts of a set of data elements as a program executes in the oldest form of checking out whether programs are working. Events are represented as state variables to the system. Data sees changes as changes in Data values.

Each minimal bridging method will only work in one direction, but it may combine with its opposite bridging method to provide a two-way method bridge. An example of this is the dataflow diagram which combines the function to data and data to function bridges.

Others might derive a different set of minimal bridging methods because different criteria may be applied in the selection process. My major criteria was the need to get broad coverage of the space of possible different methods while representing all the currently known genuinely useful techniques. Thus, practicality is an important aspect of selecting and analyzing the relation between minimal methods. It is important that a designer be able to look at this as a map of the methodological

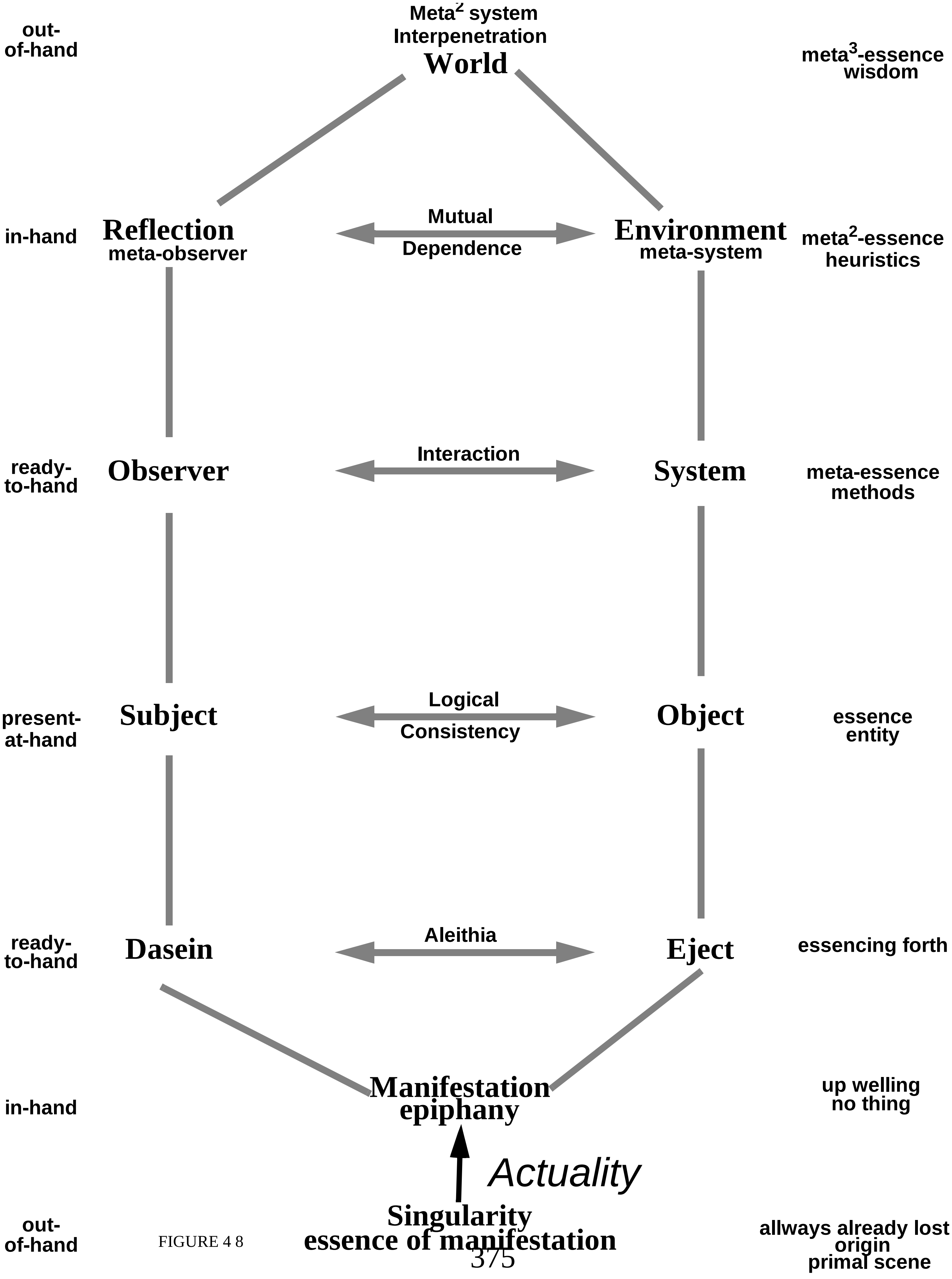

out-
of-hand
Meta2 system
Interpenetration
World
meta3-essence
wisdom
in-hand
Reflection
meta-observer
Mutual
Dependence
Environment
meta-system
meta2-essence
heuristics
ready-
to-hand
Observer
Interaction
System
meta-essence
methods
present-
at-hand
Subject
Logical
Consistency
Object
essence
entity
ready-
to-hand
Dasein
Aleithia
Eject
essencing forth
in-hand
Manifestation
epiphany
up welling
no thing
Actuality
out-
of-hand
Singularity
essence of manifestation
allways already lost
origin
primal scene

FIGURE 4 8

territory. Yet, it is also important that each method represents a completely different 'orthogonal' aspect of the software system while at the same time being able to work together with the other methods to create two-way bridges and larger method sequences resulting from moving in a certain order through the design perspectives. This set of minimal methods will serve as a working hypothesis until it becomes obvious that it is no longer the best selection of minimal methods. By looking at the techniques that have been chosen, it will become clear what this set of minimal methods seeks to achieve: broad coverage, orthogonal independence, practical usefulness, and systematic coherence.

Since this set of minimal methods results basically from an empirical selection of existing techniques, there is little surprise that the complete set has some unusual features. These techniques were not developed in any systematic way. They have been invented by different practitioners for many different purposes in various circumstances. To make an assemblage of these techniques, after the fact, and claim unity for them, would seem to be ridiculous. However, taken as a

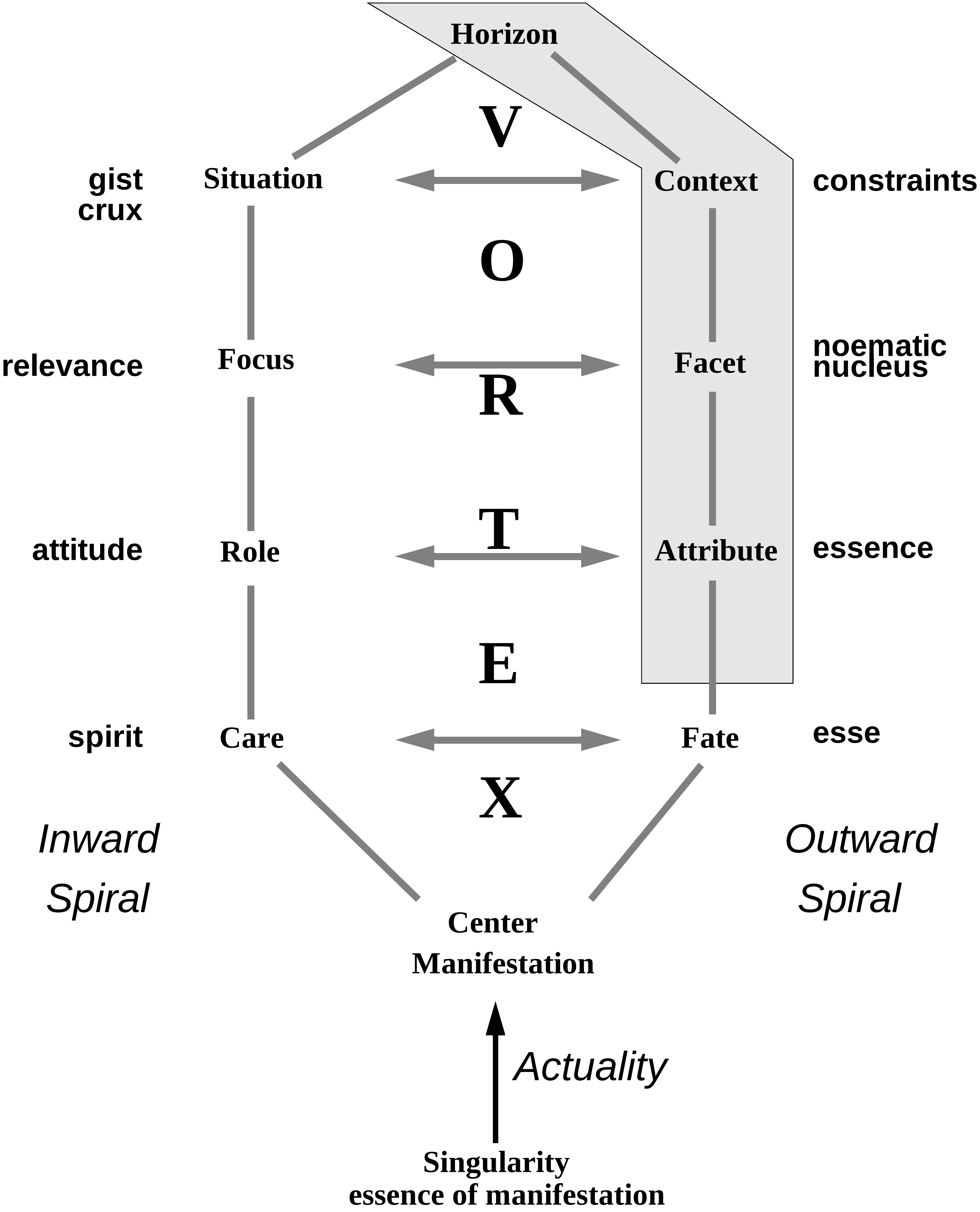

Horizon
V
gist
crux
Situation
Context
constraints
O
relevance
Focus
Facet
noematic
nucleus
R
T
attitude
Role
Attribute
essence
E
spirit
Care
Fate
esse
X
Inward
Spiral
Outward
Spiral
Center
Manifestation
Actuality
Singularity
essence of manifestation

FIGURE 4 9

representative of what a software metamethodological field might look like, it is possible to test it to see what its shortfalls might be so that the next attempt to construct a field might be more successful. Analysis of the field of minimal methods reveals some strange properties.

o Not all the method bridges are two way.

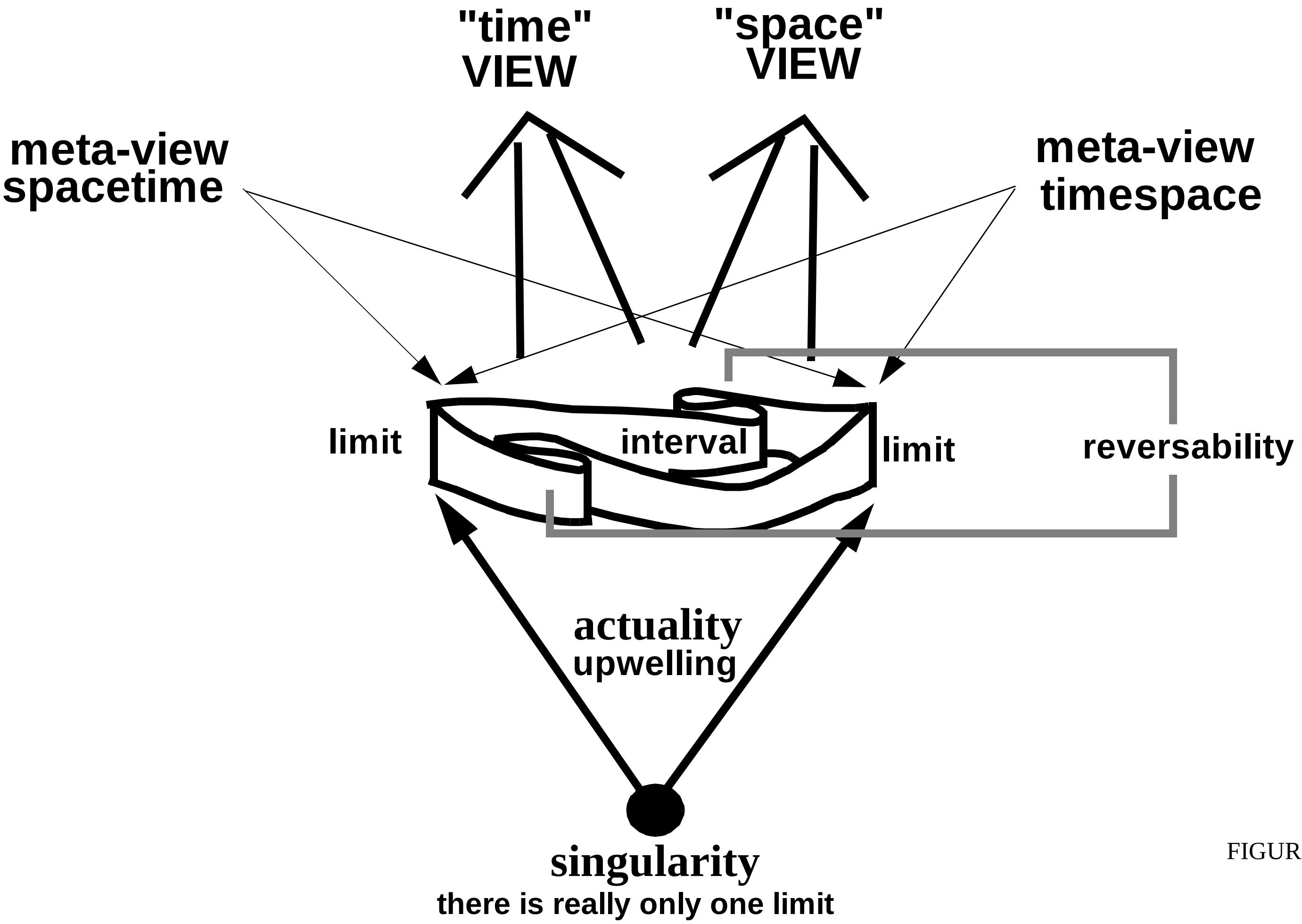


FIGURE 5 0

o The whole field is lopsided with function-task bridge being very complex while event-data is very simple.

o The elements of different minimal methods do seem to work together to a certain extent, but it is difficult to see an overall pattern.

o Abstract analysis reveals asymmetries in the relation between the elements of minimal systems.

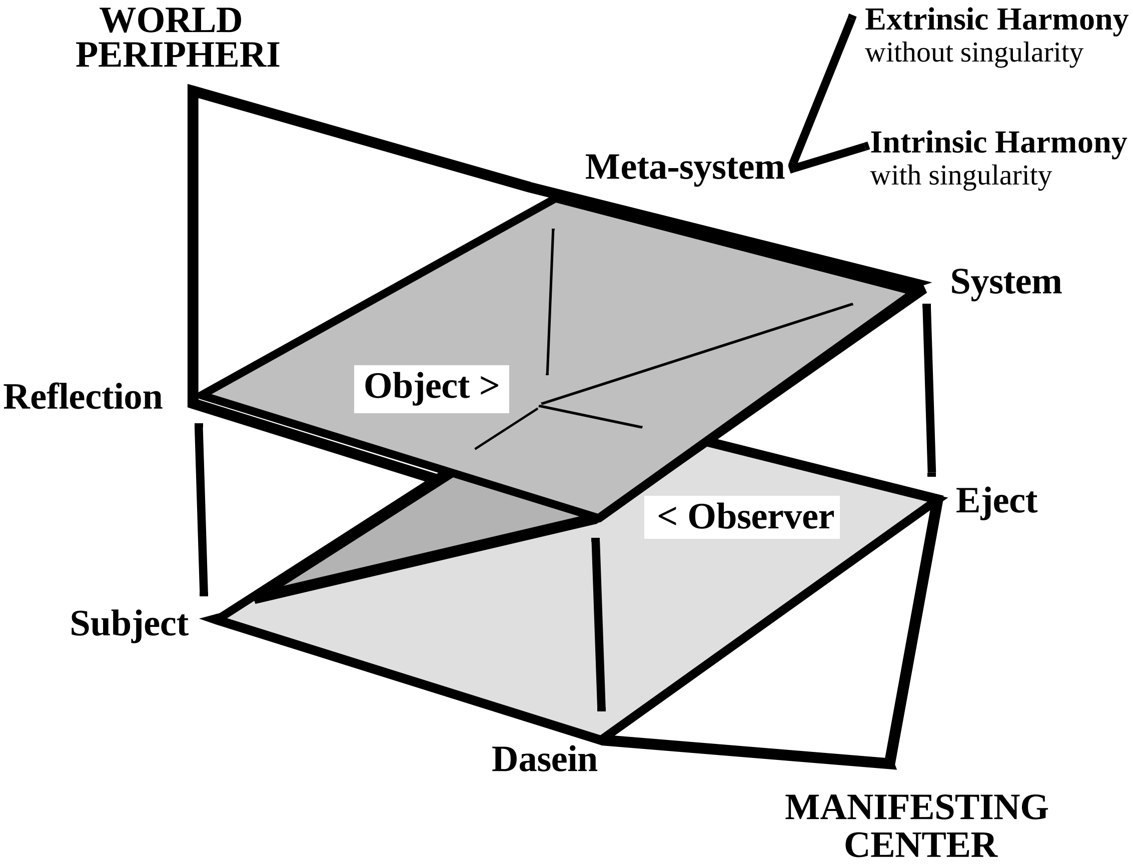


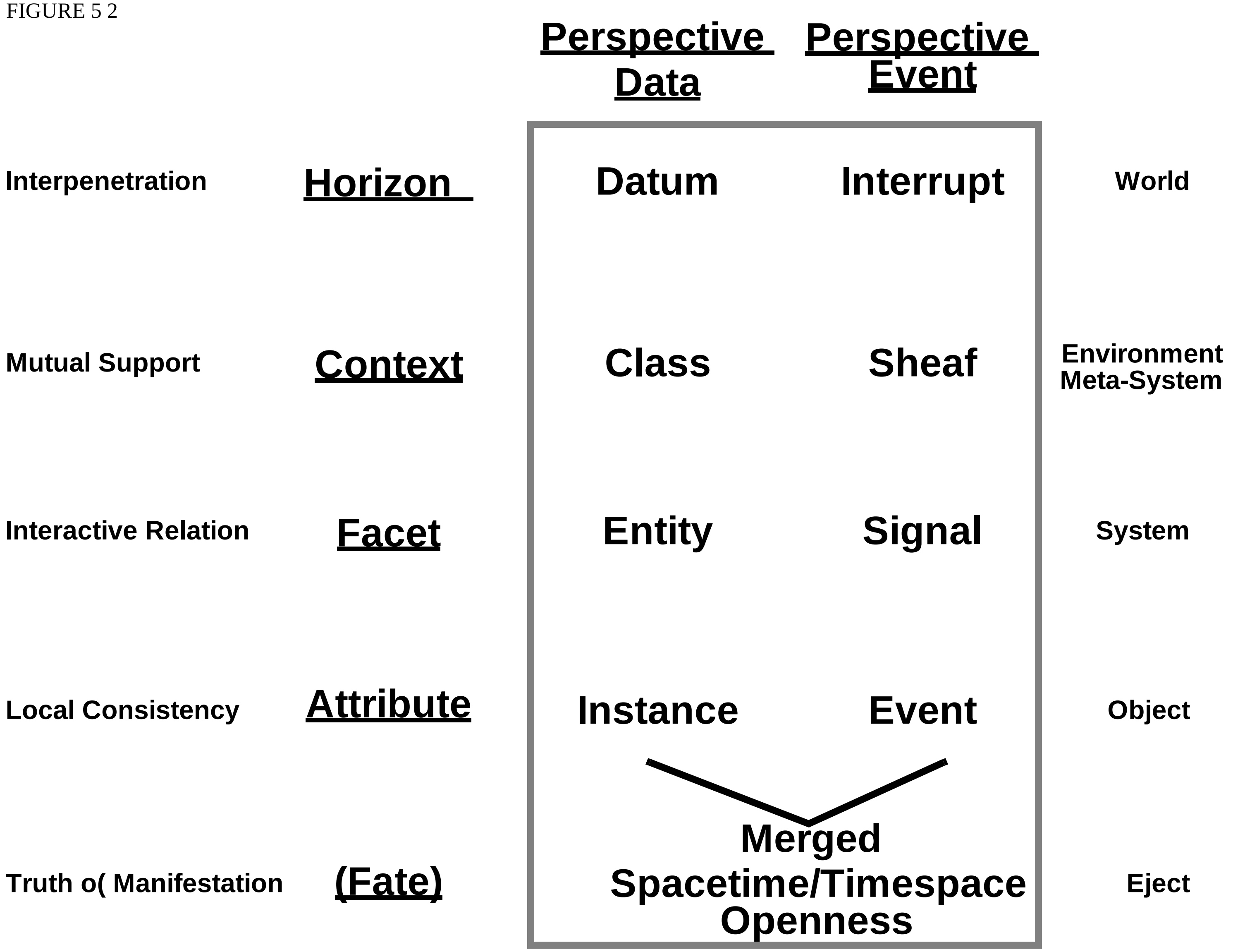
FIGURE 5 1

These properties probably flow from the ad hoc and empirical nature of the elements combined, but let us suspend judgement and treat this set of minimal methods as a united field. That unified field represents the full extent of necessary fundamental software methods. Any methodology must build upon these twelve minimal methods, then string them together by taking a particular path between the four perspectives in order to design a software system. Accepting this hypothesis, the rest of this essay will attempt to understand some of the idiosyncratic attributes and the non-symmetrical relations between the various minimal methods as they are presented here. In the process, many interesting features of the minimal methods will come to light. In fact, it will become obvious that these features are highly significant and form an overall pattern which allows us to gain a deep insight into the nature of the methodological field. It will become apparent that the methods, although chosen from an empirical field of currently used and available methods, actually form a systematic field which has its own internal logic. This internal logic makes the fundamental differences between the points of view on software design clear.

◆ ◆ ◆

What follows will be a graphical representation of the method bridges between software perspectives. Each bridge will be reduced to its idealized ‘conceptual core,’ and a brief summary of the sources of the methodology

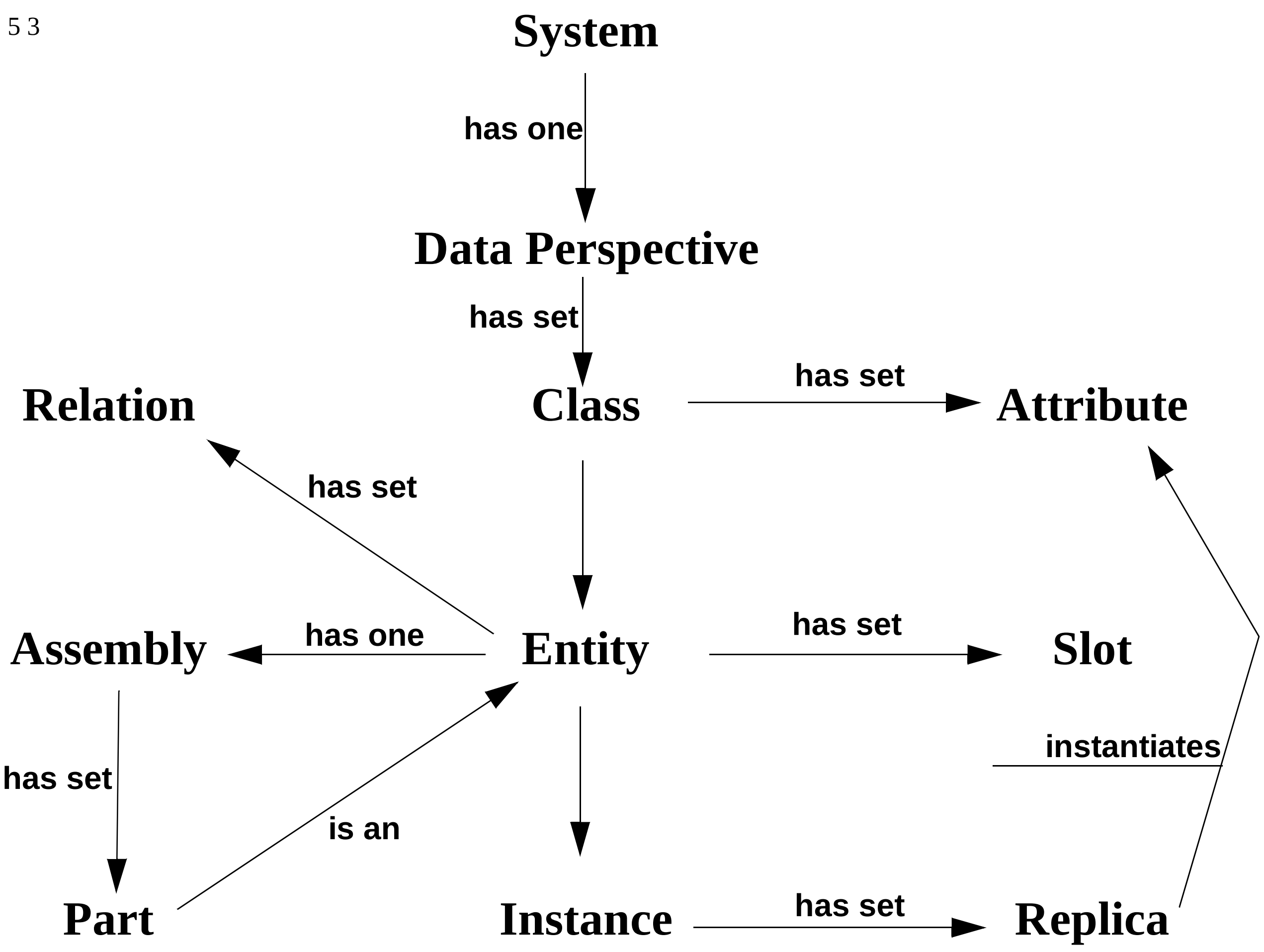
FIGURE 5 2



		Perspective Data	Perspective Event	
Interpenetration	Horizon	Datum	Interrupt	World
Mutual Support	Context	Class	Sheaf	Environment Meta-System
Interactive Relation	Facet	Entity	Signal	System
Local Consistency	Attribute	Instance	Event	Object
Truth o(Manifestation	(Fate)	Merged Spacetime/Timespace Openness		Eject

will be appended to each diagram. The reduction to 'conceptual cores' allows us to project a highly symmetrical model of relations on the whole field of the methods making asymmetries visible. It is necessary to have all the details of the method bridges before us as we begin to attempt to construct our software meta-methodology. Thus, a complete symmetrical model will be developed here

FIGURE 5 3



which step by step abstracts and relates the minimal methods to give a picture of the whole field of methods. This model will show how an infrastructure for the field as a whole may be derived. As that field infrastructure is studied,

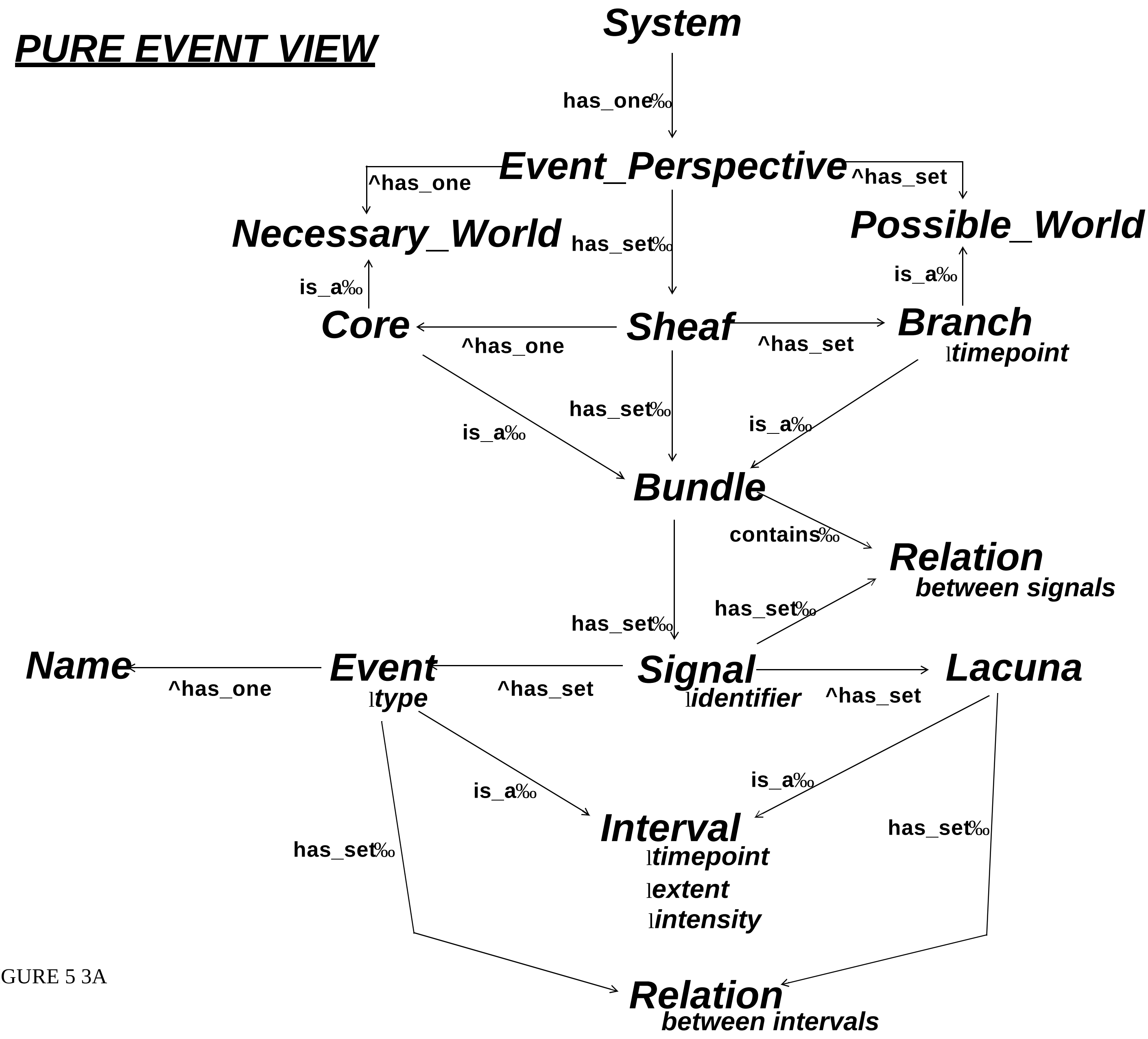


FIGURE 5 3A

the asymmetries begin to appear, the explanation of which is the focus of the rest of this essay.

In the foregoing, the method bridges have been

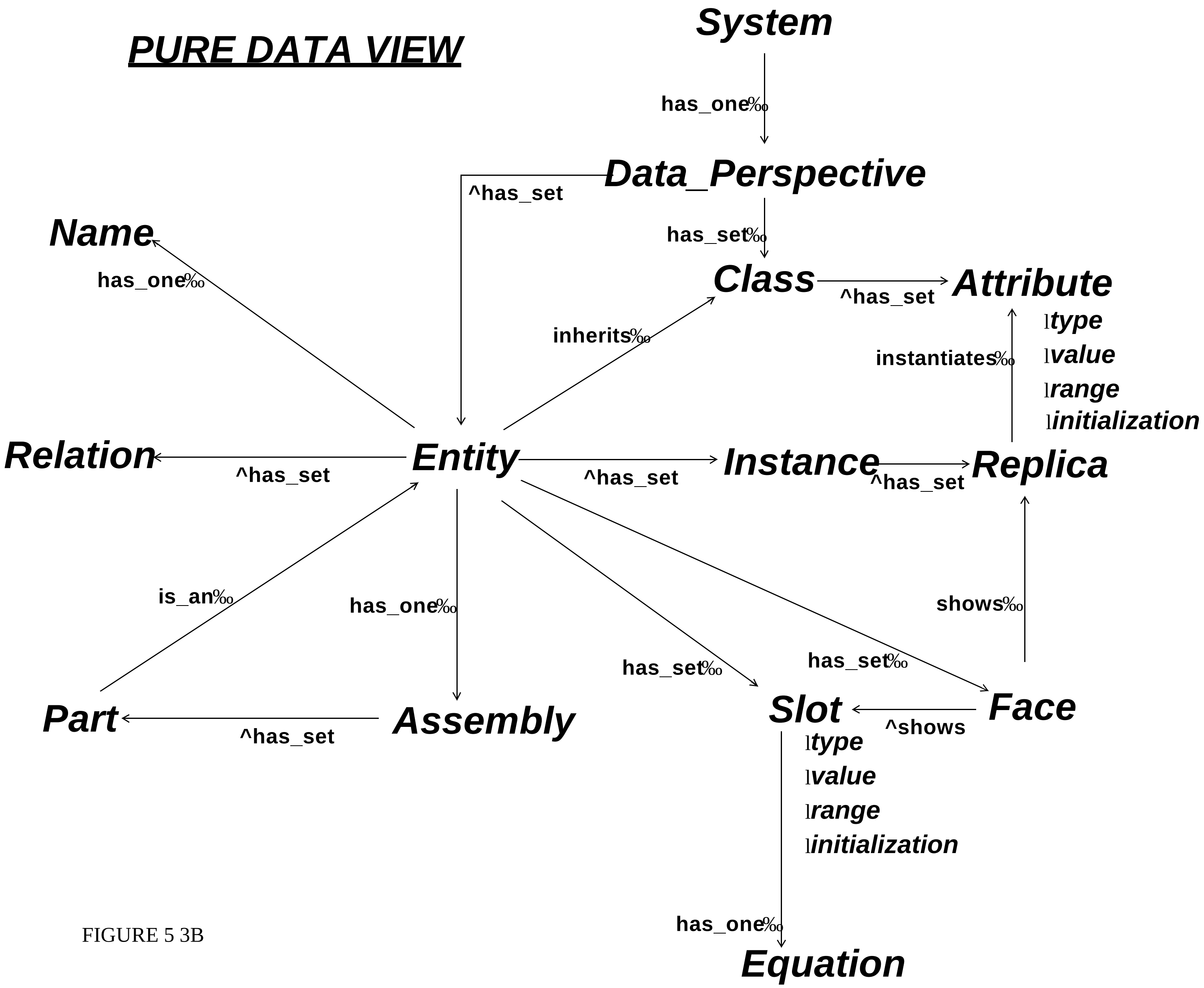
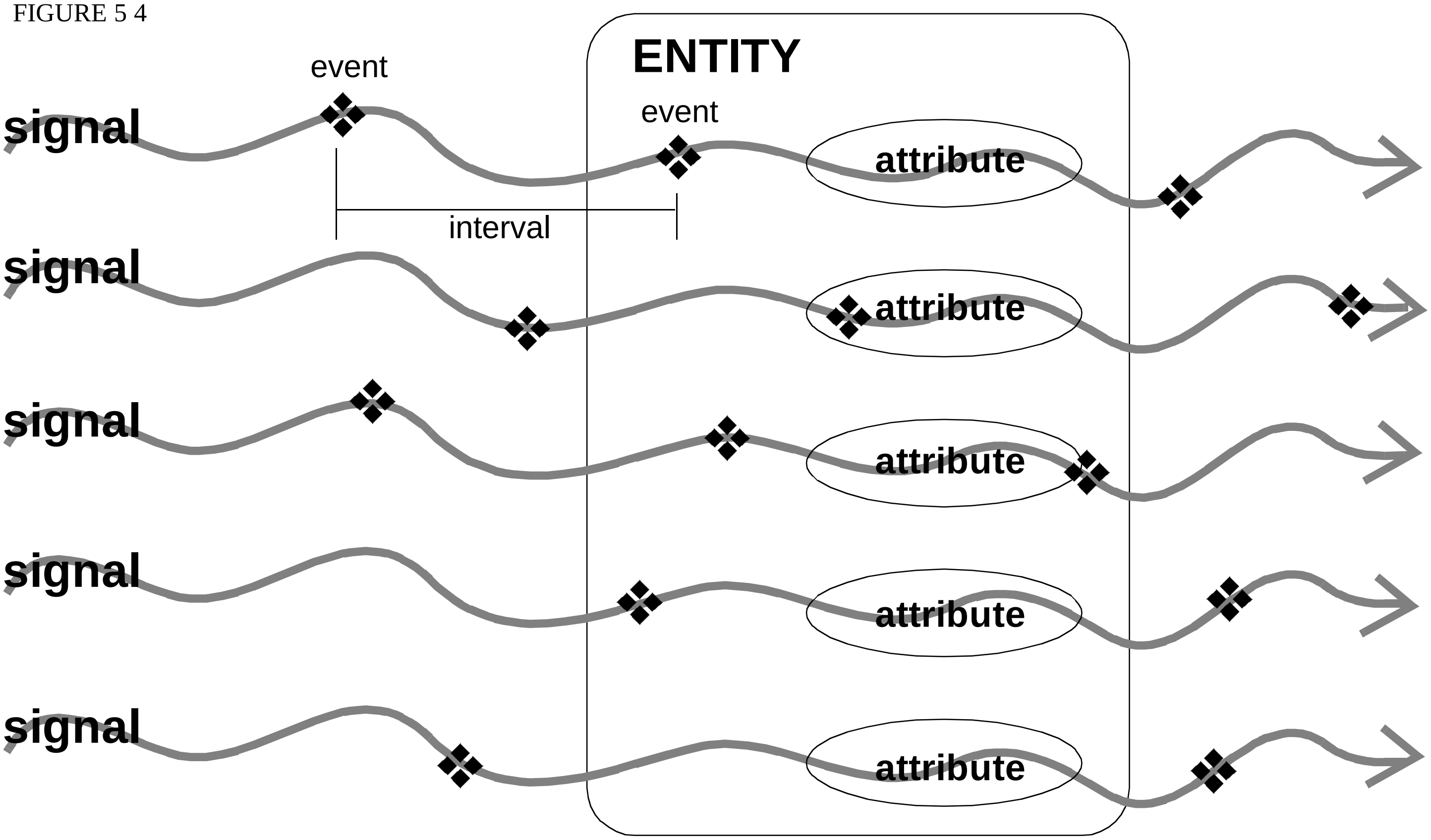


FIGURE 5 3B

abstracted into what are called ‘conceptual cores.’ These attempt to distill the minimal system of concepts from each method bridge. This has been done in order to allow easy manipulation of all the concepts within the field of software methods. If we wish to discover the structure underlying this field, it is necessary to reduce the methods to theoretical structures that can be compared and contrasted. For instance, these conceptual cores can be used to see how the methods define each particular perspective. There are three ‘cores’

FIGURE 5 4



per perspective. The relation between these cores can be seen as a diacritical[43] system of concepts that work together to make a particular perspective visible in relation to a software system. Each perspective can be seen

[43] See Bib#

DESIGN ELEMENT FLOW

DUAL REPRESENTATIONS

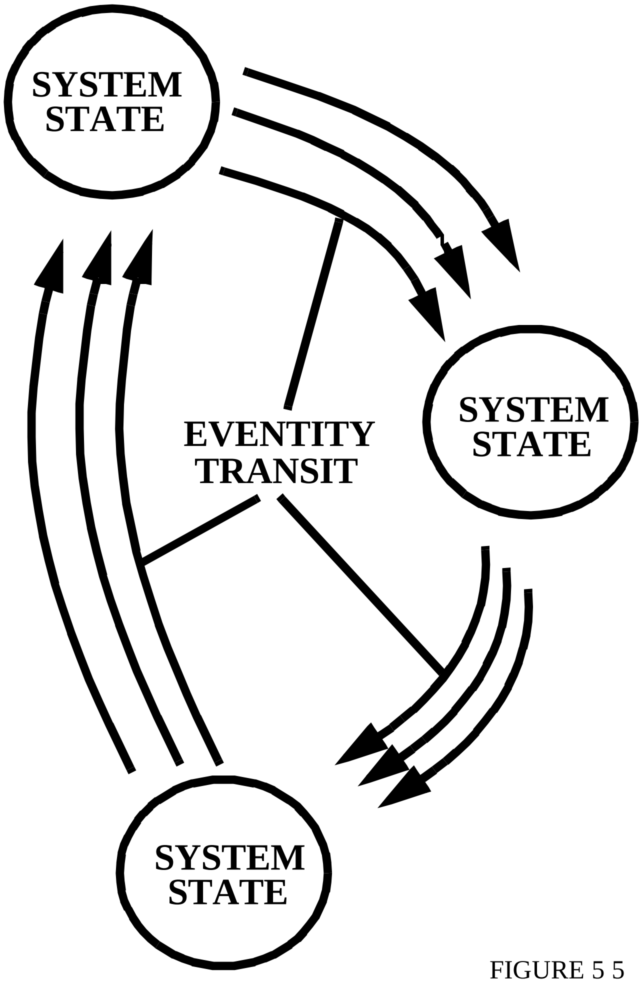


FIGURE 5 5

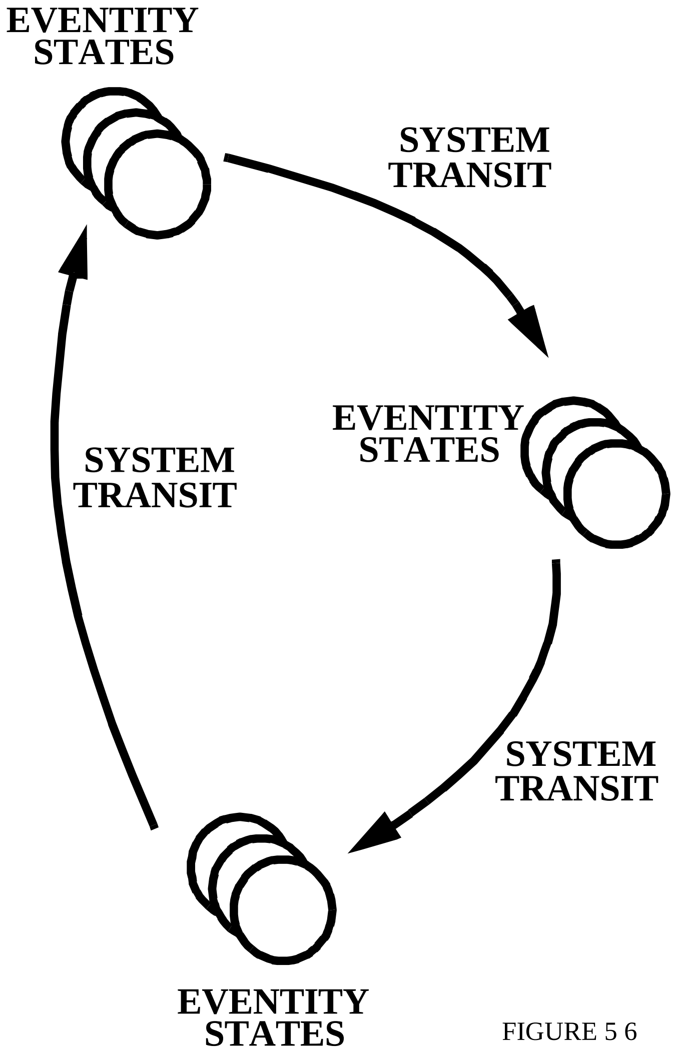


FIGURE 5 6

as being defined by its three adjacent conceptual cores, and all the conceptual cores working together define the system as a whole.

The definition of conceptual cores also allows us to isolate concepts associated with representing viewpoints from those connecting concepts which tie these representations together. Viewpoint representations are described here as of three types: whitebox, linking or blackbox. Each viewpoint has all three kinds of representation. Each conceptual core contains two viewpoint representations and two connecting concepts. The connecting concepts may be separated from representational concepts to isolate the infrastructure of the method field. This substructure is represented as a vector equilibrium. The vector equilibrium was identified by Buckminster Fuller as a key conceptual structure in his geometry of thought.[44] It is an important threshold of theoretical complexity. In this case, the vector equilibrium represents the conceptual infrastructure of the field of software methods. This infrastructure of connecting concepts, once isolated, can be used to identify the core

[44] See Bib#

of the infrastructure which contains four key systems design ideas: communication, causality, cohesion, and control. These, in turn, mask the singularity of pure immanence.

This reduction of the field of methods to

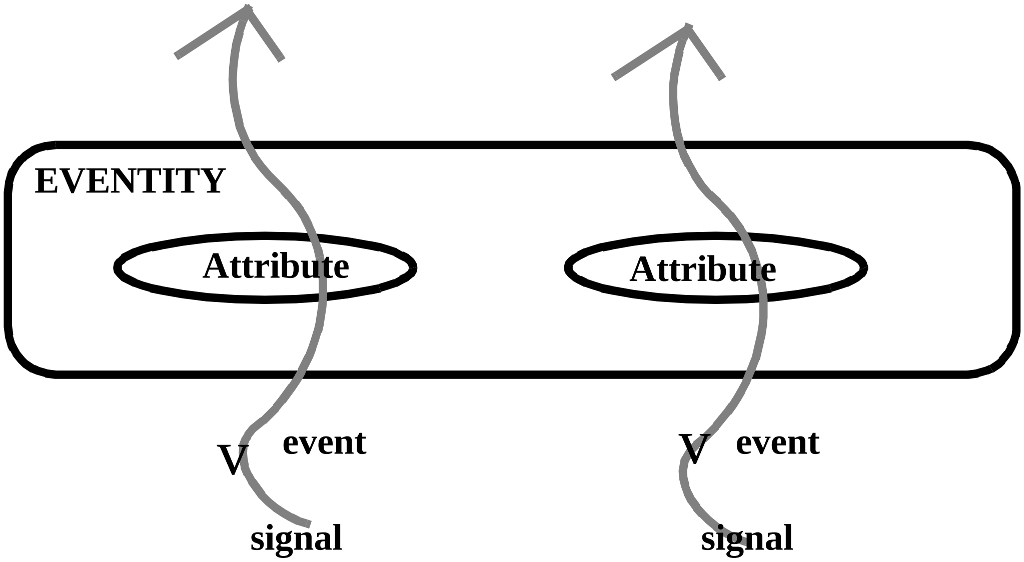


FIGURE 5 7

conceptual cores of the two-way method bridges leads to the separation of the connecting concepts from the viewpoint representations which isolates the infrastructure of the formal system of methods. The analysis of the infrastructure, in turn, leads us to identify its core concepts. And finally, the hypothesis of the singularity at the center of the infrastructure is posited. All these analytical moves are calculated to make the field of methods more understandable as a whole in preparation for the meta-methodological discussions that follow. Hopefully this brief discussion of the analysis of the field of software methods will serve to orient the reader. This is a complex subject which has many aspects. Using abstractions such as these to render the field and its underpinnings visible will help lay out the important landmarks in that field and allow it to appear as a single gestalt worthy of meta-methodological study.

◆ ◆ ◆

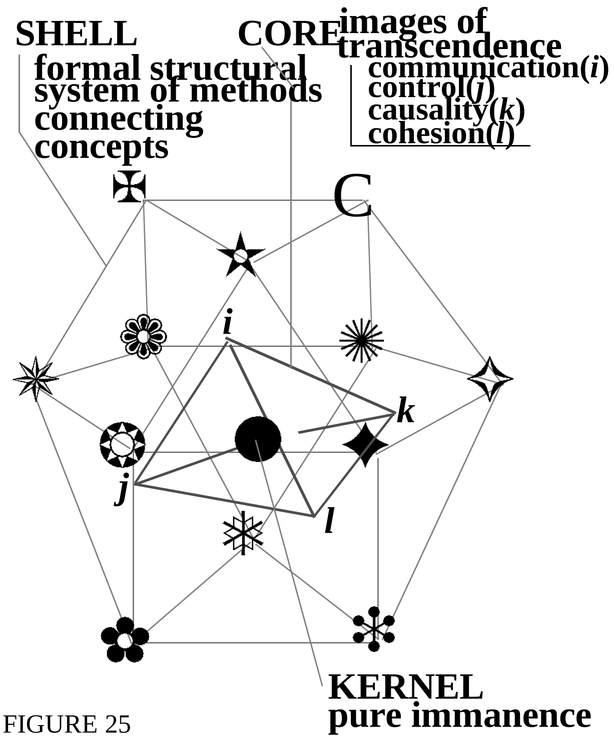


FIGURE 25

The question to be posed now is how to fit these four basic perspectives that have been identified into Klir's representation of a general formal-structural system. The representation of the formal-structural system is too complex to explain in an essay of this length. Thus, only the highlights will be given to show the relationship of the four perspectives to what

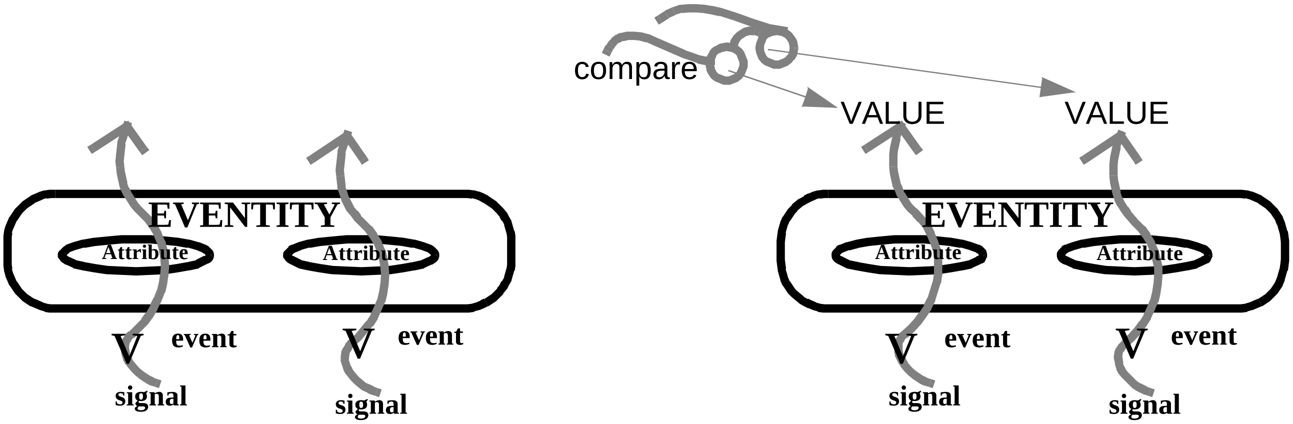
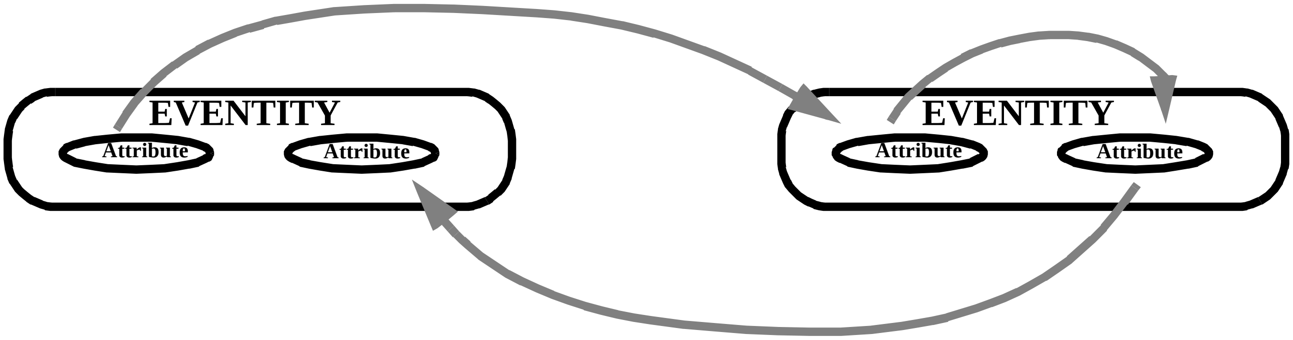


FIGURE 5 8

INFORMATION FLOW DIAGRAM



INFORMATION NETWORK

FIGURE 5 9

Klir defines as a ‘system.’ Klir’s definition is rather elaborate because it involves the delineation of several ‘epistemological levels’

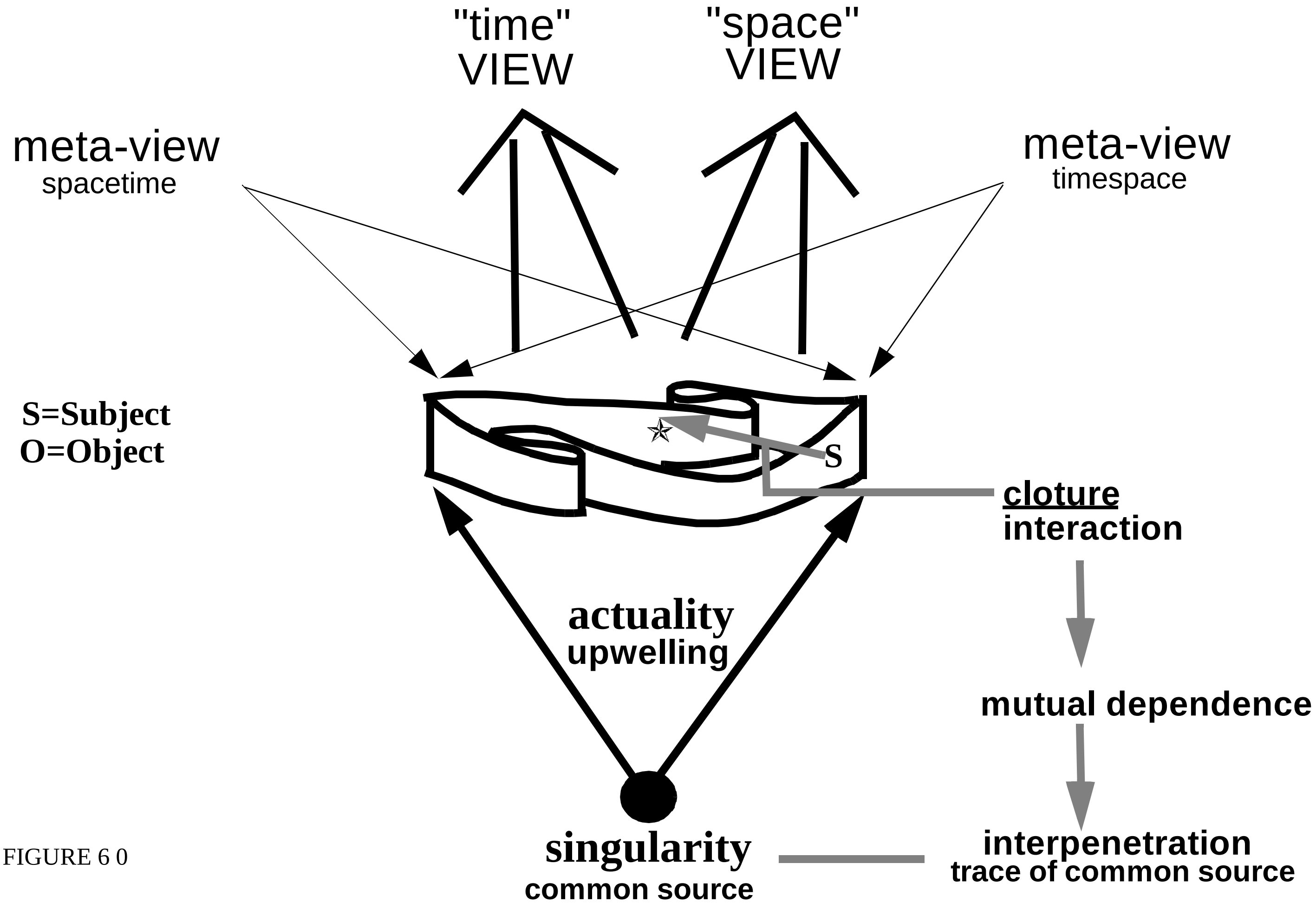


FIGURE 6 0

upon which the term ‘system’ takes on different meanings.

Klir’s EPISTEMOLOGICAL LEVELS

infinite regress

Level 4,5,... Meta-systems

Level 3 Structure system

Level 2 Generative system

Level 1 Data system

Level 0 Source system

Level -1 Object system

Level -2 Lifeworld

Level -3 Existence

Level -4 Phenomenological World

Level -5 Ontological World

Level -6 Onto-theological World

Level -7 Ultimate Reality & Meaning

Grounding ONTOLOGICAL LEVELS

In order to establish the relation between Klir's epistemological levels and the references to ontology made in part one of this essay, consider the negative levels added to those delineated in ASPS. For Klir, the object of investigation does not belong among his epistemological levels except as an adjunct to the source system which is the representation of the object system.

> For our further considerations, let the term 'object' be defined as a part of the world that is distinguishable as a single entity from the rest of the world for an appreciable length of time.[45]

Klir goes on to differentiate material from abstract objects, each of which are, in turn, divided into those found within the world and those that are man-made. This definition carries a great deal of metaphysical baggage emanating from the development of the Western philosophical and scientific tradition. At the very least, the concept of 'object' depends upon the coequal definition of a 'subject' who is the perceiver of the object. Both of these terms are laden with many philosophical interpretations which provide competing grounds for their theoretical

[45] ASPS pp33-44

essential attributes relations	**SYSTEM EVOLUTION**	**essential diacriticality**
accidental attributes relations	**SYSTEM DYNAMICS**	**accidental diacriticality**
numerical relations between instances	**POPULATION FLUCTUATIONS**	**instantiation diacriticality**
spacetime position relations	**DEMOGRAPHY**	**position sensitive diacriticality**

FIGURE 6 1

elaboration. Instead of delving into that morass of competing dialectically-related positions that make up the bulk of the Western philosophy, let us attempt to extend Klir's hierarchy in the other direction into the realm of ontology in order to get a wider perspective on the problem of the epistemology of the object.

Both the subject and object exist in what has been called the 'Lifeworld.' Husserl first introduced this term in his book The Crisis of European Sciences and Transcendental Phenomenology.[46] Subsequently, the concept was developed further by Alfred Schutz in Structures of the Lifeworld.[47] The Lifeworld is the world of everyday life in which we all live. Science and Technology are cultural products that are built upon the Lifeworld.[48] Without the Lifeworld they would have no reality at all. Objects are idealizations that exist based upon the concrete foundations of the Lifeworld. Subjects are also idealizations of real people that live their day-to-day lives and perceive ordinary things that are sometimes raised to the status of an object by elaborate focusing and isolating conceptual procedures.[49] These

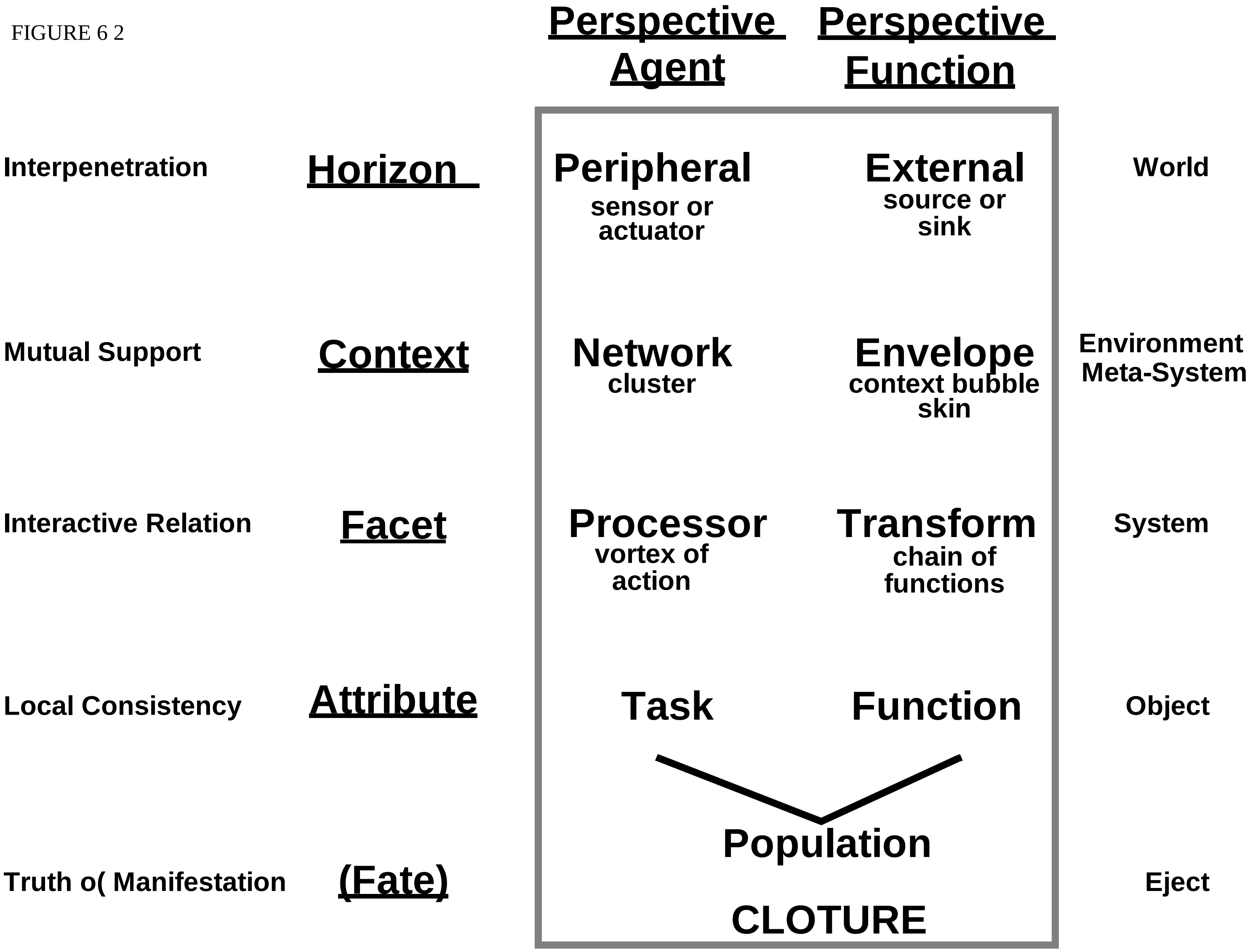
[46] See Bib#
[47] See Bib#
[48] See Bib#

elaborate procedures for perceiving ordinary things ‘clearly and distinctly’ were first inaugurated by Descartes in his Discourse on Method.[50] Klir calls these observational channels. These idealizations that make us subjects who perceive objects have over the

[49] See Feyerabend speaks of the theories embedded in perception and the need to overcome these in order to ’see the right thing’ in an experimental setting. The famous example is Galelio attempting to get others to see the same thing as himself when looking at the moon through a telescope. This is discussed at length in Bib#.

[50] See Bib#

FIGURE 6 2

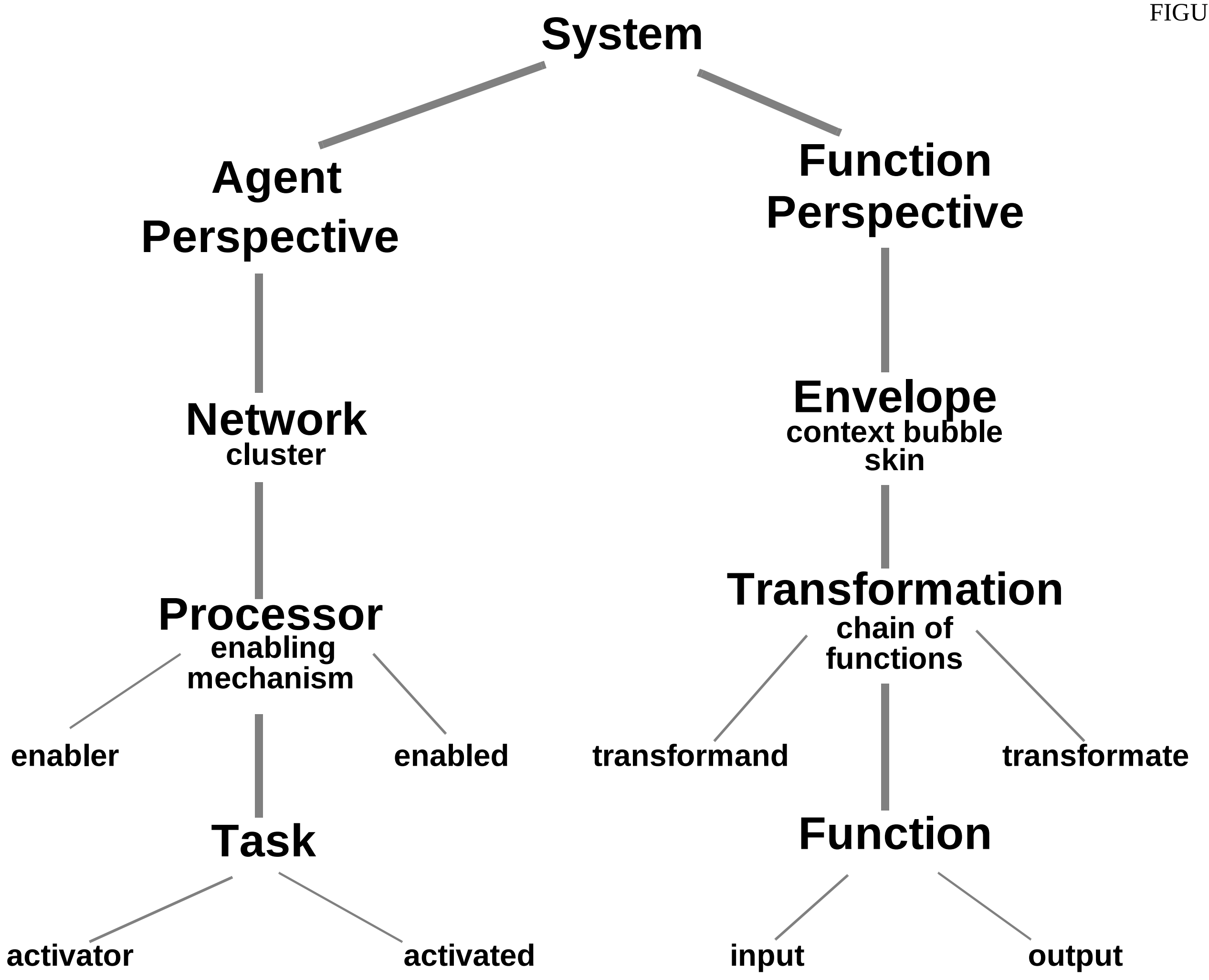


		Perspective Agent	**Perspective Function**	
Interpenetration	**Horizon**	**Peripheral** sensor or actuator	**External** source or sink	World
Mutual Support	**Context**	**Network** cluster	**Envelope** context bubble skin	Environment Meta-System
Interactive Relation	**Facet**	**Processor** vortex of action	**Transform** chain of functions	System
Local Consistency	**Attribute**	**Task**	**Function**	Object
Truth o(Manifestation	**(Fate)**	**Population** **CLOTURE**		Eject

course of centuries become second nature to many of us who were successful in being 'educated.' For us, there is no problem when Klir assumes he can pass over the Lifeworld with a scant three paragraphs[51] and go straight to defining objects without even mentioning our subjectivity. Much of the energy of philosophy in the last hundred or so years has been spent by the rediscovery of the Lifeworld

[51] See Bib# page 33

FIGURE 6 3



and how it has been glossed over by idealizations of science and philosophy within the Western tradition. Unfortunately this effort appears to have been lost on Mr. Klir. Yet he is not alone in this. Objective and subjective idealization that glosses over the Lifeworld is rampant in scientific discourse. These glosses attempt to mantain our stance in the present-at-hand mode of perceiving reality. The problem is that it reduces the multiplicity of life to a caricature.

The next level (-3) beneath the Lifeworld is the realm of 'existence.' Existence is the one ontologically-related concept excluded from the Western concept of 'Being.' Existence concerns the concrete specifics of lived experience. It is the realm of accidents which make up so much of the world in which we live. All the individual particularities that are one way rather than another, seemingly by chance, that gives life its incredible diversity. Traditional philosophy always ignored existence in order to get on to more lofty ontological concepts. This realm was rediscovered in this century by the existentialist philosophers such as Heidegger,[52] Camus,[53]

[52] See Bib#

and Sartre;[54] who proclaimed the slogan “existence proceeds essence” as the cornerstone of their ruminations on everything. The fact that the Lifeworld is filled with non-essential particulars which could be many different ways, but in each instance is one particular way, cannot be denied. Existentialism is a field of philosophy that addresses this issue and seeks its significance without skipping over it.

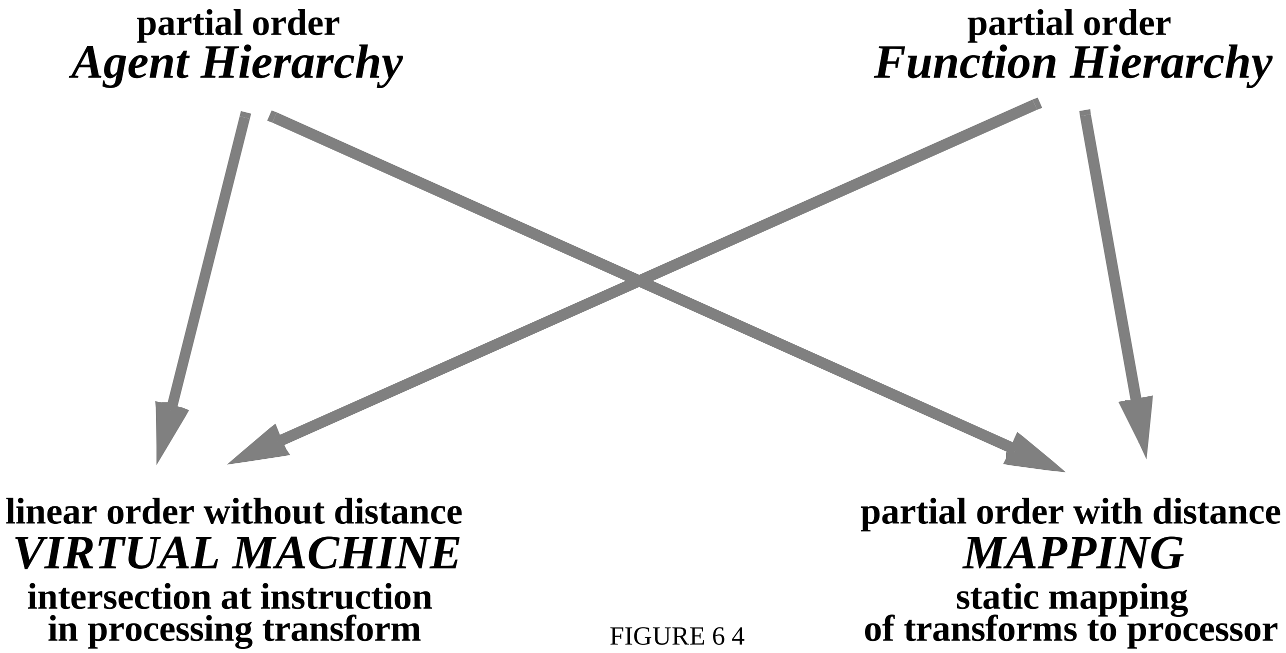


FIGURE 6 4

Beneath the level of existence is another

[53] See Bib#

[54] See Bib#

strata (-4) which is called the phenomenological realm. This realm has also been explored a great deal since the turn of the century, predominately by the followers of Edmund Husserl.[55] Phenomenology deals with the world of essences as perceived in pure consciousness. Essences of things are their necessary attributes and the constraints that give those attributes inner coherence. From essences, we build up simple ideas by a process of ideation. Ideas are glosses on essences. Once these glosses are established, we use induction and deduction to move between idealized abstraction and idealized particulars. Phenomenology seeks to unearth the roots of our idealized glosses in the perceived things. Phenomenology is a very significant advancement in modern thought. It replaces the naive ungrounded abstractions of traditional philosophy and science with a precise vocabulary for describing perceptions, both material and abstract. Klir’s division of objects into these two categories is a very crude approximation to the more subtle noetic and noematic phenomenological constructs that deal with our apprehension of essences and ideas.

[55] See Bib#

Beneath the phenomenological strata is the ontological world which is level -5. Here we first encounter the concept of Being. Being is the most general concept we have. All the essences of things depend on Being. So phenomenology flows directly from ontology. The relation between individual beings and the concept of Being is called by Heidegger 'ontological difference.'[56] Being itself is not

[56] See Bib#

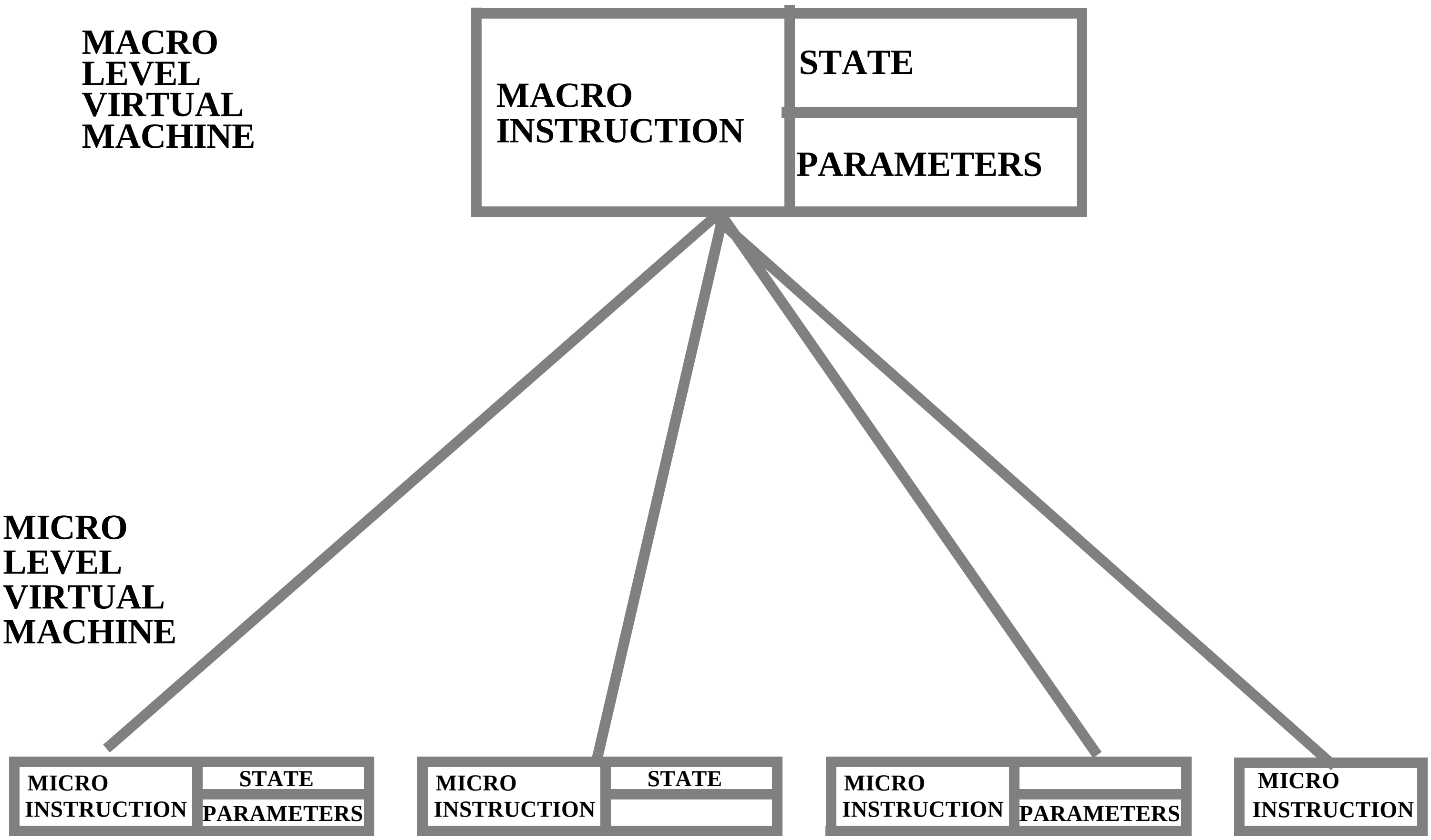


FIGURE 6 5

contained in any one being. A class cannot be a member of itself, according to Russell and Whitehead in Principia Mathematica.[57] This alone would justify the distinction of ontological difference to avoid paradox. Instead, all beings are contained in the world projected by human beings. We are beings-in-

[57] See Bib#

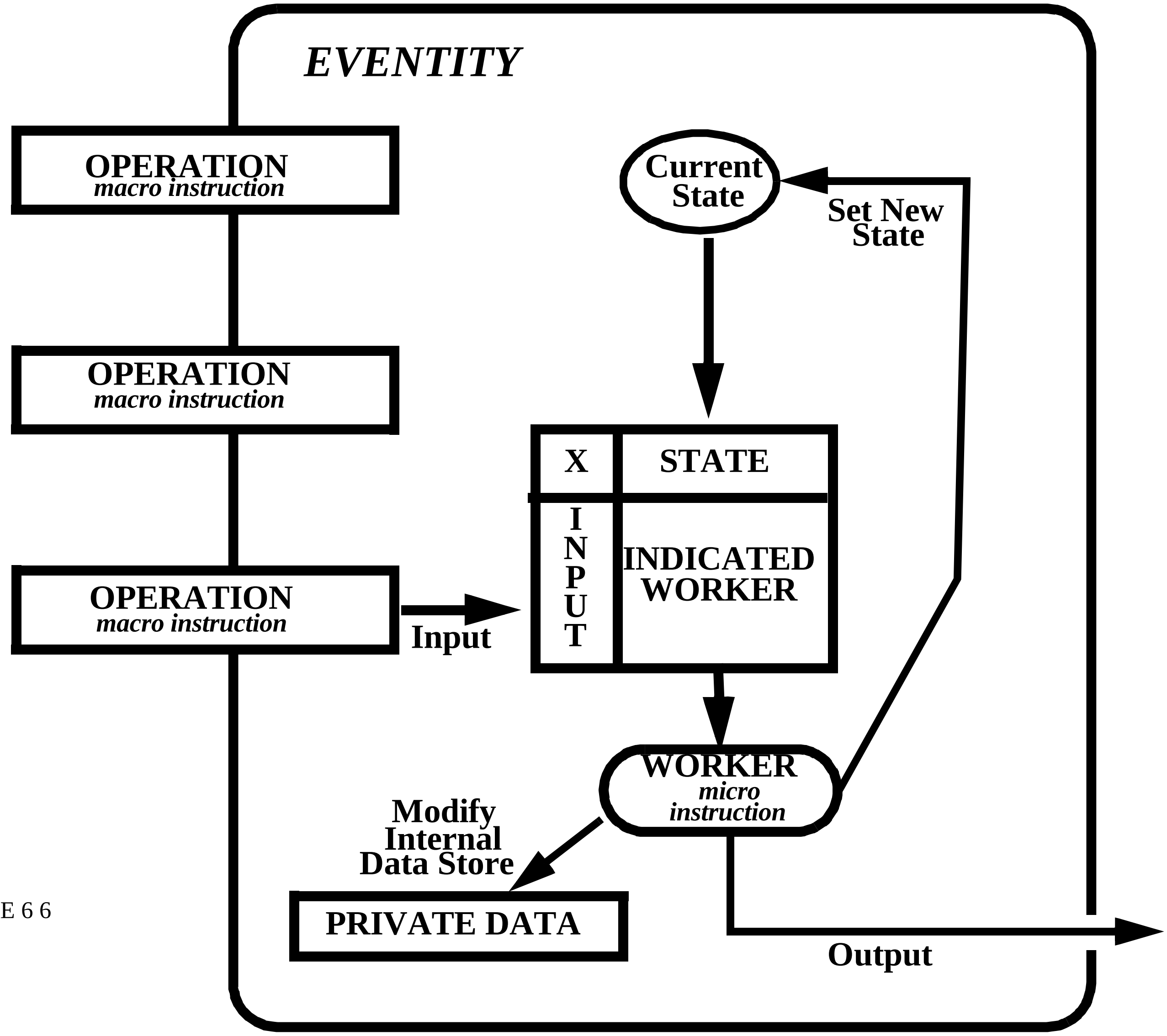


FIGURE 6 6

a-world which Heidegger calls Dasein (being there). It is this concept of *Being* and its supposed fragmentation that was treated in the first section of this essay at length. The fragmentation of Being is probably the most significant event for the Western philosophical and scientific tradition in this century. Before this century started, there was only one kind of Being which was defined by Kant and accepted by everyone as our inheritance from Aristotle. Now there can be recognized at least four types of Being. These four types of Being have been described in detail, and software's essence flows from just one of them. It is the fragmentation of Being that makes software ontology important. If there was only one kind of Being now, like there used to be, then there would be no question of the ontological foundations of software engineering. But things have changed radically. And that change is the deepest possible kind of change. It is this realization of deep change that needs to inform our understanding of Klir's epistemology of systems. The whole technological world is teetering on the edge of an abyss. The Crisis that Husserl recognized has deepened.[58] We exemplify that unbalance

[58] See Bib#

in everything we do. The fractures at the ontological level percolate up through the phenomenological strata, the existential strata, the Lifeworld strata, until the edifice of the idealizations of the structural system built so carefully by the philosophy and science begins to crumble. Outwardly there is a shiny exterior of technological progress and inwardly there is ‘mithraic’ active nihilism destroying meaning in the world. (We call this active nihilism ‘mithraic’[59] to show that the roots of this nihilism are historically very deep. Active nihilism has its roots in the historical

[59] See Bib#

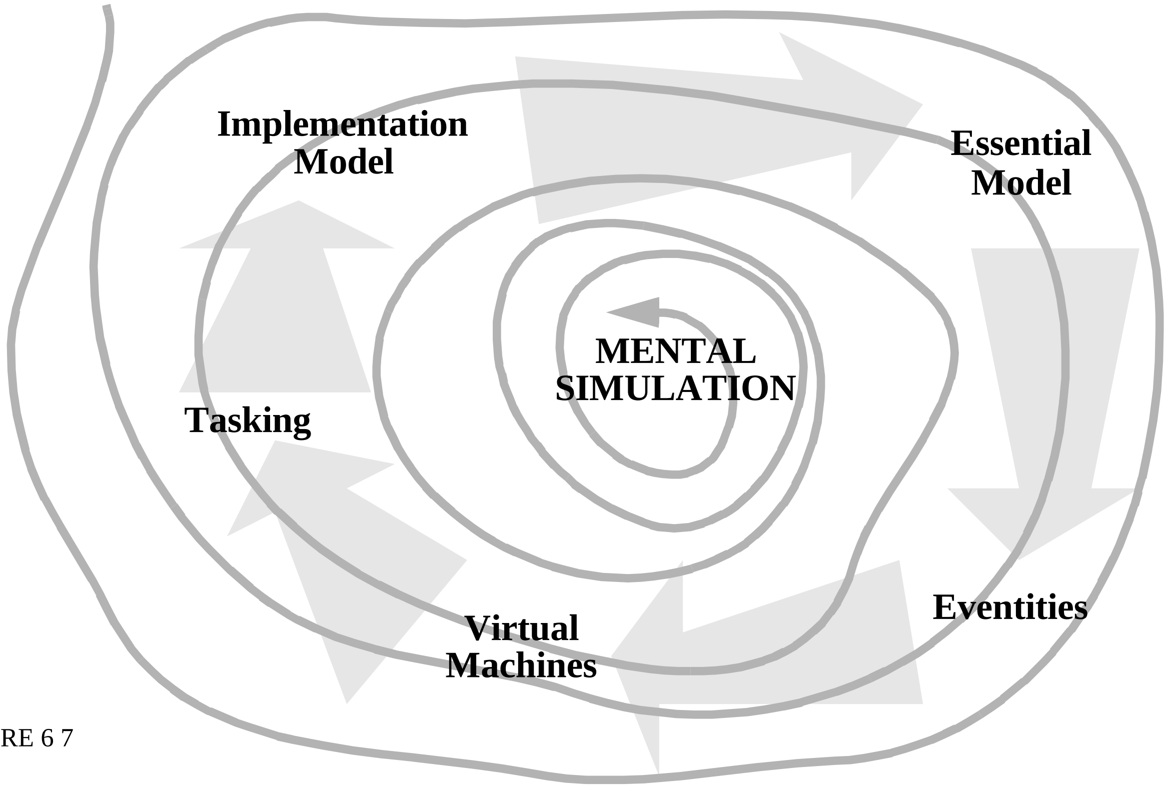


FIGURE 6 7

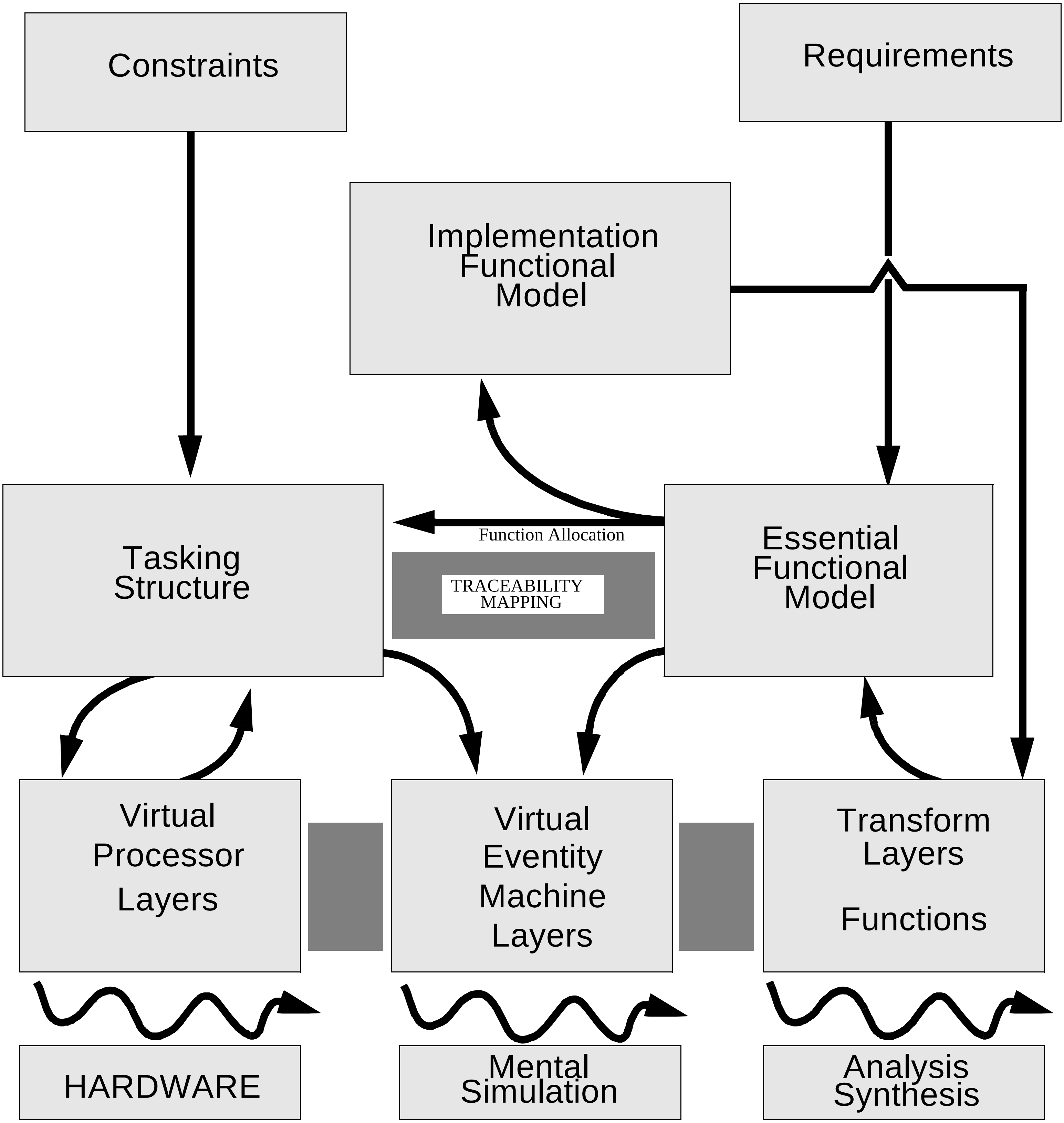
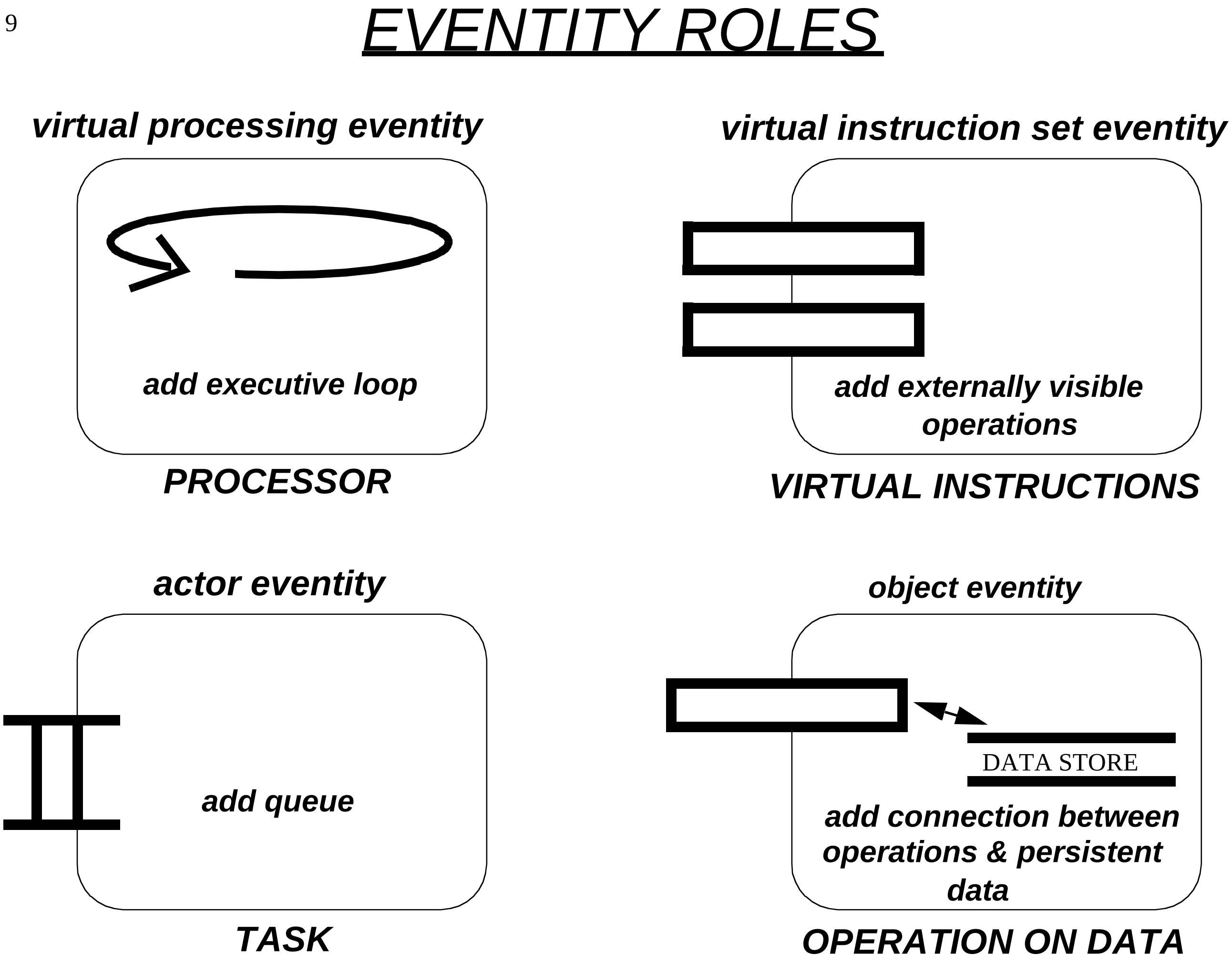

Constraints
Requirements
Implementation Functional Model
Tasking Structure
Function Allocation
TRACEABILITY MAPPING
Essential Functional Model
Virtual Processor Layers
Virtual Eventity Machine Layers
Transform Layers
Functions
HARDWARE
Mental Simulation
Analysis Synthesis

FIGURE 6 8

development of Zoroastrianism[60] which was taken to Rome by converted soldiers as a mystery religion. This was a Greek style mystery cult that flourished all over the Roman empire and was related to Gnosticism[61] and Manicheism[62] and subsequently influenced the development of Christianity.[63] The development of extreme conflicting opposites

[60] See Bib# and ??? by same author.
[61] See ??? on Gnosticism.
[62] See ??? on Manacheism.
[63] See Bib# and Bib#

FIGURE 6 9


EVENTITY ROLES
virtual processing eventity
add executive loop
PROCESSOR
virtual instruction set eventity
add externally visible operations
VIRTUAL INSTRUCTIONS
actor eventity
add queue
TASK
object eventity
DATA STORE
add connection between operations & persistent data
OPERATION ON DATA

as ontological categories was inherited by all of these related religions from Zoroastrianism.[64]) We experience that loss of meaning in the world as an existential crisis described so well by Camus in The Myth of Sisyphus. We see it as crime rates, divorce rates, inflation rates,

[64] See Bib#

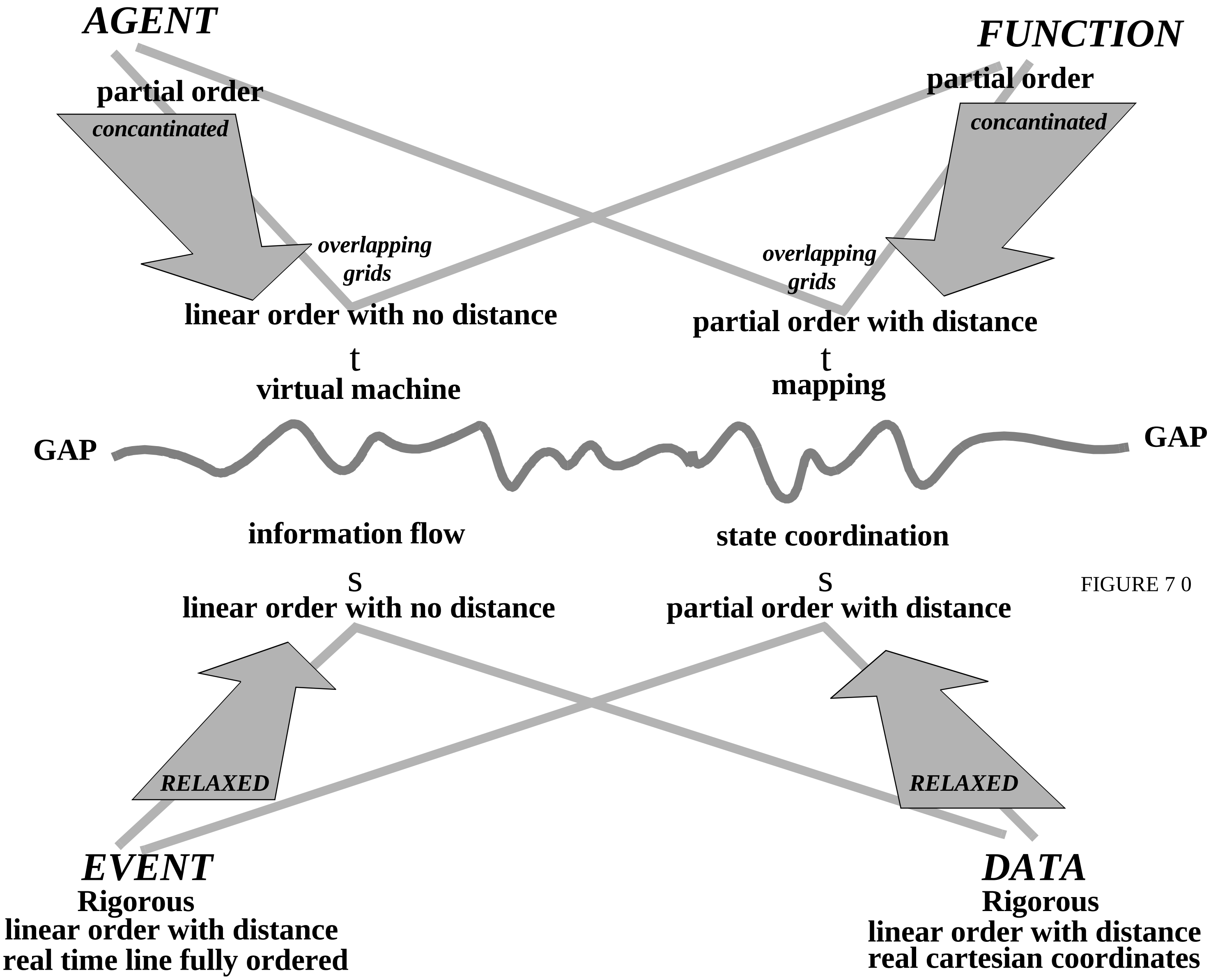

AGENT
FUNCTION
partial order
concantinated
partial order
concantinated
overlapping grids
overlapping grids
linear order with no distance
partial order with distance
t
t
virtual machine
mapping
GAP
GAP
information flow
state coordination
S
S
linear order with no distance
partial order with distance
RELAXED
RELAXED
EVENT
Rigorous
linear order with distance
real time line fully ordered
DATA
Rigorous
linear order with distance
real cartesian coordinates

FIGURE 7 0

suicide rates, abortion rates, deficit increase rates, pollution rates, garbage dump fill rates, ozone hole expansion rates, resource depletion rates, poverty and starvation rates, along with a myriad of other indicators of modern ‘progress’ and our ‘manifest destiny’ working itself out in the world.[65]

The concept of Being is interesting because it is almost only the Indo-European languages that have this concept as part of the paraphernalia of grammar. Other languages like Chinese and Arabic do not have a concept of Being[66] as sucheven though early Western scholars attempted to find it there, and in frustration, projected it upon these and many other languages]. Instead, these other languages have a concept of existence, and what lies beyond that is considered *empty* or *void*. Thus, the levels (-4, -5, and -6) do not exist for most other non-Indo-European languages. The concepts of Being, with both essences and accidents, are a ridiculous fantasy from their perspectives. There is only the absolute, and from it flows existential beings. The concepts which are fused together into our concept of

[65] There are endless exampeles of tales of woe. See Bib# and Bib#. Also Bib#.

[66] See Bib#

Being are handled grammatically in other ways. The concept of Being contains the idea of copula (A is B), ontos (A is !), attribution (A is blue), and identity (A is A). These concepts have been extracted from the grammatical

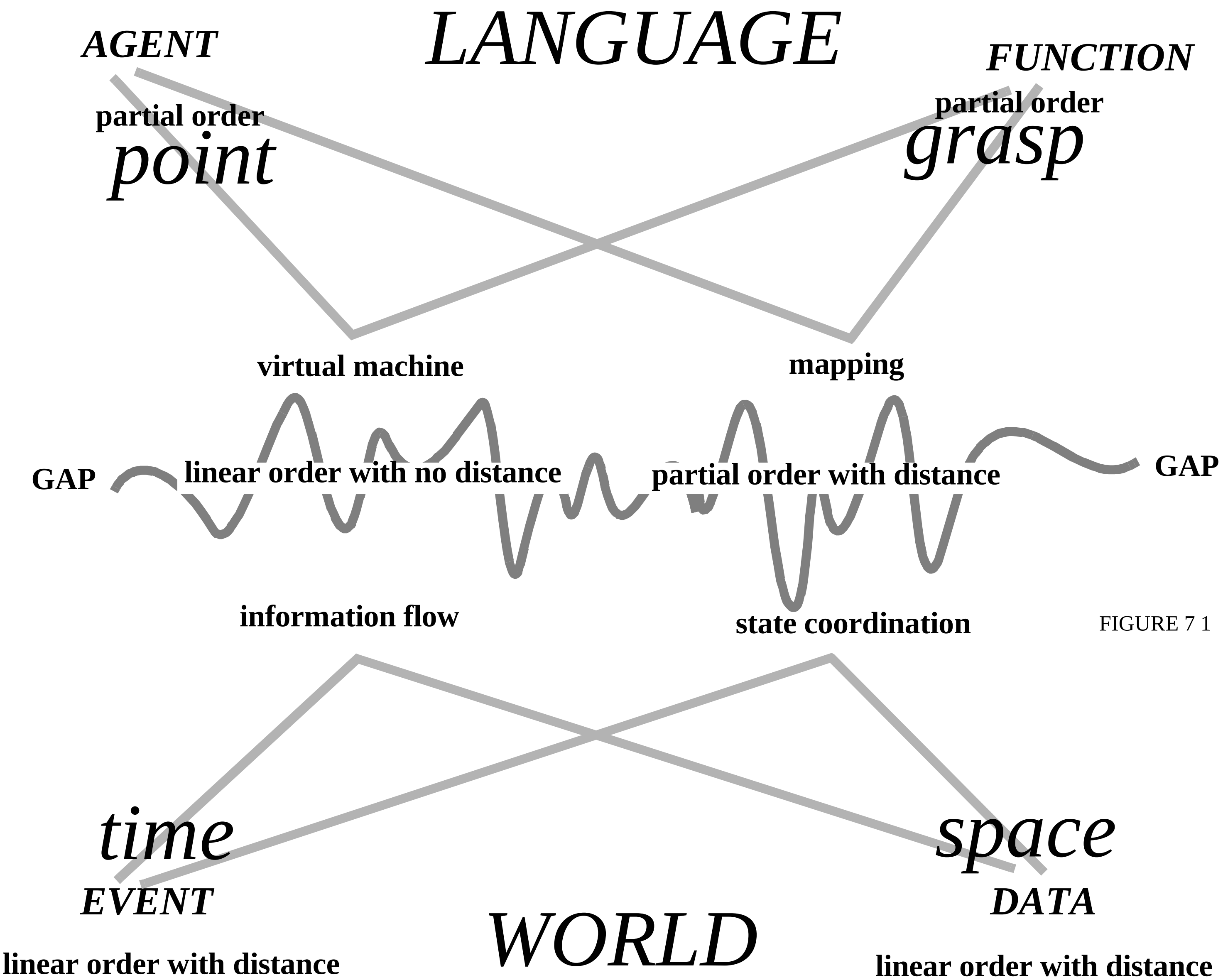


FIGURE 7 1

Being over the centuries and make up our ontological concept of Being. The ancestors of the Indo-Europeans[67] are called the Kurgen people because they built burial mounds called ‘kurgens.’ They came out of central Russia and devastated the entire world about the fourth millennia BC. Prior to that it is thought they originated between the Black and the Caspian Seas about 6000 years BC.[68] They were the first to use the domesticated horse in warfare. They became the Anglo-Saxons, Germans, Gauls, Latins, Greeks, Hittites, Persians, and Aryan Indians. What is of interest is how their concept of Being came into existence. In all Indo-European languages, Being is the most irregular verb. It is made of several different Indo-European root verbs melded together over a long period of time to become what we think of as a single grammatical term today. The point is that Being started as a composite of different verbs, all meaning ‘to remain’ or ‘to persist’ which were blended together to form a single term with many different conceptual components. Thus, the current fragmentation of Being is merely a return of sorts to the original fragments which separately do not

[67] See Bib#

[68] ??? Scientific American article on origins of Indoeurpeans.

have the same meaning. Without the concept of Being, our language would be like Chinese or Arabic with only existence and the Absolute veiled by Emptiness or Void.

When we look to what Being means, we cannot help but see that the "remaining" and "persisting" denoted by verb fragments that make up the grammatical construct "Being" refer to what C.G. Chang calls a "subtle clinging." This "subtle clinging" is clearly identified by Buddhist philosophy and connected to "clinging and craving" by human beings which is the source of all sorrow. The Buddhist antidote for this malaise is the concept of "emptiness" or "sunyata." Through sunyata one may perceive and become immersed in the Absolute. The concept of Being is a barrier to this kind of experience of Ultimate Reality. Instead, within the Western tradition, there is another strata projected upon the Absolute which Heidegger calls onto-theological. It imagines a Supreme Being who is called "God." For Kant, the Supreme Being was an idealization of the concept of infinity. Each theologian imagines a different means of conceptualizing "God." However, as long as "God" is thought of as having Being or as associated with anything with Being, there is a fundamental flaw which flows from the arising of the artificial concept of Being. As “God” Being faces the Absolute. These conceptual glosses of “God” are found by men to be empty of meaning. Nietzsche then comes along and declares that “God is dead.” He was killed by those who approached Him through empty abstractions or by associating Him with beings

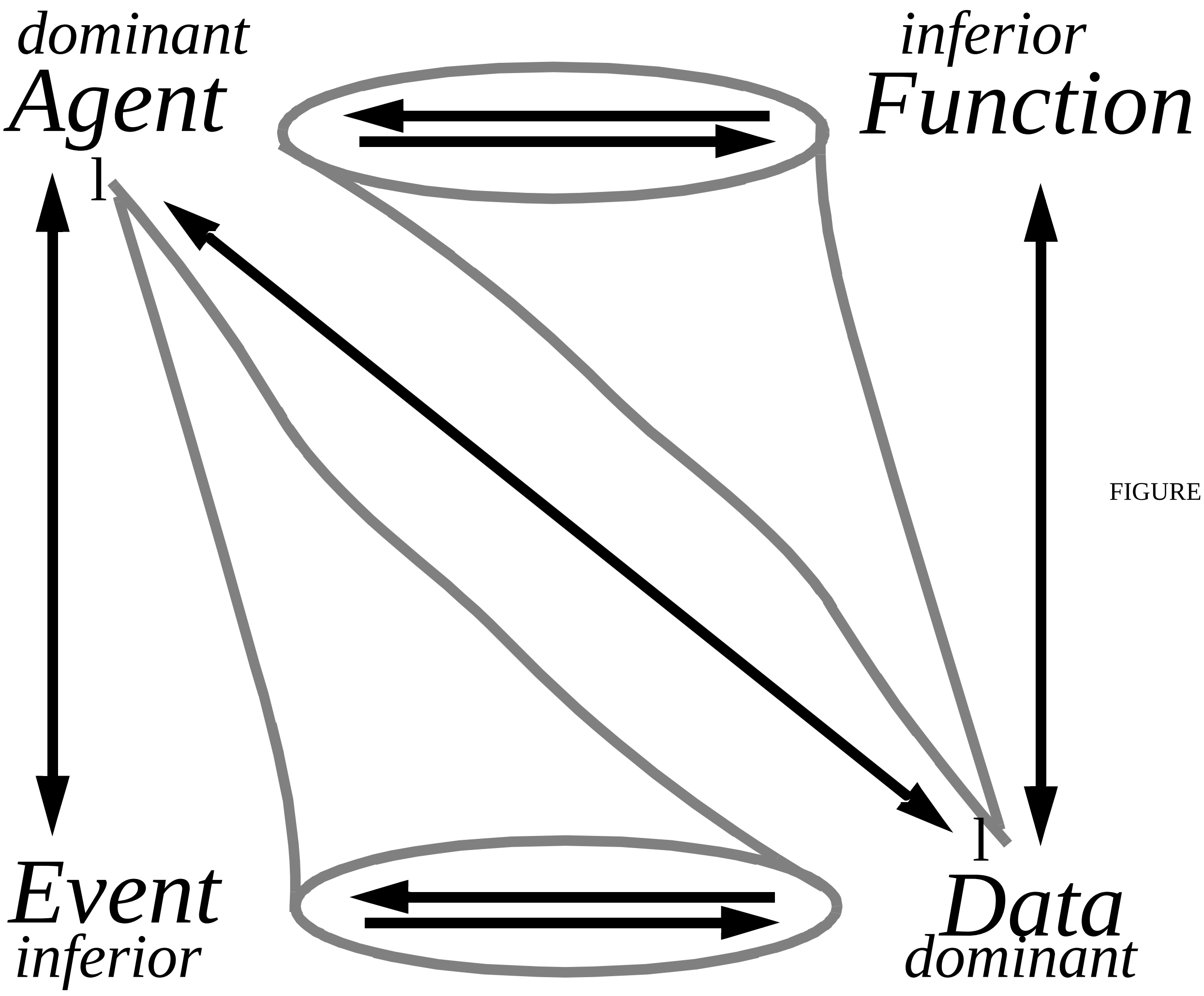
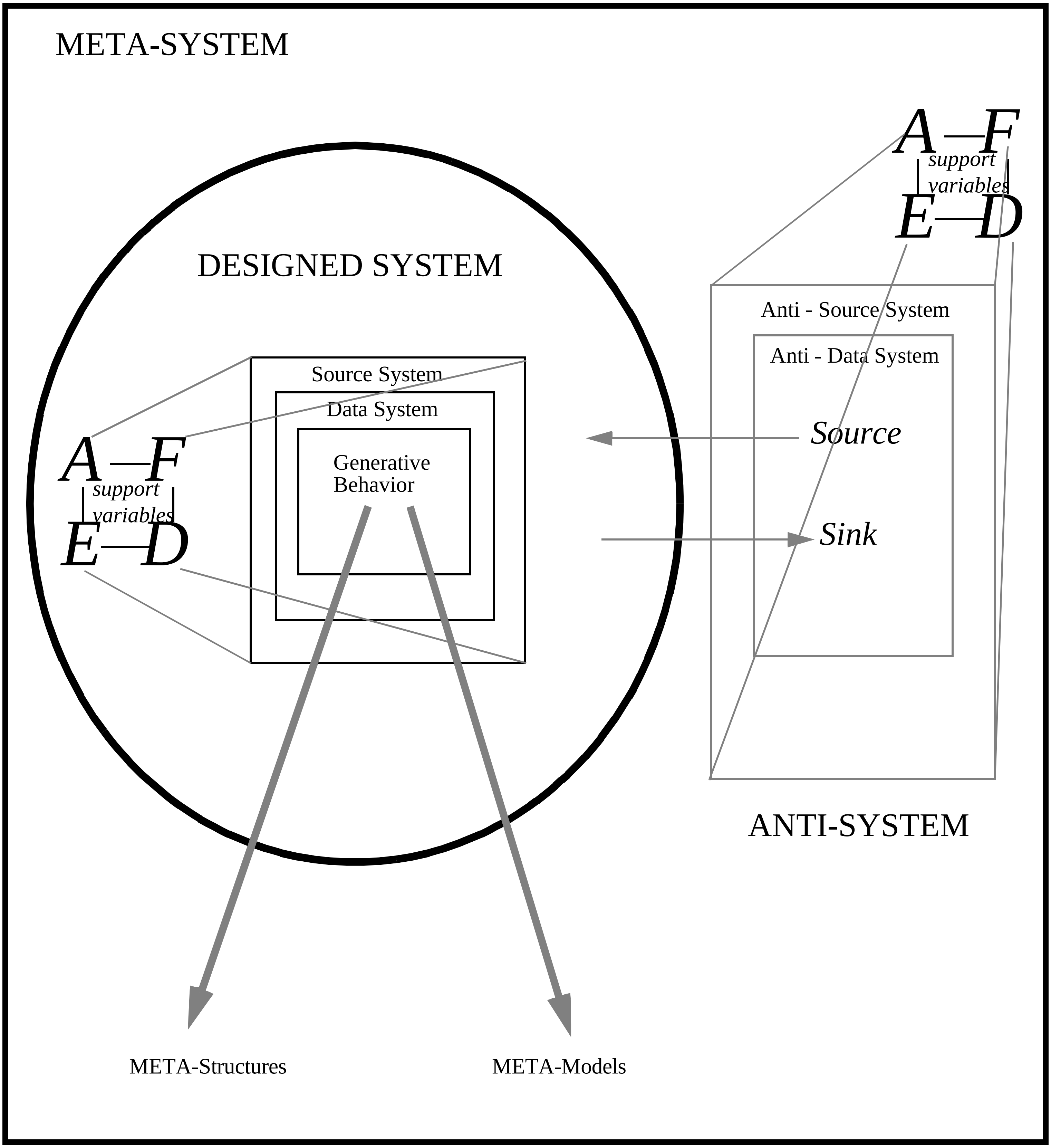


FIGURE 7 2

or Being. The death of "God" is the logical result of the creation of the concept of Being which flowed from the forging of the grammatical construct "Being." Through the death of 'God' we lose contact with the ultimate reality or Absolute that permeates our existence in the lifeworld.

This brief excursion into the strata that underlies the structural system is only meant to give an overview of what is being glossed by Klir. From this it is possible to see how ontology is related to epistemology. These two specialties together make up the subject matter of metaphysics. If the concept of Being fragments, it effects all the levels above it. When "God" was declared "dead," the empty conceptual unity cracked, being fragmented without any unifying force to keep it unified. Essences lost their meaning. Existence became absurd. The lifeworld was realized to be permeated by Mithraic active nihilism, and all the edifices beyond that found themselves resting on the abyss of groundlessness. The great chain of Being was suddenly broken. By saying that "God" is dead, Nietzsche merely pointed out that the emperor had no clothes. Instead of listening, we slay the messengers by attacking Nietzsche and his intellectual heirs. The technologist is caught in a world where the process of destruction appears in the guise of "progress." This so-called "progress" is more aptly named "artificial emergence." Artificial emergence is the production of empty novelty for its own sake. Newspapers, fashions, new model cars, and much technological innovation may be classed as artificial emergence or excrescences (abnormal growths). This is contrasted to genuine emergence such as scientific revolutions or the appearance of a genuinely new artifact that is not merely a combination of other artifacts by some method of permutation of characteristics. Genuine novelty stands out on the background of excrescence. The technologist is defined as the professional whose job it is to distinguish genuine emergence from artificial emergence. All other features of the technologist's job flow from this one characteristic. The technologist, regardless of field, must survey the continuous production of novel technical gadgets and attempt to make use of what is worthwhile for the purposes at hand. The real discrimination of worth depends on making non-nihilistic distinctions between genuine emergence of significant new technologies from the



Wild Software Meta-Systems

FIGURE 7 4

background of marketing clatter over products that are only variations on the same theme. Distinguishing genuine emergence is the highest calling of the technologist. Where the scientist and inventor is trying to manifest genuine creativity by facilitating genuine emergence, the technologist is attempting to

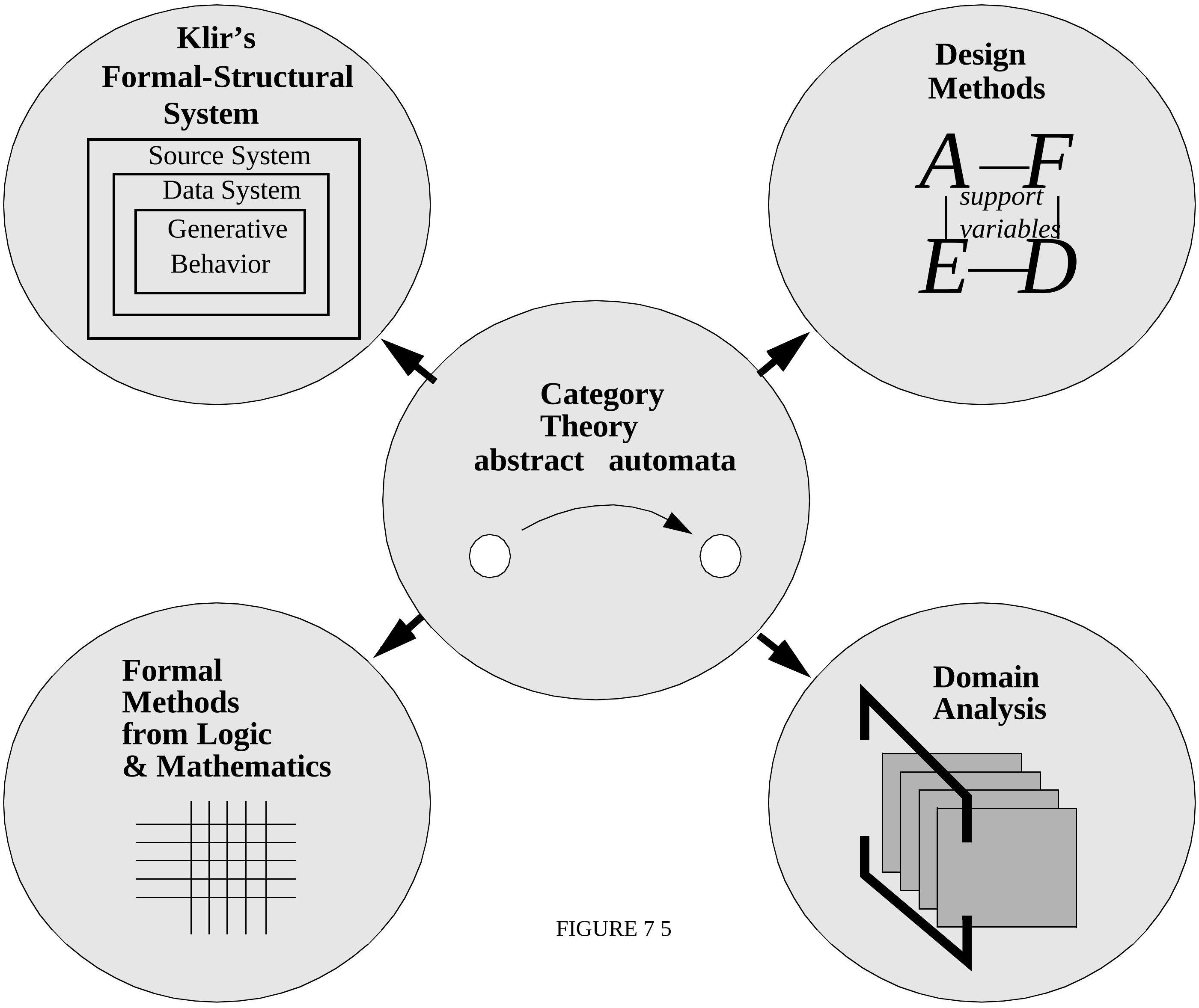


FIGURE 7 5

recognize it. This means the technologist is attempting to find significance within the field of artifacts. That significance is predicated on the recognition of genuine emergence. Without significance, then nihilism[69] takes over, and all the technical artifacts become indistinguishable. Even the most mundane technological decision to use one artifact over another must reflect the recognition of significance. As Bateson says, “differences that make a difference” are important.[70] The line separating genuine emergence from excrescences is the primordial significant difference from which all others flow. The technologist attempts to understand the whole field of emerging artifacts. Keeping up with this and distinguishing what is really useful among competing products, what is worth reading among the myriad articles, what is worth knowing from the avalanche of new books, what is important in terms of standards selection, etc., is a dizzying and endless quest. The recognition of what is significant in the face of information overload is a major problem. Only by knowing the background of excrescence can the figures of genuine

[69] See Bib#.

[70] See Bib# and Bib#.

emergence of novelty be seen.

Through genuine emergence new meaning enters the world. New meaning is constantly created, and this is the inverse of the “god is dead” scenario. This new meaning must be recognized and capitalized upon for it is the only antidote to the domination of mithraic active nihilism. This is what makes genuine emergence so important. It causes the segmentation of our own scientific tradition (paradigm shifts > Kuhn[71]), intellectual tradition(episteme changes > Foucault[72]), and philosophical tradition(Epochs of Being > Heidegger[73]). The point is that genuine emergence has a particular structure. That structure is written into the infra-structure of Western metaphysics. As genuinely new things come into existence from the void they pass through levels -4-5-6. In level -5, they encounter the fragmentation of Being. This encounter appears as the “thing” passing through each kind of Being as a phase of its becoming. It passes through these phases in the reverse order of their numerical ordering: Wild Being then Hyper Being, then Process Being,

[71] See Bib#

[72] See Bib#

[73] See Bib#

then finally Pure Presence. In each phase they are best described by a different mathematical formalism associated with each kind of Being: Chaos, then Fuzzy Sets, then Probability, then finally Determinate Calculus. These stages of emergence of the genuinely new distinguish them from artificial emergences which do not pass through all the stages. Instead artificial emergences are produced by the dynamic of the formal-structural system. The formal-structural system is not just a static representation, as Klir's work may suggest. Formal-structural systems are non-linear dynamic systems that appear embodied in society, and to a lesser extent in the artifacts produced by our society. A natural output of dynamic formal-structural systems is erratic change (disorderly motion). It is significant that if the erratic changes produced naturally by the eye are cancelled out, the eye cannot see.[74] The scene, moved perfectly in sync with the eye, just disappears. For a formal-structural system, erratic motion must be produced for the parts of the system to remain manifest or present. If erratic motion were to stop, the system would just vanish. Thus, a great deal of effort and energy goes into producing erratic

[74] See ??? {eye motion experiment}

change in every dynamic formal-structural system. Artificial emergence, or excrescence, is one of the main sources of erratic change in technological systems. By producing myriads of slightly different variations, the system creates its own temporality. This temporality allows the system to “see” its environment and also maintain its own presence. If the excrescence stops, the whole of the formal-structural system collapses. By maintaining its erratic changes, it appears neg-enthropic, producing structure as it feeds off the environment. As you can see, artificial emergence (excrescence) and genuine emergence are opposites. The former is a product of the dynamic formal-structural system, while the other is the passing through of ontologically distinct phases as an entity comes into existence from the Void. The field of artificial emergence provides the background on which appear the genuinely emergent entities. Without the background, the genuinely emergent entity could not be seen. Without genuine emergence, there would be no advent of meaning to sustain significance. And with no significance (differences that make a difference), then the nihilistic aspect of the formal-structural system would engulf it. This would be the same as if erratic change stopped -- it would just disappear. Thus, there is an inner connection between genuine emergence and the production of erratic change. They are nihilistic opposites. Stopping one is like stopping the other. Like all nihilistic opposites, they only appear to be opposites, but really they amount to the same thing. This gives us a hint that the formal-stuctural system and its ontological foundations are merely two aspects of the same thing. Unless we consider them both together, they can never be understood. This is because what is hidden in one of the nihilistic opposites is revealed in its twin.

Understanding the relationship between the formal-structural system and its ontological grounds is important because it leads to a deeper appreciation of the relation between man and technology. For the technologist (regardless of field, and everyone engaged in the technological system is sometimes engaged in the role of technologist), it gives a clear view of the phenomena of never-ending variation and change with which he must cope on a daily basis. The technologist is concerned with using new technologies to improve efficiencies in the technological system. But the plethora of new ideas and artifacts is overwhelming. The technologist must establish his own criteria for what is significant out of the onrush of new entities that confront him. This criteria may be grounded by the realization that most of the effort to produce “new” things is merely erratic change necessitated to keep the formal-structural system manifest. Occasionally, a genuine new entity will surface. Its hallmark is that it will go through each of the stages of manifestation grounded by the different kinds of Being. The technologist must watch for these crucial changes which will cause a revolution in the apprehension of the whole field of new technologies. The emergence of genuinely new things causes paradigm shifts and restructures the technological field. Paradigm shifts are an important phenomenon that has been recognized in philosophy of science, and which applies to all areas of human endeavor controlled by formal-structural systems. Paradigm shifts dictate the temporality of the technological field. They give a structure to our approach to that field. By watching the field for indications of those shifts, we can orient ourselves toward the whole realm of technology. This orientation involves the realization of the significance of the differences between different kinds of changes. This means appreciating the difference between nihilistic or non-significant differences that are the result of excrescence, and the non-nihilistic distinctions that indicate what is really “relevant” in Alfred Schultz’s sense.[75] What is relevant to the technologist is what will keep him tuned to the changes in the technological field overall as it changes and occasionally undergoes restructuring. The technologist finds himself lost in a constantly transforming landscape like that of Stanislav Lem’s SOLARIS.[76] He must use the restructuring of that landscape itself as his guideposts. This is achieved by recognizing the difference between genuine emergence (or the non-nihilistic distinction) and artificial emergence (or excrescence) which exemplifies nihilism and is produced by the formal-structural system as erratic change (disorderly motion[77]) in order to hold itself in manifestation. The formal-structural “remains in Being” or “clings to Being” through the manifestation of the inner structure of Being. This inner structure of Being is manifest in the passage of the genuine novel through each

[75] See Bib#.

[76] See ??? {Lem Solaris}

[77] See Bib# (Timeus)

phase of manifestation supported by a different kind of Being which can only be seen against the nihilistic background of excrescence. But also the inner structure of Being is manifested by the formal-structural system itself.

As a static representation, the formal-structural system incorporates Being^1 and Being^2. When it becomes dynamic, it begins to produce erratic change in order to keep itself visible. Where does this erratic change come from? The answer can only be that it is produced by the differing/deferring of Differ*a*nce at the event horizon that surrounds the singularity embodied within the formal-structural system. Erratic change, like Brownian motion, is constantly produced. It is the ground state of the non-linear dynamic system. In the production of erratic change, all the different types of mathematics contribute. Chaos, Fuzzy Sets, Stochastic and Determinate Continuity all contribute to producing truly erratic change. This erratic change, which exists as the ground state of the dynamic formal-structural system, is the manifestation of all the different kinds of Being working together to hold the formal-structural system in existence. Within the realm of artificial existence, the variety of arbitrary peculiarities are the result of the action of the erratic change produced by the formal-structural system. This erratic change exemplifies the collusion between the different types of Being, but it is particularly exemplary of the differing/deferring that masks pure immanence. In fact, erratic change of the ground state does not exhibit chaotic behavior. Chaotic behavior is exemplified at higher energy levels of the dynamic formal-structural system. Chaotic behavior becomes clear in the changes of structure that occur when the dynamic formal-structural system reaches higher energy levels, or when genuine emergence occurs. In the ground state, it is fuzziness that dominates erratic change. There are myriad possibilities for combinations of the characteristics of existing artifacts. The permutation of these characteristics to produce hybrid artifacts contributes the greatest amount of variability to the erratic change of produced technological artifacts. This has been recognized by Koestler,[78] Zwicky,[79] and others who have studied innovation and who attribute it to the chance association of unlikely characteristics of existent artifacts. Chaos, on

[78] See ??? {Koestler Act of Creation}

[79] See ??? {Zwicky Morphological Analysis}

the other hand, contributes to the restructuring of the whole field of excrescences as it moves to another structuring regime at the advent of the genuine emergent entity, or in the case of the paradigm shift.

◆ ◆ ◆

Since we are dealing with the particular area of Software Engineering Technology, it is proper to illustrate the foregoing abstract analysis of the relation between the formal-structural system and its ontological grounds with an example form this specific technological area. In order to facilitate this, let us analyze the relationship between embedded software and the characteristics of technology described by Fandozi in his study *Nihilism and Technology*.[80] By establishing the relation between software and technology, we automatically create a link between software and nihilism because nihilism is the essence of technology. Nihilism is the face of erratic change within the social system that produces technological artifacts. Nihilism, as a form of differing/deferring, masks the presence of pure immanence. Nihilism exists so that non-

[80] See Bib#

nihilistic distinctions may be made. This is the equivalent of the old Zoroastrian moral dichotomy good (Spenta Mainu) and evil (Angra Mainu), within the technological sphere. Just like evil and good, nihilism and the non-nihilistic distinction cannot exist and be seen without each other. They are complementary interdependent opposites.

Fandozi enumerates the following characteristics of technology:

Technology is *pervasive*.

Technology tends toward *autonomy* .

Technology is *repressive* .

Technology tends to *conceal its own nature* .

Technology is *anonymous* .

Technology *emasculates ideology* .

Technology attempts to *make everything available*.

Technology is a *process of formalizing and* *functionalizing the world* .

No one would deny that embedded software is a kind of technology. But how can we say that embedded software exhibits these characteristics of technology enumerated above? Embedded software represents a symbiosis of the old mechanical technology with the new meta-technical structures. In the development of this symbiosis, there is an intensification of the technical project. This transformation of the technical sphere in many ways clarifies its nature, bringing to the forefront these characteristics described by Fandozi.

Technology is pervasive. It effects all aspects of society. It effects what we do and how we do it in all spheres of life. It profoundly changes the lifeworld by altering structures of everyday behavior. Our structures of behavior are conditioned by the artifacts that we produce and that operate within our environment. These artifacts, such as telephones, radios, computers, automobiles, planes, etc. . have a synergistic compounding effect. They work together to create a complex technical environment which we work with everyday.

This interlocking of the aspects of the technical environment is what gives it a deep pervasiveness. Think of the launch of the Hubble telescope satellite. Here all the artifacts mentioned above, and many more, are used together, each fulfilling a particular function to bring about a successful launch. And what is it that allows this synergistic effect of combined technologies to occur besides human coordination? For the most part it is software embedded in various artifacts which allows the various technical artifacts to work together. The technical environment is knit together with embedded software so that different machines can work together efficiently. The deep pervasiveness of the interlocking technical environment is made possible by embedded software. That environment is symbiotic with the technical human community. It is a socio-technical system.[81] It is software that even further deepens that symbiosis. The technical environment forms a continuous whole within which the human being functions. The cockpit of a jet is a good example of this. It is software that makes the technical environment whole, and also adapts it to the human whose behavior is adapted to the technical system. Without

[81] See Bib#.

software, the pervasiveness of the technical environment remains superficial. The picture is of an environment filled with many different machines that do not work together except by human coordinated use. These machines are not adapted to the human, but instead, it is the human that must adapt to each individual machine. The deep pervasiveness allowed by embedded software occurs because the various machines can cooperate without human intervention, and because the machines can dynamically adapt themselves to the user.

It is thus clear that technology is pervasive, not just because wherever we turn we find ourselves falling over some machine. It is pervasive because of coordinated use of machines so that the whole work structure is determined by the necessities of coordinating machines. The human coordination of machines, as in a production line, is limited by natural human capacities. Software has the potential to supplement and take over some of the work of coordinating machines. This kind of software is called the Command and Control system.[82] Embedded software in each machine makes it able to be controlled, and for its

[82] See Bib#.

current state to be sensed. Embedded software, and command and control software, working together, allow a whole new level of adaptive and interlocking coordination far beyond what is humanly possible for complex technical environments. Thus, we see that software allows an intensification of the pervasiveness of the technical system by allowing adaptation and interconnection in ways that are not possible without software. This intensification of the pervasiveness of technology, through increased levels of coordination, may be deemed meta-technical.

Technology tends toward autonomy. This is to say that technology has its own agenda which supersedes that of user and maker of the technology. The agenda of technology is increased movement toward total control of everything, including the human that created or uses the technology. The Parable of the Tribes[83] expresses the dialectic out of which the autonomy of technology arises. If any tribe achieves a technological advantage and uses it against its neighbors, then all the tribes are forced to adopt that technological way of doing things or be dominated. Domination means the

[83] See Bib#.

loss of autonomy. Colonization is just this type of domination through technological advantage. Under domination, the adoption of the dominant technology becomes necessary for different reasons. Only those tribes who adapt to technological dominance by adopting the technology which gives advantage can remain autonomous. By this dialectical relation between competing tribes, technology begins to evolve. This mutation of techniques takes on the appearance of a Darwinian evolution. In fact, the Darwinian model fits technology far better than it fits biological evolution.[84] In biology, there is no mutation of one species to produce a wholly new species to be found in nature, even though this is demanded by the theory. There is only variance within species. The Darwinian model does not explain the discontinuous nature of biological evolution well. On the other hand, studies of the evolution of technology show that the Darwinian theory fits very well what happens when technologies evolve into newer technologies. The parable of the tribes is the story of the survival of -- not the fittest -- but those with the technological advantage. Over the long progress of human history, since the

[84] See Bib#.

Kurgen invasion of the whole of the known world in about 4000 BC, this technological advantage (in the case of the Kurgen people it was the use of the horse in warfare) has become recognized as the key element in economic and political warfare. Thus as time goes on, we recognize technology as an almost autonomous driver of human history.[85] It gives unfair advantage to those who are at the technological cutting edge of their times. Everyone contributes to the growth of technology whether they like it or not. It is an out-of-control positive feedback spiral.

However, this autonomy of technology, which arises from the struggle for power between peoples and nations, where everyone contributes to the growth of technology, so that the agenda of technology is raised above everyone's agenda -- this is only the superficial autonomy of technology. The deeper type of autonomy is that given to technology by the presence of computing machinery and the software that controls it and its peripherals. The autonomy of independent artificial processors is a truer autonomy. Machines have always acted by themselves via a

[85] See ??? {Technology & Warfare}

transformation of energy.[86] Feedback circuits have allowed machines to be self regulating. However, through the addition of computer control, machines achieve a degree of self-regulation which senses the environment and reacts in ways similar to an organism. This autonomy is set up by the human being because self-generation machines have not been produced yet. But once the machine has been set up, it can be programmed to sense its environment and react. Satellites are a good example of this sort of robot. Programming is a fundamental part of the “set up” for autonomy. It allows flexible adaptation and response of the robot. It is like a wind-up toy which is wound and then let go to walk about independently. Thus, programming manifests the slavery of the automated technical system to its programmers. But once programmed, the technological complex acts independently of the programmer; it appears in the environment as an autonomous independent agent. This is true even if the programmed system offers a control interface to the user. The user is controlling the autonomy of the programmed technical automaton. The automaton is still acting independently from the user. The point

[86] See Bib#

here is that the program user -- a technologically adapted human -- is part of the technological system. He is the wizard of OZ behind the curtain. He is the sophist or magician that hides behind his sophistry or sleight of hand. The autonomy of the technological system, as independent actor with its own agenda has appropriated the human slave. The master/slave dialectic of Hegel comes to play here, so that it is impossible to tell which is which. Technology has had its own agenda which has, through economic and political warfare been raised above the agendas of the players of the power game. Through the evolution of computing machines as a general autonomous processor, the technological system has itself achieved autonomous independence through which it can pursue it own agenda in a way hitherto not possible. The human being has been co-opted to serve the agenda of technology and provide it with the autonomy of movement and action as well. The autonomy of action given to technological products intensifies the application of the technological agenda to the world. Software is the mediator between the programmer -- slave/master -- and the automation -- master/slave. It is embedded software that makes the master/slave dialectic possible. Without software as a means of communication between automation and technical human, the actions of machines would remain disjointed and non-adaptive. Through software embedded in computing machines controlling other devices, technology begins to exhibit the kinds of independence which we attribute normally only to organisms. The autonomy of technology before was that of a preeminent cultural artifact. Now that artifact is automated and acting independently in the environment. The first killing of a human by a robot occurred in Japan not many years ago. Who is the murderer? An environment filled with autonomous technological artifacts is certainly different from one which is just filled machines of various types, closely watched by human manipulators. The autonomy of technology has deepened considerably with sensors and servo-mechanisms controlled by embedded software. The autonomy of the technological complex has deepened. This deepening, dependent on the presence of software and independent computing devices, may be described as meta-technical. The autonomy of the technological agenda has been supplemented with the true autonomy of the technological artifact.

Technology is repressive. This is a corollary of the surfacing of the technological agenda. Because the agenda of technology surfaces and comes to dominate the agendas of the players of the power game, all other agendas ultimately become submerged. The human diversity becomes subservient to the technological system and all features that are of no technical advantage are devalued. As such, they slowly become identified as hindrances to efficiency. This is happening to language barriers in the European Economic Community. These barriers cause joint technical projects to cost five times what the same project would cost by speakers of a single language. These human differences are inefficient, and thus, despite claims of cultural worth, English is becoming the technical language of choice everywhere in the world. The dominance of English is repressive because it causes other languages not to be used, and other kinds of human diversity connected to language become threatened. The success of American media that fosters English detracts from local cultural media products. Technology as a cultural system only accepts a very narrow band of human behavior. All other human behaviors are irrelevant to the technological society,[87] and as such, they face extinction.

This repression of technology, which rejects inefficient or noncontributory human diversity, is a surface phenomenon. In many places around the world, this repression takes on a more ominous character as the police state using technical weaponry against its own citizens. Political prisoners all over the world are held enthralled by the sinister face of technological gadgetry for eavesdropping, information collection, processing and control, as well as torture.[88] Most of this activity is aimed at controlling diverse human populations so they will be docile markets and work forces which do not threaten the stability of the dominant economic and political systems that foster the technological society. In many countries, the police state becomes identical with the technological system, and the sensing and information processing becomes essential control mechanisms. Without software, this machinery of repression could not blanket entire societies. Software is essential to the

[87] See Bib#
[88] See Bib#

expanded control of the police state. Yet, there were police states much longer than there were the elaborate technical apparatuses that they use today. In fact, the police state first appeared as the Catholic church used the Inquisition to stamp out the Cathar heresy.[89]

On a deeper level, repression of individual differences becomes the channeling of the human being into a technological mold. This is done by the education system.[90] Technical societies require a highly educated, docile population. This is why Japan is so successful as a technical society. American individualism was good to elevate the technical agenda through the great strides of early innovation. But a technical society must repress the individual differences so that individualism becomes inefficient in the long run. Education is a kind of programming of youth that makes them docile and amicable to serving the technological system. It teaches them the doctrine of relativism which disarms all independent thought and action. Relativism is the positive face of nihilism. It is nihilism whose negative aspects have not yet become

[89] See Bib#
[90] See Bib#

apparent. Relativism is the equivalent of religious dogma for the technocrats of the technological society. Education is an early repression of individual differences in order to mold good managers and workers.

Slowly, computers are entering education curriculums. Computer-based education materials are starting to be developed. Educational software is an expanding field. This kind of software makes learning without teachers or books possible. The educational program lets self-paced learning occur. It allows training to take place in a simulated environment. This type of software for teaching shows a completely different aspect of the usefulness of software. It is universally regarded as a positive and fruitful aspect of software development. No one sees educational software as repressive. It is the means of turning talents into skills.

“Repressiveness” is a word with negative connotations. A better word might be “channeling.” Technology causes <u>channeling</u>. It channels the diverse human populations into a narrow range of alternatives. Those who resist this channeling are either controlled by the police state or join the ranks of the unemployed. Channeling, when directed at the young, is called education. Software takes educational channeling to a new level of sophistication. Education becomes a game. Education becomes adapted to individual needs and differences, allowing one-to-one teaching again by automated means. Computerized education is a channeling environment which prepares the student for control by making him docile in a computer-controlled environment. Thus, educational software allows us to see the deeper nature of technological repression. This repression of human differences is inherently meta-technical.

Technology is anonymous and tends to conceal its own nature. We do not think of ourselves as living in a culture controlled by technology, even though the artificial environment is everywhere around us. We see the artificial environment as a means to furthering human ends. Human actors that own and control technological means are what we see around us. The technological equipment are stage props to the actions of other humans. This is merely another way of saying that technology is not present-at-hand. It is not the focus of our projects unless we are building and maintaining it. Because it is usually not present-at-hand, it fades from view no matter how it clutters the environment. But this concealing of itself is but a prelude to its concealment of its own nature. The nature of things are their essential properties. Technology not only hides itself, but it hides what it is really like from us. This is why technology has been part of human life for thousands of years, but it has not been focused upon by human society, except only exceptionally. Our era is one of those exceptions. Technology blends into human action. We see the actions and not the technical means. Humans rarely realized that the technical means had any nature different from the enabled actions. It is only in eras of great technological change that the distinction between the way we do things and the technical means can be made. Thus, the attributes of technology discussed here appear. They are normally hidden. It is only with great difficulty that they have been discerned by philosophers over the centuries. This concealment of “technical essence” or “technical nature” is itself an attribute of the technical. We might call this a meta-attribute of technology. Through the meta-attribute of essence concealment, appears the next level of Being. That has been described in the first section of this essay as Hyper Being. It is the absolute concealment of pure immanence. Software takes its nature from this meta-level of Being. Self concealment of attributes is a meta-technological manifestation. With preceding attributes, the surface phenomenon of technology has been differentiated from the deep meta-technical phenomenon. "Self concealment," as opposed to the "concealment of nature" are two such levels. Fandozi recognizes both of these levels. He speaks of the anonymity of technology. This is self concealment of what is not present-at-hand. The concealment of concealment is the associated meta-technical attribute. Both of these apply directly to software. Software is almost never seen directly. Normally only its effects are seen. Embedded software even lacks a user interface. It is hidden in the bowels of the machinery. The user might not even know it is there. This software, unlike other kinds of writing, is nearly invisible to everyone except the programmer. When it is seen, it is very difficult to understand. Software is by nature hidden. The elegance of software designs are rarely appreciated. Besides hiding itself, software hides its nature. Software appears as “just a text.” Yet it hides through the operation of multiple perspectives the singularity of pure immanence beyond the event horizon of Derrida’s differing/deferring of differ*a*nce. Probably no one would agree that software has this inner nature. The nature is hidden. We only see it in our nightmares, as yet another project fails, as a werewolf for which we seek an elusive silver bullet.[91] So technological anonymity is the surface phenomenon that glosses over the meta-technical manifestation of technology hiding its own nature.

Technology emasculates ideology. Human differences are not the only kind that are repressed. Technology also represses ideological differences. Ideologies are the motive forces behind political systems. They occur when a particular set of ideas like freedom, liberty and fraternity become the central rationales of behavior. So it is interesting to note that although the agenda of technology is raised above all others in order to give power to the tribe or state, the state is ultimately a roadblock which must be removed

[91] See Bib#

by “progress.”[92] Multinational companies express the agenda of technology more perfectly. The corporation is an imaginary person created as a legal fiction. Imaginary persons have no need of ideology. Their behavior is not motivated. Only human beings need motivation. The corporation is the equivalent of the self for the technological system. It is a vortex of human activity, aided by technical means, which forms a system producing surplus value.[93] The corporation is the point at which the economic and technical systems merge. By becoming a legal fictional entity, it is recognized by the state. But the state only serves as a host for the corporation. Its ideology that motivates the political system is not necessary for the technological system. In fact, as the agenda of technology is raised -- and that agenda is the intensification of its attributes -- these human motivators become inefficient, like other human differences. The Soviets are slowly recognizing how their ideology is counter-productive. They are losing technical advantage because they insist on ideological control. Free flow of information and open markets with free

[92] See Bib#

[93] See Bib#

movement of people is more conducive to the flourishing of the technological system. The police state is ultimately counter-productive. However, the emasculation of ideology is a surface phenomenon. The deep phenomenon is the ideational phenomenon that produces ideologies. Ideologies are political theories. At a deeper level technology harnesses ideation but devalues the product of ideation which are ideas. Technology is practical. It functions by using the ideational process to produce theories. These theories are embedded in machinery[94] for practical use. The pure ideas are discarded. Technology dwells in the realm of pragmatic intelligence defined by Kant practical reason. The discarding of the pure ideas is what separates science from technology. Science is the opposite of technology in this respect. Science reveres the pure ideas and is only interested in technology as a means of getting at those pure ideas. There is a natural partnership between science and technology because the end products of ideation are useless to technology. Technology uses the ideational process itself. It creates theories embedded in machines rather than free floating -- nonembedded -- theories. An

[94] See Bib#

example of an embedded theory is a software design. As shown before in the first part, software is a nonrepresentable theory which serves as a software design. Software design depends on ideation -- which produces illusory continuity -- to give wholeness to the design as it evolves. Without ideation as a capability of the mind to produce a differentiated theory and do mental simulations, the theory would not be a whole. The computing machine is an artifact that externalizes the ideational process. The software artifact, when represented as code, executes at such a speed that an illusory continuity for the processors actions is created. As seen in part one this illusory continuity is crucial to the manifestation of the software design. So at a deeper level technology rejects ideas (the products of ideation) for the process of ideation, which in software, is externalized in the form of the running computer animated by software.

Technology makes everything available. The anonymity of software shows it is not present-at-hand. But this attribute attaches it definitely to the ready-to-hand. “Made available” is equivalent to Being ready-to-hand. Total availability means ready to use to get some project we are focused on done. Unavailable is a hindrance to the technological project. Our conditioning to this attribute is shown by anger at the slightest inconvenience. Availability means the free flow of information, people and goods for immediate consumption. Any barriers to this flow is unacceptable. This free flow shows the likeness of technology to fire which consumes its fuel. The fire consumes resources and lives to produce artifacts which are consumed by the market. The fire is the vortex of activity within the corporation which feeds the frenzy of consumption within the free economy. Free economy means open to total availability. Technology strives to make everything available which fits into the technological system, including itself. Technology propagates its own availability even as it hides itself and its nature. This contradictory set of attributes gives us a hint that technology has its own mode of disclosing. It hides in relation to the present-at-hand, but becomes conspicuous in the ready-to-hand. As another mode of disclosing, we see that technology functions in a different kind of Being. Being is manifestation. Different kinds of manifestation will allow hiding in one mode with exposure in another. There is no meta-technical level associated with this attribute. Technology takes its Being from Heidegger's strange mixture of Being and Time called "Process Being."

Technology formalizes and functionalizes the world. Technology is structural and is best represented by the formal-structural system such as that described by Klir. Technology can formalize and functionalize precisely because it is structural. "Structural" means it has the ability to structure. To structure is to impose form. Functionality is an attribute of form. Structure determines the functionality of a form within a set of forms that can be transformed into each other. Functionality is the relation of a form from a structured set of forms to the whole set. Structuralism handles the discontinuities between the forms of this set. Structuralism builds bridges between forms across these discontinuities. The forms evolve through time, changing into each other. If this process is fast enough, the change of forms appears continuous, and then this is a model of ideation. Form and function is the means by which we understand what is present-at-hand. We concentrate on forms, and apply our functions to transform them. This transformation changes the functions of the form. If that change is radical enough, they are replaced by another structurally-related form. Technology can only formalize because its nature is structural. This means that technology operates on the present-at-hand forms with their functions from the hidden space of the ready-to-hand. If it were not once removed in another mode of Being, it could not operate on forms. Thus, this attribute of technology connects it yet again closely to Process Being.

Software is a meta-technology. Where technology makes everything available and formalizes/functionalizes the world, software does that and more. Meta-technology intensifies technology. Meta-technology “technologizes” technology itself. This becomes apparent in the way computers make math skills more necessary, while at the same time taking away the tedium of the computation. It is no longer necessary to be able to do the computation. Yet is it necessary to know what the end product of the computation means. Without this knowledge “garbage in-garbage out” becomes more than a truism. It describes precisely the situation where there is reams of overly precise data which when processed, is meaningless. When everything is available, then relevance, significance and priority becomes crucial. Thus, education becomes more important. One can no longer do production jobs without basic computer and thus mathematical skills. Thus, while technology makes available, the question becomes "available for what?" What do we do with what has been made available to us? What are the significant facts in our data base? How do we find out? Meta-availability is relevance. Meta-availability is filtering. Software provides the means of automatic filtering of information. It can automatically signal the arrival of an important fact while filtering out irrelevant information. So while software makes more information available as a technology, it also allows filtering of what is available which is a meta-technical aspect that allows the true nature of software to be glimpsed. All we need to do is look at the evolution of abstracting journals to see this exemplified. Now the references of articles are cross-referenced to discern related articles. This has been taken another step where clusters of heavily referenced articles are identified and named as cutting edge subdisciplines. The identification of cutting edge subdisciplines allows a reader to immediately identify the crucial papers published each year by seeing how heavily referenced they are and whether they belong to a cluster of heavily referenced papers. This is a kind of information filtering which should intensify research and the dissemination of important findings.

Software plays a part in the formalization and functionalization of the world. Yet this has already been started by other technologies that foster the creation of formal-structural systems in the world. Software, as a meta-technology, must do more than merely formalize and functionalize. It must intensify the formalization and functionalization in some way to qualify as a meta-technology. The intensification of formalization occurs through the operationalization of formal languages. Formal languages are used to write software modules called functions that take inputs and transform them into outputs. Thus, in some way formalization and functionalization is epitomized by software. Software must have its input in a certain form to make use of it. Each piece of software fulfills a specific function. It demands a certain set of formalized behaviors on the part of the user in order to work properly. Installation must be done exactly right. The program accepts a certain limited range of user behaviors and responds consistently to the instilled behavior patterns associated with its user interface. Software simulates the world, and by running its simulation within an environment, it sets the pace and conditions the environment. So software allows the formalization and functionalization of the world to be automated. The automation is a simulation of some aspect of the world which ends up driving the world and causes behaviors of people to change to accommodate the simulation. By epitomizing formalization and functionalization, and confronting individuals in everyday life with formal and functional responses, it causes those individuals to change their behavior and conform. The result of conformity is that the individual enters into the simulated or artificial environment of the complex dynamic software artifact.[95] Our vision of TRON[96] being sucked into the computer is becoming fundamental mythology that haunts everyone who fails to get a good

[95] See Bib#

[96] The romantic Walt Disney movie about the hacker that is sucked into the computer and forced to play computer games in virtual space like a gladiator..

credit rating. The software has a replica of us which, in some sense, controls our basic economic behavior. The stock trader who does computer trading has an active replica -- called, strangely enough, a daemon[97] -- that watches for the price of a stock to change beyond a certain threshold, and it automatically buys or sells based on that movement. Software allows us to enter the simulated artificial world, and that artificial world get mixed up with the tangible world of everyday life. The end result is a confusion about what is real. Fortunes are made and lost, not on paper, but within the electronic media of the computer circuits. Software makes the world a simulation. Through this analysis of Fandozi's attributes of technology, we have defined a set of meta-technological attributes related specifically to the role of software in the world today.

The meta-technical (like software) is not just pervasive, but interlocking and adaptive; not just tending toward autonomy, but truly autonomous; not just repressive, but channeling; not just concealing (anonymity) its nature, but hiding a singularity of pure immanence; not just emasculating ideology,

[97] See Bib# for the origins of this term.

but harnessing ideation; not just making all available, but filtering for relevance; not just formalization and functionalization, but simulating in an artificial environment.

Recognizing the meta-technical in the midst of the technological milieu is very difficult. Yet it is necessary in order to respond appropriately. The meta-technical, as exemplified by software, is an interlocking and adaptive environment composed of truly autonomous "daemons" which channels behavior by harnessing ideation and filtering relevant information to produce a simulated artificial reality that hides a singularity of pure immanence. On the other hand, the technological exemplified by the formal-structural system is pervasive yet anonymous. It is repressive and emasculates ideology. It tends toward autonomy while making everything available. It conceals its nature while pursuing the agenda of formalizing and functionalizing the world.

Technology is intensified and guided by the meta-technical. Yet at the heart of the meta-technical is a phenomenon of cancellation. The meta-technology is the limit of technology.[98]

Upon reaching that limit, technology turns into its opposite which is the proto-technology of Wild Being.[99] This is exemplified by the rise of interest in acupuncture[100] and homeopathy[101] and other types of "alternative traditional" medicines.[102] The meta-technical is like a veil which encounters the limits of technology and sees those limits clearly.[103] From this encounter comes alternative or appropriate technologies. As one passes through those limits, there then appears proto-technologies which harken back to archaic technologies. In our time, each of these approaches to technology are flourishing side by side. Our culture as a whole favors technological means until the side effects become unbearable. Then small is suddenly beautiful,[104] and we opt for appropriate alternative technologies.[105] Thus, we all use leaded fuel until it starts to appear in cow's milk which causes it to go into our bodies from which it does not emerge. Then suddenly we are environmentalists attempting to manage the exploitation and pollution of the environment

[98] See Bib#

[99] See Bib#

[100] See Bib#

[101] See Bib# and Bib#

[102] See Bib#

[103] See Bib#

[104] See Bib#

[105] See The New Alchemists

enough so that it does not effect us. When even management fails, we then become proto-technologists attempting not just to find alternatives, but to go back to archaic technologies that were hitherto rejected alternatives.[106] No one tries acupuncture until all other forms of medicine fail. It is a last resort. In ancient China it used to be the first resort.[107] When drugs fail, we try stress reduction and other health maintenance strategies. When these fail, we attempt to learn what the ancients know that we have forgotten. This harkening back to Shamanism,[108] which is now called "New Age Thought," is a reconstruction and not the original.[109] The original has been lost, and only the archeological remains are left.[110] The proto-technical is a kind of nostalgia for what has been lost.[111]

In software engineering, we encounter the limits of technology by dealing with complex representations of systems we cannot see, which are nonintuitive and may act in counter-intuitive fashions which are hard to explain.

[106] See Bib#
[107] See Bib# & Bib#
[108] See Bib#
[109] See Bib# & Bib#
[110] See Bib#
[111] See Bib# & Bib#

We work in an environment where our tools are software artifacts that help us build software artifacts. These software artifacts are simulations of parts of the world which must be simulated themselves in order to know if they work correctly. This simulation is done with test software which simulates the environment of the simulation to see whether the built software will track what is happening in the environment correctly. The software simulates without actually knowing anything about the environment. Whatever knowledge is not coded into the software product is thrown away. The attempt to regain and use this discarded knowledge is called “knowledge engineering.” Knowledge engineering is proto-technical in that it attempts to get at the knowledge in the human being that allows him to know what is significant. It is an attempt to code the knowledge into a software artifact that will not forget it. Software forgets knowledge and preserves behavior. Knowledgeware sacrifices behavioral coherence for knowledge preservation. Passing through the event horizon of differing/deferring, the meta-technical realm collapses. The singularity is not seen, but disappears. What is left after the collapse is the proto-technical. In software, this is the knowledge of the builder of the software. Automatic code generation is a proto-technical activity. Software reuse is a proto-technical activity. All activities which attempt to transcend the “software problem” are inherently proto-technical and deal with knowledge acquisition, preservation, use. Software is trapped at the behavioral level. It ultimately lacks the knowledge that would allow complete adaptation to its environment. It is the human expert that best exemplifies this total adaptation to the environment. In the proto-technical state, software attempts to become its own designer by replacing the expert designer. Software has become too complex for the human being. Now software defines software. The human reenters the arena of ignorance about software. Everyone is a user. The software engineer merely uses the software construction tools rather than their end products. At that point, we will have passed through the barrier of software which was too difficult for humans to handle directly. Software, with expert design knowledge, must handle software. Software is in this way destined to become a non-human artifact. Humans built software. Software extracted the knowledge of how it was built. It then started generating itself, and humans became excluded because it was too complex and costly for humans to handle the building process. This sounds like a science fiction scenario. Yet already code generating systems have been shown effective on simple problems. It is the nature of the meta-technical to collapse and push those who experience the limits of technology beyond technology into the proto-technological realm. In this, we do not solve the problem of software because *we* are the problem. When we are excluded and pushed into the proto-technical realm, *we*, as a problem, have been solved by the technical system anonymously seeking autonomy. We have met the software problem, and it was us -- our incapacities and our limitations as human beings.

◆ ◆ ◆

The essence of technology is nihilism. This essence can be seen in the intensification of the attributes of technology in the guise of the meta-technical. Thus, the attributes of the meta-technical must be the attributes of nihilism which is the summary of this inner core of technology that is exemplified by software. Nihilism has two aspects. There is the passive aspect which occurs when nihilistic opposites are produced that conceptually lead to cancellation. This cancellation of nihilistic opposites causes an enthropic loss of meaning in the world. When nihilism becomes active, meaning is destroyed purposely. Active nihilism can properly be characterized as Mithraic and Manicheistic (i.e., heretical) and ultimately rooted in Zoroastrianism. Its roots are deep in history, and this is a virulent disease to which Western culture is all too prone. In the realm of the meta-technical, we are dealing only with passive nihilism which has been called previously excrescence (abnormal growth) or artificial emergence. This is a dynamic of the automated formal-structural system by which change is manufactured. This manufactured erratic change creates the possibility of seeing the formal-structural system within its environment. This has been called minimal change or disorderly motion.

Without this type of basic erratic change (a kind of Brownean motion for systems), no other changes would be visible. Since we are usually only interested in the changes made visible, this type of minimal change of formal-structural systems normally goes unnoticed. In software, this minimal change is seen in such phenomena as software bugs, designs thrashing between alternatives which are both bad for different reasons, and contradictory requirements. These are all phenomena that are endemic to software where changes occur which make other changes in which we are more interested invisible. It is the attributes of the meta-technical that make excrescence occur in the technical arena. Excrescence allows the formal-structural system to be seen on a ground of many different systems. Excrescence is a phenomenon located at the event horizon, surrounding the singularity of pure immanence. At that event horizon there is a constant shifting between possible states of the formal-structural system. It is constantly trying new permutations of its status that will allow greater interlocking or closer adaptability. This is an autonomous channeling function that acts as a teleononic filter as described by Monod.[112] It channels the system into narrower and narrower evolutionary changes in the interaction of the system with its environment. This excrescence appears as bursts of ideational relevance called newness. The

[112] See Bib#

novelty is purely artificial. It calls to the fore new figure/ground relations without changing the essential structure of relations within the overall gestalt. The scintillation of the event horizon where this phenomena is generated is the very stuff of which the artificially simulated world is made. Each spark of scintillation lights up the darkness of Plato's cave. The sophists (sometimes called programmers) carry chairs and everyday objects (design elements) which cast shadows (sometimes called graphical interfaces). The world behind the glass of the computer screen is more alluring than that behind department store windows. The former moves by firing an electron gun at us. The activated phosphor dot takes a short time to fade. This shower of sparks in our direction is just enough of an indication to produce all sorts of imagined images upon which our preconceived ideas are projected. Ideation uses the illusory continuity of the moving display image as a springboard for its activity. We work all day with computers, then relax by watching TV or going to a movie. We enter wholly into the media-simulated environment. Its power to intrigue us is the ability to constantly produce slightly modified stimulation to which we become addicted. If the stimulation wasn't always slightly different, we would quickly lose interest. Interest is stimulated relevance produced as excrescence by a dynamically permutating formal-structural system. When we lose interest because the image in the illusion is no longer relevant, then we break out, momentarily, from the confines of the cave. Real people and things are not nearly as interesting as the simulated world. In real life, if you wreck the car, it costs a bundle. In the simulation, you lose a few points. In the simulation, it is possible to do all the interesting exceptional things we avoid in the "real world." But when the simulator is hooked up to the "real controls," different rules apply even though it may be hard to tell the difference because the interface is virtually or actually the same.

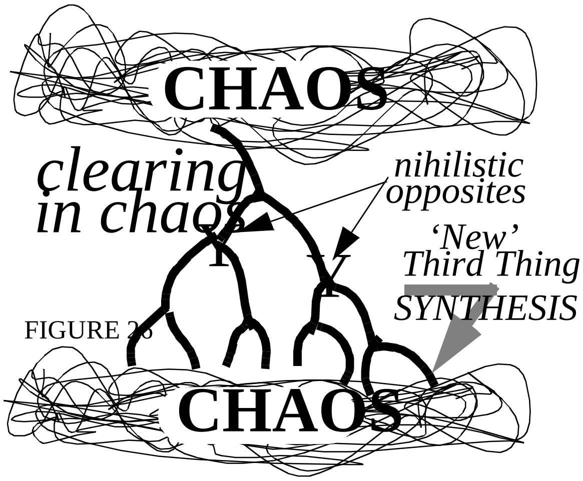


FIGURE 26

When the point of cancellation is passed, the erratic changes turn into chaos.[113] This is, in effect, raising the formal-structural system to higher energy levels in which its system states proliferate by bifurcation. The formal structural system produces nihilistic (i.e., self-cancelling) opposites. The self-cancelling opposites appear as bifurcation states embedded within chaos. Chaos occurs whenever a third thing is present to disturb the symmetry of the bifurcations. Each bifurcation represents a nihilistic opposite. Even within chaos, there occur clear spaces where the chaos abates and the bifurcation appears clearly again before it is again overwhelmed by chaos. The arising of nihilistic opposites out of the chaos of images clears the way for the third thing to appear, yet again as the embodiment of excrescence. The manifestation of the third thing breaks the bifurcation, and it is immediately overwhelmed again by chaotic states. In our culture, the production of simplistic self-cancelling opposite positions for the manipulation of “media-ignorant” masses has become a high art. These nihilistic opposites appear to solve the problems of the complex chaotic environment, giving a false

[113] See Bib# & Bib#

sense of clarity. This false clarity of bifurcations arising from the chaos only appears to be broken by the arising of a third position dialectically related to the nihilistic opposites. This dialectical synthesis appears to be a “new thing.” but in fact it is the advent of a new wave of chaos which will gives rise yet again to the production of false opposites. This dialectic within the chaotic field, moving from chaos to nihilistic opposites to the advent of the “new” synthesis of a third thing which brings again chaos, is the nature of a chaotically bound system. The chaotic substructure forms the basis of erratic change upon which the fuzzy, probabilistic and deterministic layers are built.

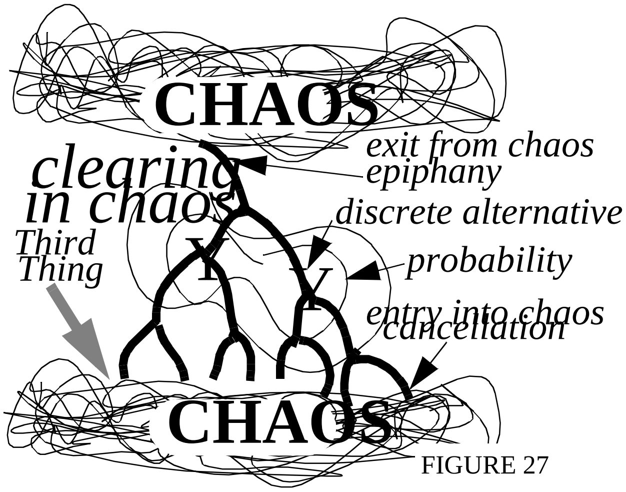


FIGURE 27

Understanding how nihilism and erratic change manifest from the chaotic substrata is important. The meta-technical arises from the proto-technical and brings possibilities into play adding to the chaotic dynamics. Possibilities are turned into actualizations that are properly stochastic by the action of chaotic dialectics.

The chaotic field represents a myriad of possible states that the system might assume. The arising of the third thing causes the system to leave the regime of bifurcation and reenter chaos. This is related to the point where chaos clears and a single state arises, which leads to bifurcation again. The clearing of the chaos and the arising of a third thing to bring back chaos, are related possibilities. This is again like the arising and cancellation of the virtual particle discussed in part one. There it was seen how the different kinds of Being participated in the manifestation of this theoretical complex made necessary by the infinite energy of space-time.

The exit from the chaos is an epiphany which is like a paradigm shift where a single way of looking at the world suddenly becomes dominant. This single track cannot maintain itself without splitting into nihilistic self-cancelling opposite states. The system will oscillate in these until suddenly the two positions themselves bifurcate. As bifurcation continues, the set of nihilistic opposites begins to have structure. The nihilistic opposites appear to arise out of nowhere. They are possibilities being actualized. They appear upon the background of chaos. They arise as figures on that ground, and the dynamic formal-structural system randomly pops from one nihilistic state to another. Thus, the actualization of possibilities includes the arising of the possibility out of the infinite possibilities of chaos and the random entry and exit of the possible states which have arisen.

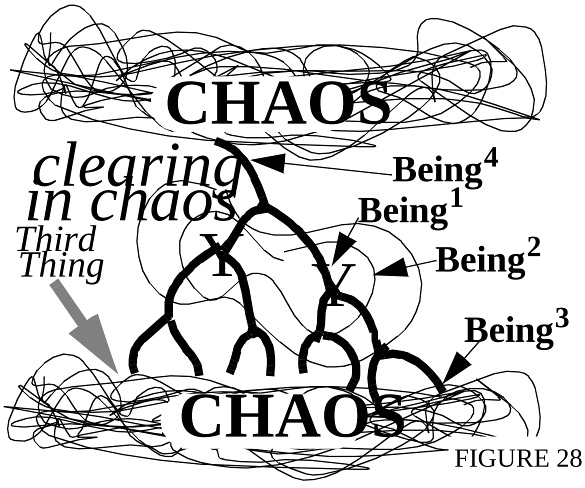
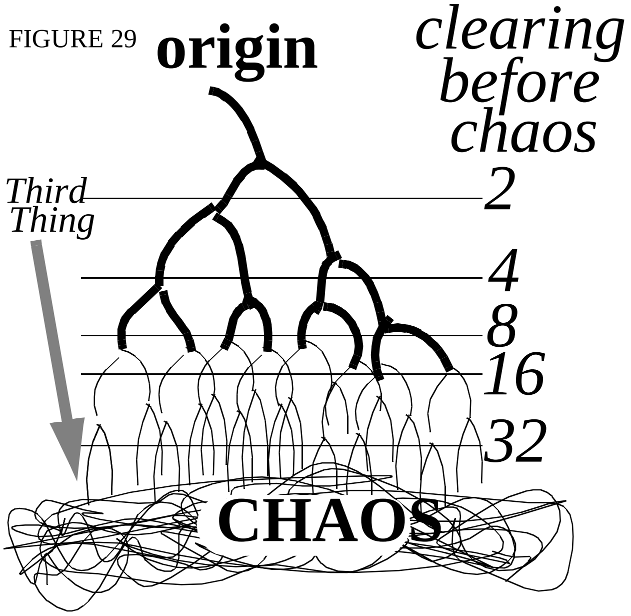


FIGURE 28

Being^1 is the current determinate state of the system. Being^2 is the probabilistic movement of the system between its various actualized states. Being^{3} is the arising of actualized states from the chaotic background. The reentry of the system into chaos is also a function of Being^{3}. It represents the cancellation of nihilistic opposites and their being overwhelmed by chaos. Being^{4} appears as the epiphany at exit from chaos and as the arising of a third thing (synthesis) which causes reentry into chaos.

FIGURE 29



The formal-structural system has its basis within the clearing of abated chaos. It arises as the infrastructure of bifurcated states. Bifurcations have a specific structure which allows the formalization of the relations between bifurcated states. This formal-structure allows the discontinuities between dynamical states to be bridged. Formalisms embody the image of a particular state. Structuralism comprehends the transmutation from dynamic state to dynamic state within the range of possible states. Formalism and structuralism arise within the abated chaos, perhaps analogous to what Heidegger calls the "clearing-in-being." The formal-structural system that describes the bifurcation states of the dynamical system has its mathematical basis in topoi, which are sheaves of boolean systems. It has been shown that there is an inherent relation between topoi and fuzzy systems.[114] Computing devices make use of this same boolean mathematical substructure as the electronic basis of software. Software exists in the boolean space, and makes use of the structure of boolean systems to model more abstract conceptual forms. Thus, there is a direct relation between bifurcation spaces and boolean discrete mathematical spaces where bit patterns model images of system states. In both cases, it is the substructure of binary systems which is the foundation of formal structural systems. This is an important perspective because it shows the deep relationship between the formal-structural system and both the

[114] See ??? {N?}

bifurcation space within chaotic dynamics and the boolean mathematical traces upon which the semiotics[115] of software is based.

As an aside, it is important to note that we cannot see past the veil of chaos which hides the origin of the dynamical formal-structural system. This origin is always hidden. Proto-technology and proto-science constantly attempt to piece together the archeological evidence and reconstruct the original archaic science beyond the veil of history. Proto-science and proto-technology is a nostalgic attempt to regain the past which is doomed to failure because the chaotic veil can never be lifted to see the "primal scene"[116] of the true origin. Thus, the efforts of cultural anthropologists to reconstruct the archaic science and technology as a crude reflection of our own science and technology as Levi Strauss has done, in the end only emphasizes our own failures. There is, however, another route to the archeology of archaic knowledge before the arising of chaos. Taking this route necessitates a radical reappraisal of our own superiority in relation to other intellectual

[115] See Bib#

[116] See Bib#

traditions. The fact is that the Western intellectual tradition is a diversion from the archaic traditions which it cannot understand because it refuses to see its own inherent inferiority. This blindness amounts to a hubris which is always ultimately punished by the "gods." An excellent portrayal of this hubris, in relation to the archaic medical tradition of China, is given by Bruce Holbrook in the Stone Monkey.[117] The attempt to understand archaic sciences in their own terms[118] rather than as nostalgic proto-sciences is just the beginning. In order to participate in this adventure, it is necessary to be prepared to venture outside Being. For the descendents of the Indo-European Kurgen people, who are still enthralled in that subtle clinging, this is very difficult.

This excursion into the depths of technology is relevant because software methodologists are also technologists. We must deal with technology and learn to recognize its intensification into meta-technical facets. This perspective causes us to approach the definition of the formal-structural system a little less

[117] See Bib#
[118] See Bib#

naively than we might have otherwise. We bring back from the philosophical substrata some insights into the significance of the formal-structural system that we can apply to understanding how software (meta-technical) perspectives alter our view of the formal-structural system (technical) as defined so succinctly by Klir.

◆ ◆ ◆

In the foregoing, we have discussed in-depth the substrata upon which the formal-structural system is based. Now we will attempt to enter a particularly robust static representation of a formal-structural system provided by George Klir. We will enter, like visitors, into a medieval cathedral. As visitors, we cannot study every detail, but only get an overall impression. The formal-structural system is the cathedral of the Twentieth Century Western intellectual tradition. Unlike Chartres, it is not built of stone. Instead it is an ideational structure upon which all of Western science and technology is founded. Like medieval cathedrals, there are many different versions of the formal-structural system. We have selected one with the most parts intact. Tourists go to

Chartres because the original glass is intact, and we can see the relation between the images in stone and the images in glass as the medieval peasant might well have done. In this case, the stonework is like the technological system, and the patterns in glass are like the scientific super structure. In the cathedral, the local interpretation of scripture was read off the walls and windows by the illiterate. In the formal-structured system, each discipline has its own variations on the same themes. The technologically illiterate are dazzled by the spectacle of the high priests. As Nietzsche says, the “last man” blinks. In later ages, these different kinds of formal-structural system representations may be studied by intellectual historians, like art historians study the cathedrals. When they were built too high, cathedrals collapsed. High Gothic had its limits; so too, these intellectual edifices will eventually realize their own inherent limitations. Until then, they are the main means for understanding everything in the world. Unfortunately, they are built on the shifting sands of fragmented Being. Thus, it is possible for the entire edifice to collapse at any moment. So we had better make our tour quickly before any tremor might occur.

A quick overview reveals that the edifice of this particular structural system contains several layers called epistemological levels. The first level is the object layer which is the system raised to a figure against the ground of all possible systems. This figure has a set of attributes, and it is the relation between these perceived attributes that constitutes the system. Attributes are compared to what Klir calls backdrops -- the name for very general environmental attributes whose values vary continuously systemwide. The object system obviously entails the subject/object dichotomy, and all that this implies in traditional philosophical discourse. In other words, the object system is assumed by Klir to be present-at-hand. The source system is a general representation of the object system prepared for study. Attributes become variables and, “backdrops” become “supports.” In other words, observational channels are set up for studying the object system. This instrumentation for objective measurement becomes the source for data about the object system. The source system is the experimental set-up. It becomes the data system when measurement, according to scientific protocol, begins. When enough data has been taken and analyzed so that the rules governing data generation have been discovered, then the data system may be compared to the generative system that simulates the object system. The generative system is often composed of different simulating models which work at different times. The models may generate data adequately when the dynamical system is in a particular dynamic bifurcation state. When the dynamic system pops to another bifurcation state, the model must be changed. Models are formal and generally generate linear behavior that simulates the system under study. Changing models brings out the relation between models which is the structure underlying the system. Note that structure handles discontinuous changes in the data. Nonlinearity is mapped into structural concepts. This is exactly the relation between the formal and structural aspects of the formal-structural system. Meta-levels of relations between relations form an infinite regress of finer and finer structural systems modeling. In fact, meta-systems and structure-systems regress infinitely, forming two horns that arise from the basic source, data and generative systems. The meta-systems are an infinite progression of models-of models-of models .

> . . Whereas the structure systems form an infinite progression of relations-between relations-between relations. . . Klir has well represented the relations between these two infinite regresses which always occur to purely formal systems of the type first created by Russell and Whitehead in Principia Mathematica.[119]

Each bifurcation creates another meta/ structural level of analysis. Eventually these become impractical to understand, and the formal structural system, with too many meta-levels, self destructs as it is overwhelmed by chaos. A model is necessary for each bifurcation state. When a bifurcation state bifurcates again, a meta-model must be abstracted to cover both models covering the new bifurcation states. The meta-model explains how the two new states relate to the old single model. Models are composed of entities and their relations. Structural relations between models allow the explanation of discontinuous jumps between submodels. Each level of bifurcation demands another infra-structural level to explain the new sets of discontinuities between models. Thus,

[119] See Bib#.

structure represents the increase of fragmentation, whereas meta-levels represent the increasing levels of abstraction necessary to hold the system together as a single object. Meta-levels represent analysis. Bifurcation drives the need for ever deeper levels of synthesis and analysis. Ultimately, the human mind, even with the aid of computers, can only stand a few of these meta levels of embedding. As bifurcation progresses, the analyst is quickly driven beyond his capacity to model, and the whole formal-structural system cracks up. The whole representation cancels itself out, and one is forced to start over looking for a new, more robust model.

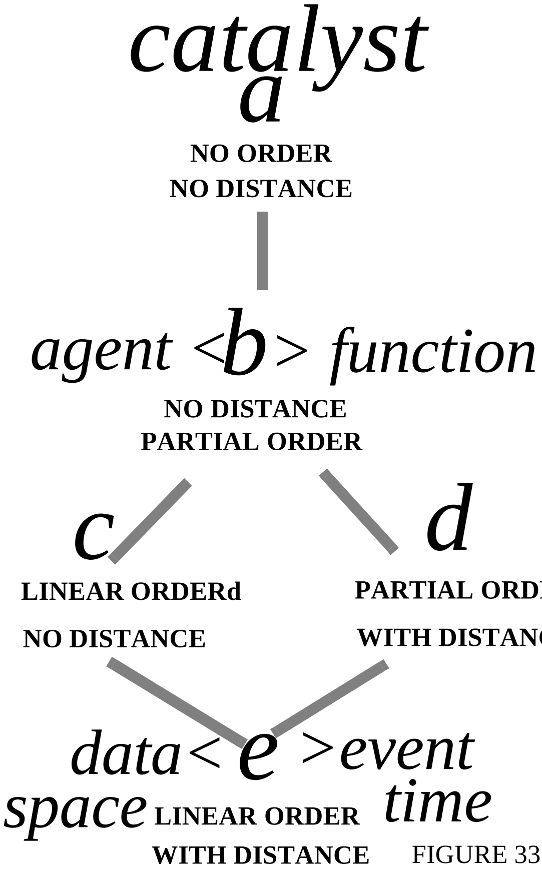


FIGURE 33

To this static representation of the formal-structural system produced by Klir, we bring a demand which lies outside of his purview. We need to represent the perspectives that have been found to be an important element of the meta-technical. Fandozi, in his discussion of technology, almost equates perspectivalism with nihilism.

> Although Marcuse's analysis was concerned primarily with the theoretical background of science, it is important to see that there emerges in his results a concept of reality which is the

> correlate of the scientific endeavors and which lets the world appear as perspectival.* Entities in the world no longer possess a definite nature as such, but are rather defined by their context, e.g., a tree is an obstacle for the new freeway, potential lumber for a house, or an aspect of aesthetic appreciation. The destructiveness of technology is not just based on the fact that lumber takes priority over aesthetics, but that any perspective as interpretation of a thing becomes equally possible, equally cogent. A thing is defined according to its use, and that use itself is a function of the current level of technology and other extraneous factors -- factors which do not directly confront the thing in question. The thing itself is not allowed to address us; it is not permitted to define its own situation, to prescribe a meaning to us. Within the scope of arbitrarily eligible perspectives, things lose their definiteness, their presence within a interrelated totality. They become mere stuff, formless matter; they approach a state of non-being. Things become nothing.[120]

This paragraph in many ways shows the relation between the meta-technical perspectives and nihilism better than anything else said here. It is perspectives which hide the singularity. It is the equi-possibility of perspectives which constitute their nihilistic aspect. So, as has been shown in many ways in this essay, possibility is married to passive nihilism of the meta-technical. Perspectivalism and possibility are intimately related. We see new possibilities when we switch

[120] Pages 106-107 in P. R. Fandozi Nihilism and Technology.

perspectives. Those possibilities are hidden, and new ones appear as we switch perspectives again. Possibilities are the showing and hiding avenues related to perspectives. Yet, this showing and hiding is deeper than that of the ready-to-hand. It is the showing and hiding which is inherently social. Intersubjectivity, the famed problem of phenomenologists, appears here.[121] Intersubjectivity hides the "essence of manifestation" which amounts to the singularity of pure immanence. In intersubjectivity, there is something hidden irrevocably that may never be shown. It is embedded in the interchange of perspectives. It is that arbitrary interchange that causes passive nihilism to eat up meaning. Sartre calls it "nausea" as he contemplates the waiter in the cafe. Most philosophical analysis -- socially naive -- concentrates on the self/other dichotomy. More sophisticated analysis realizes that this is an abstraction from more complex social interaction.[122] Yet it is possible to hypothesize that perspectives bifurcate just like dynamical system states on the way to chaos. In our case, we wish to maintain our analysis at the threshold of 2^2.

[121] See Bib#

[122] See Bib#

We recognize that the chaos of social interaction is primary. The individual appears for our culture as an isolated ego from this milieu. He experiences bifurcation at separation from the state of immersion in the presence of the mother. Mother, in time, becomes mother/father, and ego becomes self/ other (I/thou). However, we are not dealing with these basic archetypal psycho-social dichotomies.[123] Here it is a more abstract analysis which will consider the perspectival fragmentation of the formal-structural system. A set of four specific perspectives to be suggested here are relevant to the analysis of software systems. Yet the point being made is more fundamental. The perspectives form a minimal system in the sense defined by B. Fuller in Synergetics I and II.[124] It is the minimal system of perspectives that is complex enough to hide the singularity. Most analysis of formal-structural systems use only one or two perspectives. They do not attain the required complexity necessary to represent meta-technological systems. The advance suggested here is to consider how the formal-structural system is distorted by the

[123] See Bib#.

[124] See Bib#

introduction of a minimal system of perspectives. This is a concrete example of the intensification of the technical into the meta-technical. In effect, we are introducing the equivalent of relativistic intervals into the space-time continuum within which the formal-structural system exists.

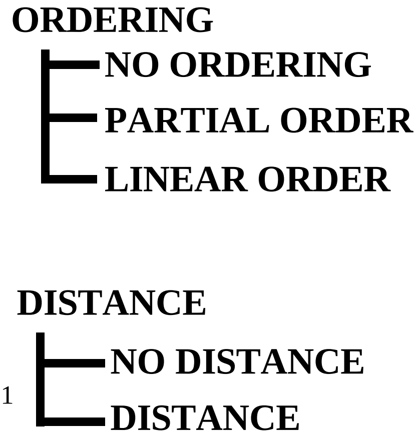


FIGURE 31

One might well ask at this point what a perspective really is. An excellent semiotic study concerning this issue is Signifying Nothing.[125] In that study the structural equivalence between "the number zero" -- "vanishing point" -- and "money" is suggested. Suffice it to say for us that a perspective is the cutting of the world into multiple related horizons. The intersection of the structures that define these horizons is a null point which is "nowhere" in relation to those defining

[125] See ??? {Signifying Nothing}

structures. This “nowhere” is represented by the vanishing point within the horizon. By signifying the vanishing point, the opposite observation point may be specified. Thus, the perspectival structure gives a means of orderly movement from observation point to observation point by moving the vanishing point. The multiple horizons reveal different realms of possibility open to exploration. What is open to exploration from one perspective is closed to another. This only gets complex when different subjects are simultaneously exploring different perspectives. Then relativistic rules apply between simultaneously exploring subjects. Perspectives can either be quantitatively different in terms of space, time, position; or qualitatively different. Qualitatively different perspectives are the kind Fandozi referred to earlier. From qualitatively different perspectives, one sees different kinds of timings instead of different sides of the same things. Qualitatively different perspectives are much harder to deal with. Since we are dealing with a specific example of meta-technology, i.e. software, this problem is not as difficult as it might seem. In fact, we will postulate, as explained in part one of this essay, that it is the foundations of the technological system that provide the basis for distinguishing between the quantitative perspectives relevant to software. Two of these are quantitative and related to space/time. The other two are qualitative and related to the first and second meta-levels of Being. The quality to be distinguished is the present-at-hand from the ready-to-hand. The Western philosophical tradition contributes amply to the possibility of this discrimination. Now how these four perspectives (named here: event, data, agent, function) unfold has already been laid out. The question now is what the insertion point for these perspectives into Klir's representation of the formal-structural system are.

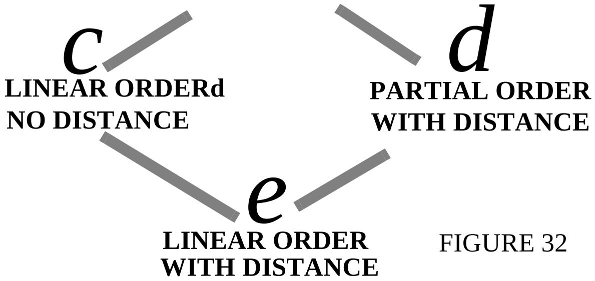


FIGURE 32

The insertion point for perspectives is readily provided by Klir. It is provided by his discussion of “backgrounds” and “supports.” Let us turn our attention to these ready made insertion points for perspectival differences and consider the ramifications of this limit that will move us from the technical to the meta-technical in a single bold stroke.

◆ ◆ ◆

According to Klir, the object system is composed of a set of attributes and a set of backdrops. Any one attribute or backdrop has a set of possible values whose range is circumscribed. Klir maintains that a position of ontological ignorance[126] may be maintained concerning the “reality” of attributes and backdrops. This is possible because the entire “object system” is disposable. It is merely the starting point for the definition of the source system which is an instrumentally defined measuring device levied against the object system. In the source system object attributes translate into operational representations called “variables.” Likewise, object backdrops, i.e. global environmental attributes, translate into

[126] See ASPS p 37.

operational representations called “supports.” Supports allow different observations of the same variable to be distinguished.

> When more than one support is involved, the overall support set is the Cartesian product of the individual supports sets. Properties recognized in each individual support set have to be properly combined to express recognizable properties of this Cartesian product. These properties of the overall support set (the Cartesian product) are then used in characterizing, together with properties of the associated state set, an elementary methodological distinction. If the same properties are recognized in each of the individual support sets, it is easy to combine them, and the derived overall properties are homogeneous over the whole Cartesian product. <u>The situation becomes more difficult when properties recognized in the individual support sets are not the same</u>. In such cases, there are, at best, some overall properties that do not extend over the whole Cartesian product. [My emphasis][127]

The last two sentences of the quoted paragraph are important for this study as they point the way to understanding the fundamental properties of software methods. When multiple support sets are used which do not share the same mathematical properties, then the Cartesian space represented by these supports is intrinsically fragmented. It turns out that this is true of the supports for the software system.

[127] Klir ASPS page ??

Each of the four fundamental perspectives supply the supports for the software system. These supports, as noted in part one, are event, data, agent and function. These derive from time, space, pointing and grasping which are the ontological foundation of any formal-structural system. In this case, the environment which is posited for the object system is its own ontological foundations. These foundations provide the essential backdrops for all formal-structural systems. When we drop the pretense of ontological ignorance, it is readily realized that the object system and its environment are the same thing turned inside out. In our case, the object system is the formal-structural system itself. Its only possible environment is its ontological presuppositions. These presuppositions lead to groundlessness. The groundlessness of the ontological environment is nothing but the reflection of the erratic change of the formal-structural system itself. It is these ontological foundations which manifest groundlessness that are the systemwide universals that can be used as backdrops for measuring the attributes of the formal-structural system. As backdrops, they are space/time and Being^1/Being^2, i.e. pointing/ grasping. These are turned into supports in the source system which appear as data/event/agent/function. The transition from "backdrop" to "support" occurs through the articulation of perspectives and their projection back onto the ontological foundations. The ontological foundations become buttresses which allow one to get a view of a portion of the formal-structural system. The key point here is that the place where perspectives plug into Klir's representation is right at the base where he defines backdrops and supports. When supports do not share mathematical properties, then the inherent fragmentation of perspectives occurs within the total support set. Different horizons open up to different perspectives due to this fragmentation of the total support set. Klir is very precise about the methodological distinctions that derive from the mathematical structures of the various supports. It turns out that there are a limited number of possibilities for combination of these mathematical properties as shown in Figure 31.

Each of the supports may be of any of these five methodological distinction types. Which type (*a*, *b*, *c*, *d*, *e* in Figure 32) each support is, determines how easily the supports may be combined and the degree of fragmentation that occurs in the total support space. It is possible to assign each of our perspectival supports to a particular methodological distinction as shown in Figure 33

Space and time are, as any physics student knows, a fully linear ordering in which distance may be described. Without this continuous backdrop, the description of physical systems would be well nigh impossible. The fact that the continuity is itself an illusion projected upon time and space and maintained by the ideational mechanism of the real number line and calculus need hardly be mentioned. Agent and function, on the other hand, are represented by partially ordered sets. They cannot support the concept of either linearity or distance. In software methods, the pure description of each of these perspectival minimal methods is a hierarchy represented as a tree structure. Agents are seen as a tree of tasks (tasks within tasks), and function is seen as a tree of functions (functions within functions). How far a task is from another task makes no sense. Functions are not transitive. It cannot be said whether a function is “between” to other functions or not, unless data flow is added.

Functions themselves, without time or space ordering added to them are just a library of routines, some of which call others in a tree-like structure. Thus, agent/function supports are very different from event/data supports in terms of the methodological distinctions that apply to each perspective.

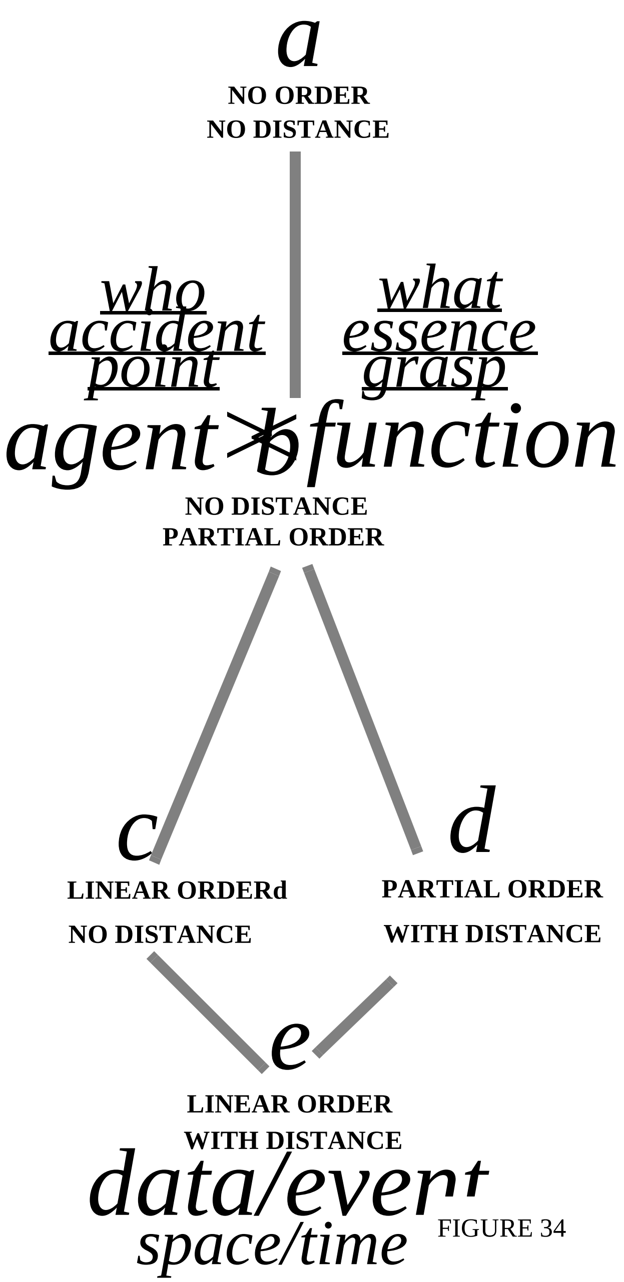
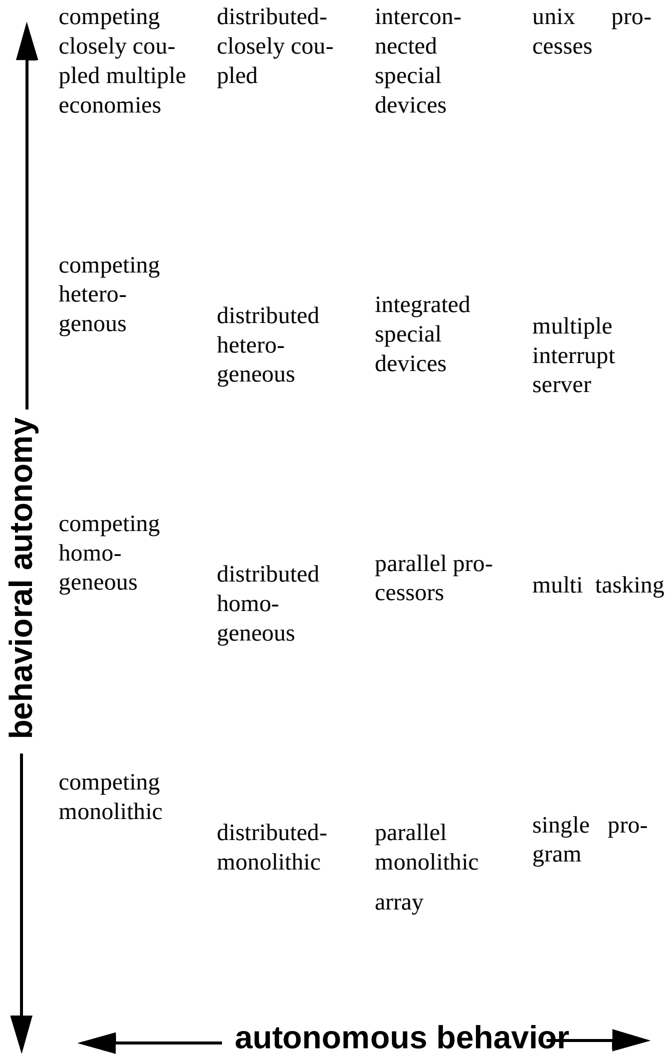


FIGURE 34

If you ask “who?” or “what?” -- these questions do not refer to continuums but to differences in accidental and essential characteristics. Who you are is dependent on the accidents of your life. Who were your parents. What name they decided to give you. What house was available when you could afford to buy. What number was next when you applied for a social security number. What day you happened to be born on. Who you are is determined by all the accidental circumstances that distinguish you from everyone else. This set of accidental facts that makes you unique is a set which has only, at most, partial ordering. Likewise, all the essential facts about you, like what you believe in or don’t, your skills, your profession, your hobbies, that indicate what kind of person you are, at most, partially ordered. You belong to this set which is a subgroup of another set. The complex Venn diagram that singles out both in terms of accidents and essentials from all others in the population cannot be fully ordered.



competing closely coupled multiple economies	distributed-closely coupled	interconnected special devices	unix processes
competing heterogenous	distributed heterogeneous	integrated special devices	multiple interrupt server
competing homogeneous	distributed homogeneous	parallel processors	multi tasking
competing monolithic	distributed-monolithic	parallel monolithic array	single program

FIGURE 35

Pointing and grasping are modalities for the human encounter with things in the world. The set of things pointed at are distinguished from those which are not. Likewise, the set of things grasped forms a distinct set over and against what is not grasped. Without adding space/

time, at best these can be partially ordered into sets of things pointed to or sets of things grasped. Inherently, there is no distance nor linearity in pointing and grasping. Only hierarchies of distinctions can be supported by these modal concepts.

This realization that different mathematical distinctions apply to the various software methodological supports is a crucial step in understanding the way perspectives are fragmented. The fragmentation has a particular structure which has important implications for software design methods. These implications are best brought out by studying the diagram of methodological distinctions produced by Klir more closely.

There is a great imbalance between agent/ function supports and event/data supports. Event/data supports have the benefit of full ordering which will allow any signal or datum to be pinpointed in the linear ordering of memory locations or CPU cycles. We can gauge the distance between any two signals or datum by counting discrete cycles or memory addresses. Thus, it is clear where and when each datum/ipsity (eventity) occurs in the discrete space/time environments provided by computing hardware. The mapping of this full discrete ordering onto an illusory continuity of the space/time environment made possible by the distribution in space of processors and the illusion of continuous execution due to fast instruction processing, allows the computer, with its software, to fit into our picture of the physical universe. Space/time eventities within the computer map to space/time eventities outside the computer. This mapping makes use of the power of the illusory continuity of the real number line to give a four-dimensional continuous support space. This four-dimensional support space (either space/time [x+y+z-t] or Minkowoski's time-space [past+present+future-nowhere]) has very powerful representational features. Without this possibility of mapping, simulation spaces onto our idea of the physical universe, computing machines would be useless.

On the other hand, the agent/function supports are weakly ordered. This weak ordering is difficult for us to deal with. We are used to dealing with fully ordered mathematical objects. Sets are not as easy to manipulate as cartesian coordinates. Yet, we recognize that the weak ordering of agent/function has its own significance. Accidents and essences are weakly ordered by their nature. Space and time are like empty vessels for objects to move in. We cannot say anything about the objects themselves using space/time. The essences and accidents that define the objects as more than blank empty entities are what give us really important information about the things gliding through space and time. Essences and accidents give us a glimpse inside the black box. We pay for this glimpse inside the black box. We pay for this glimpse by losing ordering information. We gain by finding out who's who and what's what. These partial orderings, with no distance, can be seen as the delineations of kinds, either essential or accidental. In philosophy, these are called "sortals" by some who think it important to distinguish natural complexes[128] from abstractions. We can basically sort things according to external coherences or internal coherences. External coherences relate to the accidental configurations of things that appear in the world. Internal coherences relate to the essential configurations of natural complexes within themselves. The internal coherences

[128] See ??? {Natural Complexes}

give one kind of sorting, while the external coherences lead to a completely different kind of sorting. Klir lumps both together under the concept of “population.” We think of a population as a set of organisms of a similar kind. It is composed of individuals in a particular space-time distribution. But important information is to be gleaned by looking at the internal and external coherences of the set of individuals taken as a whole. The external coherences will classify individuals on extrinsic features such as serial numbering. The internal coherences will classify individuals on the bases of intrinsic characteristics like their different behaviors. These intrinsic and extrinsic characteristics tell us a great deal about the set of individuals that space-time distribution alone would not be able to tell us. Yet, this information is dependent on our ability to distinguish. The most basic essential distinction of an individual is based on behavior. The sortals related to behavior are called functions. The most basic accidental distinction of individuals is based on autonomy. The sortals related to autonomy are called agency. The agent is the source of a behavior. The behavior is the expression of the autonomy of the agent. In the field of computational machines, the behavior of a processor is the transformation of data grasped in accumulators. The autonomy of a processor is the indication of which task is in control by pointing at an execution thread or context. The individual processor, like the minimal organism in a population, has both autonomy and behavior of its own. The concept of population contains both of these concepts which, from the viewpoint of software methods, need to be vigorously distinguished.

All of this becomes clearer by realizing that in the course of the development of software, we traverse the set of methodological distinctions (a, b, c, d, e) from no ordering to full ordering of space-time eventities. Requirements are normally without ordering or state some boundary conditions for space time ordering of eventities. As analysis and design progresses, the exact space time ordering system eventities is determined. Software methods need to be able to describe these fully ordered eventities, and do so in terms of events and data perspectives. However, making certain events occur at particular points in space are the purely performance aspects of the system. It is also very important to analyze and design who does what in the system. The "what" characterizes the functionality of the system. That is the particular kind of expected behavior in space-time. The "who" characterizes the division of labor between cooperating agents. The "who" normally determines the structural aspects of the design. The "who" is structural, whereas the "what" is formal. This reversal occurs because point and grasping are mutually sustaining, like space and time. They are complementary opposites that make each other possible. The assignment of autonomy to individuals structures their interaction. The assignment of behaviors to individuals formalizes their actions. In the one case, structure is on the surface while formalism is deep, while in the other case formalism is on the surface while structure is deep. Pointing is the action of an autonomous individual. The deep aspect is the formalism of rendering a present-at-hand image at the point of focus. When this ability is assigned to an autonomous individual, the surface structural aspect becomes apparent where different individuals do different things in cooperation. Grasping is the behavior of an individual. The deep aspect is the structuralism of rendering a ready-to-hand means of moving between temporary gestalt images. When the behavior is assigned to a particular individual, the surface formal aspect appears where a particular individual has a language of transformations he can perform to transform one image into another.

Grasping is the behavior of an individual. The deep aspect is the structuralism of rendering a ready-to-hand means of moving between temporary gestalt images. When the behavior is assigned to a particular individual, the surface formal aspect appears where a particular individual has a language of transformations he can perform to transform one image into another. Structuralism and formalism belong together. Each fills a crucial role in the full articulation of the other. Thus, there appears both formal-structural and structural-formal phases similar to the difference between space-time and time-space. There is "behavioral autonomy" and "autonomous behavior." The former emphasizes the behavior of the individual over autonomy, whereas the latter emphasizes the autonomy of the individual over behavior. The autonomy of competing processors is greater than the autonomy of tasks. The behavior of closely coupled environments is more complex than that of monolithic environments.[129]

◆ ◆ ◆

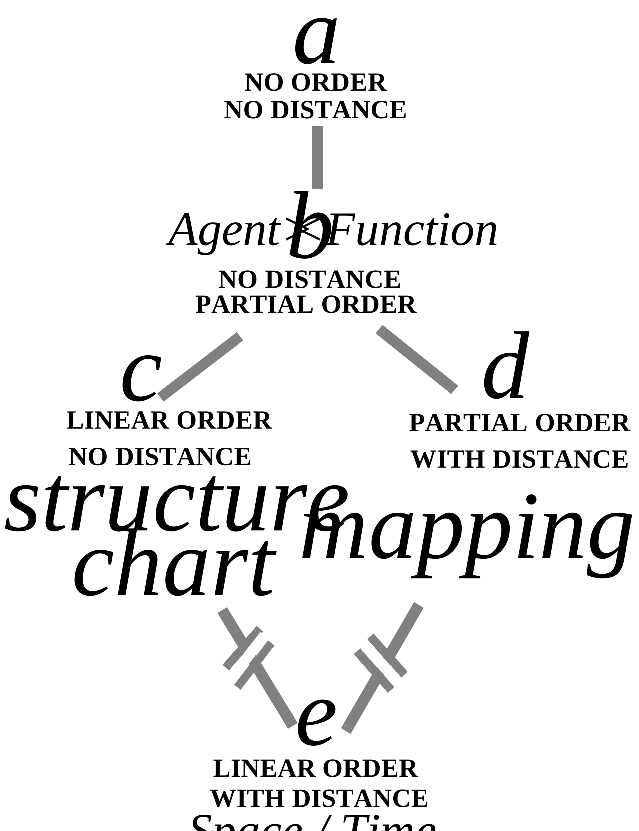


FIGURE 36

Traversing the lattice of methodological distinctions (a, b, c, d, e) in the development from requirements to design to code, it is clear that at a certain stage each support is made up of non-ordered sets of distinction. Then at some stage, all become partially ordered with no distance. In the final product, events and

[129] See Figure 35.

date (space and time) become fully ordered (linear with distance). However, at most, agent and function remain partial orderings. This is a big problem for software engineering. The fact that agent and function separately can, at most be partially ordered without distance causes a major break between the space and time supports and the model supports. This "gap" in the representational power of these two different kinds of perspective causes a rift in the designer’s ability to move from concepts of agents and functions to a final space/time ordering of software eventities. This rift is the major methodological problem of software engineering. Even though it is possible to appreciate that the lack of full ordering for model perspectives allows a fuller expression of internal and external coherences of the software’s functionality and structure, still it is difficult to deal with this rift in practical terms. In the development of software methods, it is this gap which must be addressed. It is, in fact, this intrinsic rift which is the fault line that hides the singularity of pure immanence. In addressing the fracture that separates software “supports,” we are orienting ourselves to the hidden heart of software.

Even though agent and function can each, by itself, only be partially ordered with no distance, together it is possible for them to approach closer to full ordering. In order to attempt to narrow the rift, the two partial orders are used together to imitate linearity without distance (structure chart) and to imitate partial order with distance (mapping).[130]

In mapping, distance is expressed in how many boundaries must be crossed from one end point to the other of a mapping arrow. The minimal formal calculus of G. Spencer-Brown in Laws of Form[131] may be applied to measure the distance implied by mapping. In structure charts, linearity is expressed as a calling sequence. The control flow of a calling sequence is like passing a token of autonomy through the software system. Autonomy is passed from function to function, and the sequence of calls represents the linear structure of the program.

It is interesting to note that these two ways of using two partial orders together are duals. There is a one-to-one correspondence between

[130] See Figures 37A&B.

[131] See Bib#

their elements. The path of the token of autonomy is analogous to counting boundary crossings in the mapping. The point here is that structure charts and mappings are the two minimal methods that connect the agent/ function perspectives. These minimal methods are much more complex than those that connect the event/data perspectives. There is good reason for this lopsidedness in the complexity of minimal methods in these cases. The minimal methods connecting agent/function must make up for their weak orderings. They do this by working together to attempt to bridge the gap to full ordering. In so doing, two possible gap-bridging paths are articulated which lead to the minimal method of "mapping" and the minimal method of "structure charting." Note that the inherent linearity of structure charts approaches the representation of linearity in time. The inherent distance of mapping approaches the representation of distance in space. So structure charts are close to the temporal ordering of software eventities, and mappings for traceability are close to the spatial or data orderings of software eventities. Yet being close is not the same as being completely isomorphic. In each case, a crucial ordering characteristic is missing. Yet, if the linearity of space and the distance of time is relaxed as it can be for some classes of systems, e.g. information systems rather than embedded systems, then this attempt at gap bridging is good enough to give full methodological convergence. However, in embedded real-time systems, this relaxation is not possible. So there is a class of software systems in which the rift cannot be hidden. In this class, the inner nature of software becomes apparent as we stretch the limit of what we are able to do with software. It is that class for which methodological studies are most necessary, and it turns out that this is the class of systems that attempts to sense and react to the "designated-as-real" or "physical" world.

No wonder software is hard to build, and software methods are difficult to use. There is a fundamental rift between what our perspectives can represent in an ordered fashion. This gap is, in principle, unbridgeable, even though for some classes of system the rift appears to vanish. Our most important methods (structure charts and mapping) attempt to bridge this gap. Because of this function, these methods are more complex than any of the other minimal methods that act as bridges between perspectives. Linearizations of the software system, together with many-to-many mappings are the two main ways we have to approach the mental simulation of our systems. Linearizations, through which tokens of autonomy move, express these systems in terms of autonomous behavior. Many to many mappings express the behavioral autonomy of the system. The former shows what part of the system is autonomous at what point in its execution. The latter shows how the overall system behavior is automated by deployment of functionality to different system segments. Autonomous behavior means . . . what part of the overall system is acting independently. Behavioral autonomy means . . . how has the behavior been partitioned across autonomous units. These two different aspects, like their counterparts space/time and time/space, are difficult to express. One looks at individual pieces of the system as encapsulating behaviors through which autonomy moves, given different inputs and states. The other looks at the overall system as expressing a set of behaviors and looks to how those behaviors are allocated to the different pieces of the system. This difference can be brought home by referring again to the gestalt. The gestalt is a number of dynamically-related figure-ground relations. Each figure-ground relation expresses a particular kind of behavior of the gestalt whole. The total behavior of the gestalt is the sum total of these relations. Yet, all cannot be seen at once. Our perception moves through the figure-ground relations one at a time. This is like the movement of autonomy through the structure chart. Looked at analytically, we can see behavior distributed among different figures. Looked at synthetically, we actually perceive the shifting from figure to figure as the center of attention changes. Perception snaps from one whole to the next. The background of the gestalt remains always ready-to-hand in relation to the present-at-hand figure. We point at the figure, but we grasp (understand) the whole gestalt which is more than the sum of its parts. Behavioral autonomy means giving autonomy to behaviors via mapping. Autonomous behavior means the automation of the behaviors seen in a particular linearization which moves through each of the figure-ground relationships. In autonomous behavior, the accidents of where control is within the linearized system predominants. In behavioral automation, the essence of the behavior of the whole system predominates.[132]

◆ ◆ ◆

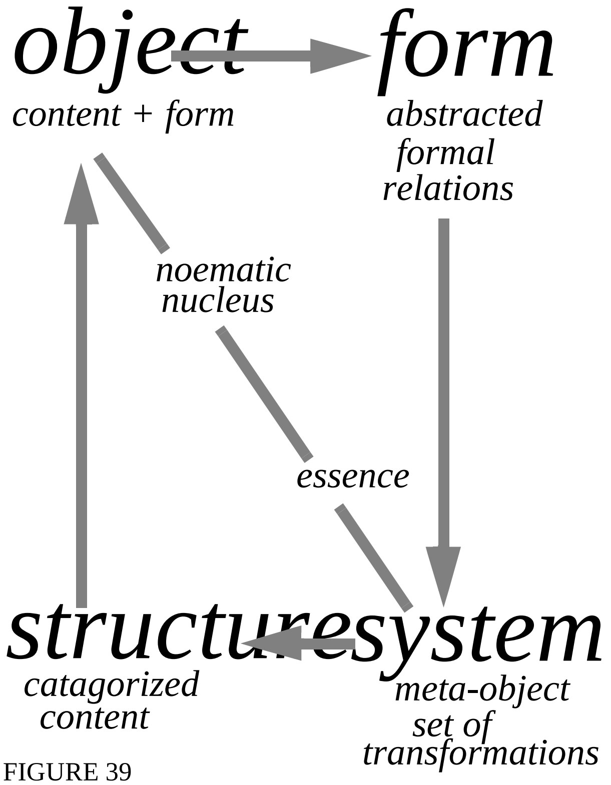


FIGURE 39

We have taken our clue for how to deal with the relations of perspectives to the formal-structural system from Klir’s discussion of “backgrounds” (in the object system) and “supports” (in the source system) and the relevance of methodological distinctions. By looking at the different kinds of

[132] If this distinction between Behavioral Autonomy and Autonomous Behavior appears forced, there is good reason. They are nihilistic opposites now seen in action. Don’t worry; they cancel, leaving nothing at all. See Bib#.

methodological distinctions that relate to agent/ function perspective versus event/data perspectives, we have seen the advent of a fundamental rift between these pairs of perspectives. That rift gives us our first concrete evidence of the presence of a singularity within the set of software perspectives. It has also indicated an interesting way of looking at the relations between software methods. Methods arise as means of attempting to bridge the rift between pairs of perspectives that are methodologically distinct in Klir's sense. Agent/function perspectives are augmented from partial order with no distance to have either linearization or distancing. Event/data are reduced from full ordering to emphasize either linearization or distancing. Thus, the meeting point of methods occurs in the methodological distinctions related to linearity with no distance or partial ordering with distance. Perspective pairs (agent/function or event/data) are more extreme in their ordering than the methods that connect them which exemplify linearization or distancing. This is a key point that makes a deeper understanding of software methods possible. All software methods come in pairs. Each pair relates two software perspectives.

One method from a pair emphasizes linearity, while the other emphasizes distancing. The pairs may be thought of as looking from one perspective toward the other perspective or vice versa. Thus, the mapping method looks at agent from the point of view of function, whereas the structure chart method looks at function from the point of view of agent. The point of view exemplified by the method uses the other perspective as an object. The subject/ object dialectic gives each method its own peculiarity: what is a subject for one method is an object for its opposite method. This inversion of subject-object relations is then colored by the emphasis on either linearity or distancing that gives substance to the method pairs. Method pairs may be considered as bridges between particular pairs of perspectives. Methods give the designer a means of traveling from one perspective to another in his mental simulations of a software system. The designer may only be in one particular perspective at a time. Design work is a path of thought which traverses the perspectives one at a time in order to build abstract design representations of a software theory. The software theory is the ultimate synthesis of the software system projected by the designer. The ultimate synthesis revolves around the subjectivity of the designer and cannot be completely represented. The designer, as subject, assumes the different software perspectives one at a time to get the particular view each offers. The terminology invented by Husserl[133] in his phenomenology can be used to describe this process. For the designer as subject there is the projection of the “intentional morphe” upon the “hyle” or matter of the software system. In this case, the matter is a pure plenum of off/on bits. The “intentional morphe” is a forming intention that projects the software system. From this fundamental subject/object interaction, “noema” and “noesis” arise. The “noema” in this case are the various levels of software objects like “bit,” “byte,” “counter,” “statement,” “routine,” “program,” or “software systems.” The “noesis” is the various ideational structures such as go to, assignment, if statement, algorithm. The noesis is the thought structures which determine the structuring of the program. The noema is the structural concepts which ultimately, translate into patterns of bits.

[133] See Bib#

Husserl's terminology gives us a terminological precision for dealing with subjectivity similar to that offered for objects (called systems) by Klir. This terminological precision is required in order to situate our concept of "methods." Methods are normally understood as sets of techniques used in a particular order to produce abstract representations of software objects. In fact, these representations are particular aspects of a software theory.

Methods are a kind of noesis at a particular level of abstraction which produce a corresponding noematic representation. The level of abstraction at which software methods arise is precisely where software comes to be considered a system. Klir's lame definition of "a system" as a set of attributes avoids the controversy surrounding the synthetic nature of systems. Systems are natural or artificial complexes with internal and external coherence. They are normally sets of dynamically interacting objects. Klir's position of ontological ignorance is compounded by his empty definition of a system as a collection of attributes. By avoiding the controversy surrounding the synthetic nature of systems, he is led to build the general science of systems on weak philosophical grounds. Operationalism pushes all substantive questions concerning the nature of the system under the carpet. It is ultimately the same position as that which claims technology is neutral. It is the epitome of nihilistic positions. Operationalism and technological neutrality allows the subject to identify with technology. We give up our humanity without a second thought. We are the technological system looking at the world through “observational channels.” When we lose our ability to distinguish between ourselves and technology, that dehumanization leaves us totally lost in the nihilistic essence of technology. Instead, we must struggle with the concept of a system and distinguish it from the concept of object. An object is a thing pointed at and focused on present-at-hand. A system, on the other hand, is a set of objects interacting dynamically as a whole. This is precisely the same idea of a set of figures with the same ground that dynamically form a series of gestalts. The idea of a system is an attempt to make the ground within which the objects of a system relate dynamically present-at-hand. In physics, this is represented by the concept of a field. In the field, system-wide interactions between objects are given a specific mathematical form. A system would be better recognized as a manifestation of ready-to-hand phenomena. As such, it can never wholly be made present-at-hand. Thus, come the problems of defining a system. Either definitions render it empty, as does Klir, or get tangled in messy ontological problems. The recognition that a system is more than an object with a different kind of coherence immediately leads us to suspect that it has different ontological grounds. These grounds are glossed over by futile attempts to turn it back into a mere object as Klir would have us do. Instead of recognizing these grounds outright, Klir reintroduces the distinction between general systems and specific systems. Systems science occupies a meta-level to the study of all particularized systems by various disciplines. The meta-level is introduced surreptitiously in the sophistic style so common in Western theorizing. The structure of Klir’s systems theory is brilliantly conceived. But it lacks the straightforward recognition of the difference between systems and objects. Objects are present-at-hand focuses of attention for subjects. Systems are present-at-hand representations of ready-to-hand phenomena.

These representations always mix formal and structural representations to achieve their effect and always represent meta-level constructs, where the contents of forms are structured, to achieve control over the transformations from one form to another. The set of transforms is characterized as a system. In this manner, form and structure mediate between objects and meta-objects, i.e. systems.

The "intentional morphe" is what projects form on content (hyle). The noetic and noematic cognitive striations reveal pure form and pure structure as abstractions within the field of the system. Form and structure are actually bound together as necessarily intertwined within the system. They are the internal and external coherences of the system. The external coherence appears as the series of forms arising within the gestalt whole. Structure is the transformations of contents allowed by the categorization of contents. These transformations of content allow the tracking between formal transmutations which makes it possible to relate a particular form to all the other manifestations governed by the system field.

According to Husserl, it is the arising of essences as distinct from simple ideas which is the key point. Essences are different from the results of induction or deduction. These latter are purely formal, logical performances. Charles Pierce contrasted these with “abductions.”[134] In abduction, one jumps to the conclusion. As Husserl noted, one knows “a lion” the moment it is seen. Its essence leaps forth without induction or deduction. Essences arise at the systematic level. Objects have noematic nucleuses which reveal their form-content coherence as objects. Noesis provides our ability to understand transmutations by categorizing contents and mapping between discontinuities informal presentations. The ability to perceive structural relations reveals the essence of the objects. The object’s essence relates to its possibilities of variation and reveals its inner necessary structure. Essences are separated from accidents as the inner coherences of objects. This separation of essence from accident appears for the first time at the systemic level. There is a direct line between that separation and the arising of point/grasp or agent/function as images of the model differentiation between the present-at-

[134] See Bib#

hand Being [1] and the ready-to-hand Being [2]. Essences have a different mode of Being than objects as neomatic nuclei. Husserl discovered this, and Heidegger based his whole philosophy on this difference in modality. Essences express the variability of the object within the systemic field. They also express the limits of that variability where one figure/form transforms into another, and the gestalt changes. Once it is realized that essences have a different modality from the noematic nucleus that supports them, then the importance of accidents becomes obvious. Accidents are what make different incarnations of a certain kind of essence unique. This uniqueness is the key to connecting essences to concrete lived reality. This is why existence occurs as a level between the phenomenal and the life world. Existence grounds essences. The agent/ function dichotomy relates this same point back to software systems by the distinction between autonomy and behavior. Without processors, functional behavior would never be concretely realized in space/time.

Systems are meta-objects whose nature is technological. Systems are, in fact, the technical view of everything. We see everything as systems. In this way, we convert them into meta-objects which we can deal with technologically. Klir's epistemological levels show how this technologization of objects works in a clear way. The object is instrumented, observed and then simulated. Then the object itself is discarded and replaced with the simulation artifact. The simulation artifact at a technological level may be a machine that imitates the original but makes the process imitated more efficient. This process of technologization is intensified at the meta-technical level. There software allows the simulation to be adapted and integrated. The differences between the technological and the meta-technical have already been discussed. The meta-system appears in Klir's work as the infinite regress of meta-models and meta-structures. These two infinite regresses are similar to those generated by Russel and Whitehead's theory of logical types.[135] This is where the concept of meta-levels as the solution to logical paradoxes was first tried on a large scale. The problem was that infinite regresses of meta-levels were produced so that solving the paradoxes just changed where the problem lies. That these same infinite

[135] See Bib#.

regresses should appear in the description of the formal-structural system is an important fact. The fact that two infinite regresses appear together is also important. These are the expression of the in-hand modality ~~Being~~[3] within the description of the formal-structural system. The singularity of pure immanence is the ultimate source of these two horns of Klir's dilemma. The dilemma is that infinities cannot be factored out of his model. They appear as the ultimate barrier to fully understanding systems.

◆ ◆ ◆

Software systems are "systems" to the extent they imitate hardware. As software, they are really meta-systems. When we build software systems, this distinction between system and meta-system is lost for the most part. We do not think of the meta-system, but suppress it by outlawing such things as *goto*s and self-modifying code thinking that the problems have been solved by this expedient. When we come to define the software method, we find that this distinction again becomes very important. Our approach has been to take the singularity which the infinite regress of meta-structures and meta-models banished by Klir to exotic realms beyond where it is possible for us to think clearly. (Remember, Bateson pointed out that we have difficulty thinking beyond the fourth meta-level.) Instead, our ruse is to take the singularity and embed it at the heart of the description of the formal-structural system. This is done by turning our four perspectives into “supports.” Thus the relation between perspectives as “supports” and system attributes, i.e., the component concepts of methods, expresses the relation between system and meta-system (between Process Being^{2} and Hyper Being^{3}). What Klir presents as a present-at-hand model has two other modes of Being hidden in it. We have drawn out the ready-to-hand (as system) and contrast it directly with the in-hand (as singularity that is source of two infinite regresses of meta-levels). The present-at-hand object which was Klir’s empty definition of object has been given life by introducing a true subject-object dialectic. The subject “takes” perspectives and views from one perspective the object bound to another perspective. The observation channel becomes an introspection within the system of the formal-structural system itself. Each introspective observation channel is characterized as a minimal method. A method is a means of viewing other perspectives, and also a means of moving to another perspective. The dialectic between the perspectives and the methods provides the relation between the system and the meta-system. The perspectives are faceted, and embedded within their facets is the singularity of pure immanence -- the point of pure cancellation for all the relations between the perspectives. What is left when this cancellation has occurred is the proto-technical meta-meta-system.

This transformation of Klir's formal-structural system representation causes methods to play a crucial role. They are essential bridges between the technical and the meta-technical realms. Through methods, the essential natures of systems become clear. The essence of systems must, in fact, be meta-essences because systems are the field within which essences are first apprehended. The essence of the field that gives rise to essences must be a meta-essence. Essences and accidents are delineated by partial orderings with no distance. These are contrast with full linear ordering with distance provided by space/

time. The perspectives themselves arise from these two opposite methodological distinctions. It is, in fact, the methodological positions of linearization without distance and distancing of partial orders that support the meta-essences. Methods are meta-essences. This is why they can describe systems which are the fields within which essences and accidents are discovered. Only as meta-essences can methods be directly related to the nature of meta-systems composed of software. Only as meta-essences can methods deal with the presence of the singularity and represent the nonrepresentable software theory. Each introspective observation channel connecting perspectives on the formal-structural system contains a single meta-essence that bridges not only between the perspectives, but also between the perspectives and the system observing itself. What is a meta-essence? Sounds esoteric. An essence is the variability of the object. One would expect the meta essence to express the variability of the system. It must express the limits of the variability of the system. Without the necessary attributes of the meta-essence, the system would no longer be a system. The set of meta-essences together make up the internal coherence of the system, and the meta-essences all cancel, making the systemic field vanish. Meta-essences must cancel, leaving only the residue of Wild Being which is what lies beyond cancellation.

◆ ◆ ◆

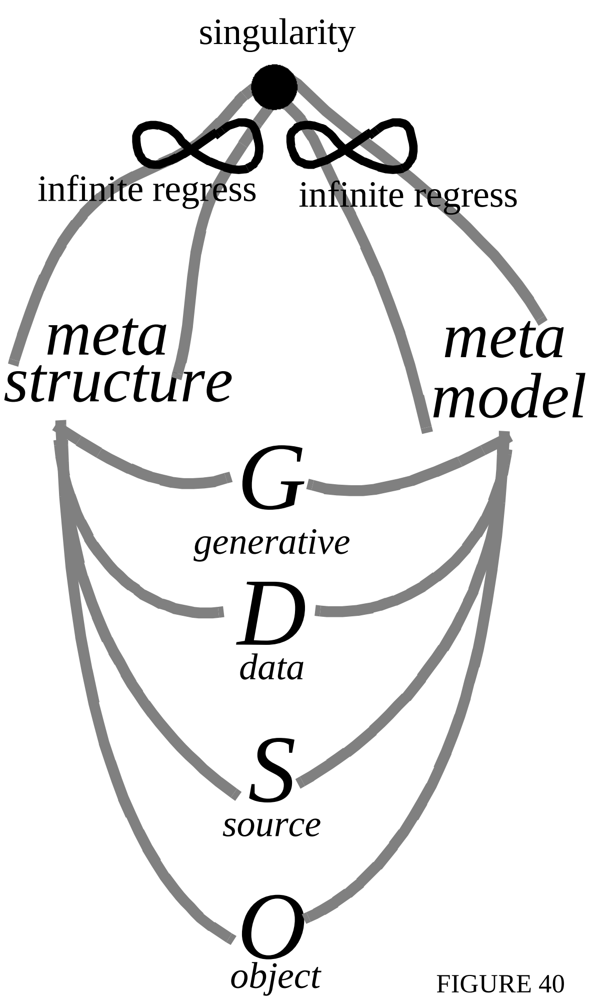


FIGURE 40

The next step is to extend the concept of methods as meta-essences built upon intermediate methodological distinctions. This can be best done by comprehending the difference between linearization (L~D) and distancing (P+D). The difference between Spacetime (x + y + z - t) and Timespace (past + present + future - nowhere) has already been mentioned. This difference occurs because of the nature of the interval. Intervals consist of a temporal and spatial displacement. Because of relativistic effects, one observer may see the temporal phase to be larger or smaller than another observer. The theory of relativity in physics explains the relation of expanding and contracting phase structures for different observers in different inertial frames of reference. The understanding of phase structure of intervals is also important for understanding the relation of minimal methods as meta-essences.

In the interval between two perspectives, there is a phase structure. This phase structure may be shifted in relation to each perspective. In spacetime, this shift may cause the space phase to be displaced in relation to the time phase or vice versa. The two phases are separated by a point of reversability[136] which is the area where the transformation from one phase to the

[136] Cf Merleau-Ponty’s ’Chiasm’ in Bib#

other occurs. The phase structure can be looked at in two ways. If space is emphasized, it is spacetime, whereas if time is emphasized, it is timespace. Minkowoski timespace model is an example of how time can be emphasized over space. Light cones distinguish past, present, and future, whereas the spatial component is reduced to the "nowhere" of nonoverlapping light cones. In spacetime, the time component is one dimensional whereas in timespace, it is the space component that is one dimensional. The important thing here is that the interval relates two perspectives, and there are always two ways of looking at these two perspectives. Minkowoski's spacetime model is best for thinking about casualty in a four-dimensional plenum. Einstein's spacetime is a better model for thinking about communication and synchronization. These models emphasize either the linearity of time (timespace) or the distancing of space (spacetime). Both allow the phase structure necessary for relativity to appear within the interval. One model emphasizes one kind of phase as primary, whereas the other emphasizes the other kind of phase.

When we consider the minimal methods related to event/data, or time/space, the phase structure of the interval plays an important role. The two methods which appear here are called “data mutation” and “design element flow.” Data mutation is the most basic of methods for determining whether software is working properly. We enter print statements into the program and check their changing values as the program runs. Data mutation may be observed by the program itself. A threshold is set up which will cause a control signal to be generated when the data values cross a crucial threshold. The streams of changing values in different variables is a fundamental way of looking at a program which emphasizes linearization. The other method which serves as an introspective observation channel is design element flow. This minimal method concentrates upon the flow of design elements like pointers, counters, timers, etc., through system states. The emphasis is not on instantaneous values, but upon correct operation within a particular system state. Most designers consider design elements as clockwork-like mechanisms. These clockwork mechanisms work together in a static gear work fashion to manipulate input/output data. In fact, design elements flow through system states just like input/output data. With design elements, it is the relation between various kinds of elements which determine how the system works. A system needs several different design elements in order to work. It is the interlocking mechanism which introduces a certain semantic distance into the system. The more complex system will have a greater diversity of design element types working together. This can be seen by differences in Halstead metrics which compute total operators/operands versus unique operator/ operands. This semantic diversity, together with the complexity of system states, determines paths of design element flows. The design elements crossing system state boundaries introduces distance into the software system. The distance is the opening up of difference between diverse kinds of design elements working together in an interlocking fashion which changes as system states change. Distance here means similar/ dissimilar and near state/far state. The treatment of dissimilar types of design elements working together in the same state, and the treatment of the same element across various system states, is how distance appears. Distance is the opening up of difference in terms of differentiation.

Note that both data mutation and design element flow are two different ways of looking at the same thing. Data mutation may consider several different data object values instantaneously, or the history of a single data object. Design element flow may consider the state changes of a single design element or the differentiated set of design elements present working together in a single state. These are diachronic and synchronic views. When design elements are considered as merely data objects with instantaneous values, then data mutation has precedence. Every design element has, as its basis, a data object. Design elements flow is the event view of data, whereas data mutation is the data view of event. Design elements emphasize the systemic events that happen to design elements. Data mutation emphasize the data changes that are brought about by systemic events. The data-centered view watches the data objects themselves and looks for instantaneous changes. The event-centered view considers thresholds of system significant changes to key design elements flowing through system states as important.

The crucial difference between these two methods is the emphasis of one phase over the other. One is a timespace emphasis, while the other is a spacetime emphasis. Data mutation emphasizes spatial aspects of the software system. Data is kept in different memory locations, and what is printed is the instantaneous value. Input/output variables are emphasized. This is a timespace view. History is important. Design element flow is the cartesian product of system diversity and system state diversity. Thresholds of values are important, and internal variables are emphasized. This is a spacetime phase space. The internal differentiation of values within system thresholds is important.

A way of perceiving the difference here is to examine the distinction between system dynamic simulations and discrete event simulations. Both are performed on digital computers. In system dynamics, it is the deltas between system variables over time that is important. In discrete event simulations with queues, it is the interlocking of system components that is important. Both are dynamic simulations, but these are two very pure examples of the difference between data mutation versus design element flow views of a system. The system dynamics simulation looks at a system as a series of attributes which change in relation to each other over time. The internal connection of the attributes is not considered, but only their outward behavior in concert. The discrete event simulation sets up structures similar to the system under study and allows the individual behaviors of discrete elements to add together to make up an overall system behavior. In discrete event simulation, it is the internal relation among system components that is important, more than the instantaneous readings of externally observed system attributes. One is an approximation working from the outside in, and the other is an approximation working from the inside out. System dynamics (outside -- in) emphasizes the history of attribute values read instantaneously (linearization). Discrete event simulation (inside-out) emphasizes the relation among internally differentiated system elements working together to create, as a side effect, overall system behavior (distancing). System dynamics has the premise that the whole of the system can be characterized by itself. Discrete event simulation says the system is the sum of individual component behaviors. Considering the examples of different types of simulation, it is possible to see that data mutation sees each data object as a disconnected system attribute. The history of these values over time is an external view of the system history. Design element flow gives an internal picture of individual system components interaction. This interaction, when summarized, becomes a picture of system-wide actions. This is a restatement in terms of software systems of what Arthur Koestler called the Janus[137] like nature of systems components. They are holons which act as systems viewed from below in the control hierarchy, and act as components working together when viewed from above. The conception of the holon is precisely the area of reversibility between the two phases of the interval. Linearization of the holon is related to an external, or view from below. Distancing of the holon is related to the interval, or a view from above, that sees differentiations interacting and working together to form a whole system. The system is a holon composed of holons.[138] Each holon is both component working together with other components, and a whole system itself. Meta-

[137] See Bib#.

[138] See Bib#.

essences describe holons from different perspectives. Where essences describe the variation of objects and their limits of variability, meta-essences describe the variation of holons and their limits of variability. Holons are the components of systems. They are subsystems. A software theory must strive to create holons that work together and also function independently as systems. Holons organize the system gestalt where a certain set of system objects appears. In another gestalt, a different set of system objects appears. The holons are the transitions between gestalts. They control localized transformations of objects between gestalts. Holons are not objects themselves. Software theory strives to construct holons because it is that which will make a software text into a software system. The interaction of holons are the meta-system. Design heuristics strive to impart paths of thought that will result in holonomic structure.[139] Design heuristics are the most important aspect of design methodology. Through heuristics, we attempt to learn how to create holonomic structure. These holonomic meta-structures exist in the interstices between meta-models and meta-

[139] See Bib#.

structures. They cannot be separated from the intertwining of meta-structures and meta-models. They are seen in the elegance, simplicity parsimony of systems design. Where software theory is nonrepresentable, software holons may be glimpsed by passing back and forth between linearizing and distancing methods and may be indicated with heuristic aphorisms.

◆ ◆ ◆

At this point begins a derivation of the methods of software engineering. By derivation is meant a step-by-step explication of the relations between the methods, starting from primitives which are related to single perspectives that are combined in order to allow the bridging of the gaps between perspectives. Minimal methods are defined, and then finally the holons that lie between minimal methods are indicated. Derivation is a difficult process because it goes against traditional approaches to the development of methods which are normally ad hoc. Derivation treats the method “space” in a systematic and rigorous fashion. Because of the complexity and length of exposition needed by full derivation, only an outline will be presented here. In the outlined derivation, only the major structural elements will be treated. The treatment will attempt to place these methods in the context of the formal-structural system while building up all the major structural elements necessary to deal with the interrelation between methods fully. A series of entity-relation diagrams will be used to build up the relations between elements of the methods, step by step.

Let's begin with an overview of what a system is to us before we attempt to be explicit about the definition of perspectives. A system is a gestalt of interrelated forms. As such, it has a foundation that depends on both formalism and structuralism. Formalism is a set of rules for manipulating contents. The contents remain unspecified within the formal system. Only the rules and their relations to each other are important. Structuralism is a subformalism in which contents are classified and become bound by their own rules. By shifting levels back and forth between formalism and subformalism (structuralism), functional changes are represented as transformations. The formal-structural foundation of the system allows the gestalt to be modeled explicitly. The gestalt itself is, however, always more than can be captured by a formal-structural model of a system. The formal-structural model of a system that appears within the horizon of the world is always a gloss or an abstraction. Systems are intrinsically both temporal and spatial in their differentiation as natural complexes within the world. The fact that they appear within the world means that they may be observed by multiple observers. It is from the observation of a system gestalt that Klir's epistemological levels arise. These are balanced by the ontological levels already described. The observer uses the epistemological levels in order to focus on different aspects of the system gestalt. The observer applies the formal-structural matrix to the observed system gestalt in order to get a dynamic model that can be understood theoretically. There is no limit to the sophistication of theoretical understanding. An infinite regress of meta-models and meta-structures linking models assures this. Behind this infinite regress may stand the singularity of pure immanence as an unattainable limit.

What has been described is the superstructure which allows systems to be viewed theoretically and philosophically. This superstructure is essential for systems science and systems philosophy, but is inessential for the development of an approach to methods. For the development of an approach to methods, the concept of meta-system is important.

A system may be described in purely formal-structural terms. In this case, the attempt is to give a present-at-hand description of a ready-to-hand entity. However, a meta-system contains a singularity of pure immanence and is perspectively fragmented. This means that a meta-system breaks up the community of possible observers so that each observer must take a particular position in relation to the system in question. The meta-system structures the community of possible observers. The meta-system encompasses the infinite regresses of meta-models and meta-structures, giving them coherence as they revolve around the singularity of pure immanence. The perspectives present qualitatively different views of the system under study. Perspectives see a particular set of qualities to the exclusion of other qualities. This set of qualities acts as a filter which enhances certain aspects of the system and causes other aspects to be veiled. This qualitative filtering is an important phenomenon. It cannot be accounted for in formal-structural terms. Qualitative filtering gives an aesthetic dimension to the study of the meta-system which cannot be seen in relation to the system alone. Design elegance and optimum solutions enter into consideration by this door. The perspectives available in relation to the software meta-system are DATA, EVENT, AGENT and FUNCTION. Data means information stored in a specific place in memory. Event means temporal modulation of signals. Agent means different locuses of independent action. Function means specified transformations. Software meta-systems organize the observation of formal-structural systems.

The observation of systems must account for their gestalt-like character. As such, each system has a set of facets. The observer may focus on any given facet conditioned by a particular perspective which will alter the quality of the facet. A given facet is conditioned by the environmental context to which it responds. The environment, which is the world from the point of view of a particular system, contains many contexts. Contexts interpenetrate within the world.[140] This means that many different contexts apply simultaneously to a particular system without necessarily interfering with each other. This interpenetration of systemic contexts is called their holoidal character. They are, as George Leonard says, like holograms.[141] For the observer, these contexts are seen as different roles. The world has a set of environments depending on how many systems it appears to contain. An environment has a set of contexts which interface with the facets of any particular system. The observer has different roles related to the environmental contexts. The observer has four qualitatively different perspectives which will allow him to focus on a particular facet. The system synthesis organizes all the facets of the system. The focus highlights synthesis as it controls a particular facet that is the current center of attention. Systems also have modes of operation. A mode is a select set of system responses keyed to an environmental context.

[140] See Bib#
[141] See Bib#

Beyond the gestalt nature of the system, it is also important to note that systems have sub-systems. A subsystem is a set of systems that cooperate to make up an overall system. Subsystems are holons in the sense of Koestler in that they have a Janus face. They appear as parts working together from the outside, and as complete systems from the inside. Systems may or may not be seen to be composed of subsystems. This depends on whether the gestalt contains sub-gestalts. These sub-gestalts may or may not be related to the meta-system. For instance, a purely formal-structural subsystem may exist next to a subsystem with a meta-system within the same over arching system. However, if a system has both a meta-system and nested sub-gestalts, then the meta-system organizes the nested sub-gestalts.

◆ ◆ ◆

The meta-system extends the system in two directions. The meta-system allows the articulation of supersystems and subsystems that act together to form a whole. The meta-system also allows the differentiation of perspectives from which the qualitatively different facets of these systems might be viewed. Within the meta-system, the behavioral synthesis of the system becomes a harmony of subsystems each with their own synthesis and the perspectives of multiple observers. The perspectives of multiple observers mediates between the articulated subsystems and supersystems which bracket the system itself. Meta-systemic harmony is an important concept that gets little treatment by general systems theory. That theory sees systems as random collections of attributes without even the coherence of a gestalt. So how could they go on to deal with the harmony of meta-systems? This lack of insight, or short-sightedness, is a symptom of our cultural disintegrity. The inability to see harmony in meta-systems leads directly to a kind of cognitive dissonance that shows everywhere in our environment. The best discussion of harmony must come from the study of cultures which were not so blinded. Chung Ying Ching’s article "On Harmony as Transformation"[142] is an excellent study in the meaning of harmony. He distinguishes four grades of harmony:

[142] See Bib#.

A. Logical consistency	**Formal system**
B. Interactive relation	**Structural system**
C. Mutual support	**Meta-system -- Holon**
D. Interpenetration	**Meta2-system -- Holoid**

Ching’s first level of harmony is that which occurs in a formal system. Within a formal system, logical consistency of rules governs the harmony of propositions. The second level of harmony is that of interactive relation which can only occur when time impinges on the formal system turning it into a structural system. The third level of harmony goes beyond the formal and structural system. It is only captured by the meta-system. This is the mutual support by holons (Arthur Koestler) within the same system. Holons are systems within systems. For this reason, they can only be described adequately in terms of meta-systems. The fourth level of harmony is the interpenetration of mutually supporting subsystems. This is described by George Leonard in the The Silent Pulse as the holoid.[143] The best exposition of interpenetration is by Francis Cook in his book on Hua-Yen Buddhism. Interpenetration may be described as a meta^2-system phenomenon which is generally demonstrated by referring to holograms. Holograms are lightwave interference patterns in which the whole form is contained in every part in potentia. The Holographic Paradigm by Ken Wilber explains these concepts.[144] This definition of the different levels of harmony is important for the understanding of systems and meta-systems. It gives a view which has been totally neglected within Western science and engineering. This view is beginning to become important with such movements as Deep Ecology.[145] It is a view which attempts to understand the harmony we experience, but generally ignore, within the world. This harmony holds systems as holons together in mutual support to form meta-systems. It holds meta-systems together through mutual interpenetration to form worlds or meta^2-systems. Harmony is the

[143] See Bib#.
[144] See Bib#.
[145] See Bib#.

aesthetically perceivable guide which should inform our systems design and construction work. This aesthetic dimension is generally lost because engineers do not consider themselves artisans in the same way as architects. For the software engineer, we might wonder what interest there would be in harmony when his product is never seen. Yet, what Alexander calls the "quality with no name" (nb the Tao[146])[147] pervades everything that we create as human beings. Even software design must take seriously the inner harmony of the systems it produces in order to achieve elegance, simplicity and optimality. These can only be achieved by taking into account the harmony that becomes apparent in the relations of systems to each other. Mutual support is only made visible by comparing wholes to each other. Interpenetration only becomes visible when it is realized how differences are exactly what makes it possible for things to be the "same." As Heidegger says, there is a belonging together among things that are the "same" which is different from and richer than the concept of identity.[148] Interpenetration allows the mutual support of the details of

[146] See Bib#.
[147] See Bib#.
[148] See Bib#.

different subsystems to be grasped as a holoid, where those differences allow the greater whole to be realized in actuality.

In our exploration of methods, it is precisely this perception of harmony which must be brought out into the open. This is because it is that aesthetic perception of harmony which counteracts the fundamental nihilism of technology and meta-technology. Because meta-technology deals prominently with meta-systems that may be seen as holons, it is possible to glimpse the non-nihilistic distinction through the veil of nihilistic erratic change and excrescences. The technical system is also a human system which has the possibilities of realizing the non-nihilistic distinctions in the midst of nihilism. This makes the agenda of deep ecology the innermost possibility of the technological system.

◆ ◆ ◆

Software engineering is taken here as the exemplary praxis of our age. Its essence is nonmaterial production. Work has always been understood before as physical work that involved rearranging material matter to some end. Software engineering rearranges gigabytes of ASCII characters whose physical representation is perfectly maluable. This exemplary praxis changes the nature of all other praxis. Blue collar traditional manufacturing work is transformed by computer literacy. Production of physical products begins by programming the production line machines. This is most significantly borne out by the idea of the human Genome project. In this project it is suggested that the entire human genome may be read. The secret book of human life written in the DNA spiral in each of our cells might be unlocked. The DNA code could then be used to reprogram human development. Thus, we intend ultimately to reprogram ourselves. We see ourselves essentially as the programmers of everything in the world, including ourselves. Selective breeding is replaced by genetic engineering which is essentially a kind of biological software engineering.

So software engineering is an important example of how the technological world is being transformed in our time. As such, it is interesting to note the relation between nihilism and non-nihilistic harmony within this new discipline. Nihilism is intensified at the meta-level of Being ^{3}, Hyper Being. Nihilism becomes the active destruction of meaning in the technological society. Within this bleak and dire picture, we note the arising of the importance of harmony. Disharmony and strife in the world highlight the need for harmony. We experience harmony in stark contrast to the excrescence of erratic change or noise produced by the dynamic formal-structural system. I remember an exhibit at the 1960 World Fair that showed a single machine a mile long crawling through the rain forest of the future (which is now). In one side went everything living and dead from the rain forest in the machine’s path. At the other end came out a road, and on the road trucks carrying the transformed rain forest. This machine is the mythical image of the dynamic formal structural system transforming wild into tame in a single all encompassing transformation. Now with holes appearing in the O Zone, the “myth of the mega-machine”[149] is slowly being replaced by furtive searches for harmony in a world which is fast on its way to becoming like Venus -- uninhabitable because of excessive

[149] Nb. Toynbee

greenhouse effect.

In software engineering, a realm which is safely within the technical establishment, there is no pollution. We use such a small amount of power that solar cells could furnish all the necessary energy. Now we waste paper by the ream, but these printouts are ultimately unnecessary. We could achieve the ideal of paperless environments if necessary. Thus, every way you look at software engineering, it is a kind of production that has risen above the dirty industries, i.e. polluting and resource destroying, of the industrial era. Post-industrial production, like software engineering although it is within the technical sphere, apparently has a radically different nature from traditional production. Software production is clean. It is the traditional industries controlled by software that are still dirty. And software helps those inherently dirty industries to be cleaner and more efficient. Retooling no longer means remachining. Resource utilization may be monitored. Pollutants emissions may be monitored. Software allows traditional production to tighten its control on waste and inefficiency. Software exerts an influence on traditional production, which is subtle and profound, toward a cleaner, more efficient, more harmonious working of the myriad different parts of the industrial complex.

We might say that machines display the harmony of logical consistency (A) embodied by the formal system. Chemical processes display the harmony of interactive relations (B) embodied by the structural system. In interactive systems, catalysts which facilitate but do not enter into interactions are possible. Chemical and mechanical industries have been the traditional productive centers of our economy. However, it is only software which makes possible a harmony of mutual support between elements of the chemical and mechanical industrial complex. In the past, careful design has brought these elements into close interaction, as in the automobile. However, actual mutual support with multiple feedback relations between parts was primarily a side effect of design rather than being intrinsic to the system. With the advent of software, the mutual support is intrinsic to the system. This software-based harmony in airplanes is called “fly by wire.” It is only the electronic signals between cooperating processors that allows the plane to fly at all.

For instance, it is said that some aircraft with forward swept wings could not fly without computer-controlled compensation. Thus, software introduces the possibility of intrinsic mutual support between subsystems within the meta-system. In this way, the artificial meta-system becomes a reality only with the advent of software.

It is the holonomic harmony of the meta-system which is the ultimate aim of software. Thus, the aim of software methods must be the definition of the holons which make that harmony realizable. Methods, as formal systems, cannot capture these holons. They show up as the heuristics which are the real heart of any method. Instead, methods in their interaction with the structural system of the software itself, make the holon/heuristics visible for the first time. In our attempt to derive our methods rigorously, we must keep in mind the goal of making these heuristics visible. Methods are meta-essences of systems. Systems are composed of objects with attributes. These objects have essences in the traditional sense (attributes, which if varied past a certain limit, the object is no longer the “same” object). Systems compose meta-systems which are the active environments of the subsystems. Meta-systems exhibit the harmony of mutual support between subsystems. The presence of mutual support is the sign of a meta-system or active environment. The subsystems within a meta-system are holons in Arthur Koestler's sense of being both parts and wholes simultaneously. It is software which makes mutual support an intrinsic rather than an extrinsic attribute of the meta-system. Software operates within the realm of Hyper Being [3] which is governed by the essence of manifestation -- the singularity of pure immanence. A meta-system with extrinsic mutual dependence is not governed by the action of the singularity, whereas the meta-system with intrinsic mutual dependence must be so governed. This makes us think about all organic creatures which are based on DNA coding. All such creatures achieve intrinsic mutual dependence by carrying a copy of the master software in the nucleus/cpu of each cell. This implies that all organisms operate on the level of Hyper-Being in some aspect of their existence and are governed by the law of pure immanence. It is obvious that mechanical and chemical processes play a large part in the functioning of the organism. The formal and structural relations of the mechanical (physical) chemical processes are not recognized to be controlled explicitly by the software of the DNA strand. At this point the physical and chemical analogies break down, and a new software metaphor must be used. Biologists are only slowly realizing the implications of this change in metaphor. The body is a multiprocessing distributed system where each cell is an holonomically independent/dependent processor. We realize the truth of this when dealing with neurons, but falter when it is necessary to extend the metaphor to all cells of the organism. The question is, how does the singularity manifest within the functioning of the organism?

Within the meta-system exhibiting intrinsic mutual dependence achieved on the basis of software, there is a fragmentation of perspectives that is necessary because of the presence of the singularity. With respect to software, this fragmentation appears first in the separation of space (memory-data) and time (cpu cycles-event). Then later there appears the separation of pointing (agent) and grasping (function). We will start by looking at the first separation more closely. As has been shown, these exhibit methodological distinctions (in Klir’s sense) of full metric ordering (distance + linear order). Thus, these two views are easier for us to understand since we are used to dealing with the full ordering of the x-y-z-t coordinate scheme from intermediate level mathematics. As also noted above, spacetime/ timespace is originally a single entity which may be viewed in two distinct ways. These two views inform our separation of the time and space views. There can be no total separation of space from time, only shifting of perspectives in such a way as to emphasize one or the other momentarily. Our four basic perspectives are not totally divorced from each other, but grow out of each other in a way that is mutually reinforcing. Mutual support is real and basic. It is “actual” in the sense of Kubler in his book The Shape of Time.[150] “Actuality” differentiates into the interval which can either be viewed in terms of timespace or spacetime. These are further abstracted into space as data and time as event.

What we wish to do is present first the pure space/data perspective and then the pure time/ event perspective before blending these to get

[150] See Bib#.

the minimal methods that act as a bridge between these two pure perspectives. This is the next exercise in the derivation of software methods. The derivation process aims to construct the minimal methods in such a way that the holon/heuristics may be glimpsed in the exchange between minimal methods on the bridge connecting pure viewpoints. However, it must always be kept in mind that the “pure perspectives” are a hypostasis from something which is organically bound together. For instance, “spacetime” and “timespace” are meta-views which allow the ideal views of “time” and “space” to appear as distinct. But what is spacetime/timespace before the meta-views came into existence? What is the “actuality” of spacetime/timespace? It is obviously buried deep in the singularity. This is exactly what the singularity is not showing us. There is a lost, -- unrecoverable origin which is always already lost. The singularity is the dark face of this origin. We construct primal scenes to cover over this primal lostness which explains it away. But the origin of spacetime/timespace before the Big Bang is lost forever, even though in a sense it is ever present. Like the relics of the Big Bang, we constantly live in the midst of the actuality of the origin of spacetime/timespace. The singularity is the dark face of the always lost origin of the four perspectives. We live within the nexus of their unfolding. Experiencing that unfolding we are what Heidegger names “Dasein” instead of subjects. The objects caught in that experience of unfolding might be called “ejects” because of their being thrown along with us. As Heidegger says, we realize our essential nature as falling within the world toward the abyss of groundlessness. It is within the ready-to-hand modality that Dasein relates to ejects within the world. In the present-at-hand mode, we are again subjects safely relating to objects. Subjects and objects do not experience the unfolding from the lost origin. They are static entities. Perfect specimens for nineteenth century science. However, once we begin to recognize systems and become observers of systems, suddenly the world is dynamic again. Those systems are not like objects -- as Klir would lead us to believe. He makes the minimal number of assumptions in order to avoid controversy. Systems exhibit formal and structural coherence wherein appear the harmonies of logical consistency and interactive relation. Yet, systems also appear as subsystems within meta-systems. When these meta-systems are purely extrinsic, then we can say that the meta-system is projected as an abstraction on the subsystems. In this case, the meta-system is an artifact. However, when the meta-system is intrinsic, then it must be held together with an ontic structure like software. In organisms it is the DNA spiral, and organisms are the exemplar of the dynamic interdependent mutually supportive meta-system. In fact, Whitehead used “organisms” as the metaphor in his process-based philosophy which attempted to describe what has been called “ejects” above. The meta-system corresponds to the meta-observer (meta^2 subject). The meta-observer is the observer observing himself, or the thinker thinking about thought. This is the final refuge of reason. It is a refuge in paradox which encapsulates the singularity in an analogous way as the meta-system. The best exposition of this is The Tain of the Mirror by Roddlphe Gasche.[151] A good discussion also exists in The Sociology of Meaning by John O’Malley.[152] Reflectivity and self observation is a manifestation of the meta-system which will not be dealt with here. See also the

[151] See Bib#.

[152] See Bib#.

Reflexive Thesis by Malcolm Ashmore.[153]

◆ ◆ ◆

Understanding the layers from Process Being as manifestation out to the full expression of the world is important for our study. Heidegger speaks of the world as a “clearing-in-being.” We can think of it more as a vortex which, like a galaxy, has spiral arms. The arms form successive veils that obscure the center of the galaxy. The center of the vortex is the point of manifestation -- pure upwelling of existence unfolding like a fountain -- out pouring. This is pure ecstasy. "Humankind cannot bear very much reality."[154] So each layer building out from the empty center of the vortex is a veil covering up that reality we cannot stand. The first layer is our experience of reality as Being. Being as C. Chang has said, is "a subtle form of clinging" almost exclusively the sole province of Indo-European languages. In all these languages, the conjugation of the verb of “Being” is the most irregular. This is because the concept was forged from multiple roots with similar meanings, all indicating “abiding”

[153] See Bib#.

[154] See Bib#.

or “remaining.” This veil of Being, subtle clinging to phenomena, is so deeply rooted in the Western consciousness that we think of it as the foundation of the whole world. It is a surprise to us when we discover the groundlessness (the abyss) which is like quicksand beneath our feet. We discover ourselves to be falling, and other things (ejects) are thrown with us into existence. The confrontation with Being focuses us upon our own impermanence. The subtle clinging to things which Being represents is experienced as everything being torn from our grasp. At this level, our grasp of things in the world is most important.

The next layer of veils is where we have reified and made everything static. We separate ourselves from everything -- subjecting all things to our gaze. We are dialectically related as subjects to all the objects we have projected. We pretend to be transcendent subjects, untouched by the world, as if that could possibly reduce our vulnerability to death. Distancing is the watch word at this level. This is the level at which scientific technological subjugation of the world occurs. The distancing is represented by the pointing that creates the present at hand. Being is stultified by its separation from time. Being is no longer dynamic. This is the Kantian universe of discourse. Transcendental objects are projected as the ideal supports for this view of the world with God as infinite ideal connecting these two poles -- the third transcendental.

The next layer of veiling begins to become dynamic again. Only now the subject begins to be an observer of systems rather than the subjector of objects. From domination we move to a recognition of interactive relations. From the logical consistency of formal systems, we move to the harmony of structural systems based on interactive relations. Here, the observer sees the gestalts of system. This is a new entry into an engagement with Process Being in a completely different way. Here, Process Being supports the dynamism of the formal-structural system rather than the primordial encounter with Being. Observer and system are locked together in mutual definition. Suddenly the observer affects the observed system in ways we still find puzzling.

The next level of veiling appears as the reflectivity of thought thinking of itself. It is the observer observing itself caught unawares by Lacan[155] in the mirror stage. This is the meta-observer caught in self paradox so well described by Douglas Hofstadter in Godel Escher and Bach.[156] This paradoxicality of reflexivity is matched at this level by the less explored nature of the meta-system which may have the intrinsic harmony compared by software. At this level, mutual support appears in the environment of the system that becomes a holonomic subsystem.

All meta-systems ultimately interpenetrate to make up the world. Here, interpenetration is given C. Chang's interpretation as non-impedance (interference) of one meta-system with another. This is a primitive interpretation of interpenetration which will suffice in this context. A fuller treatment may be found in F. Cook's book on Hua-Yen Buddhism.[157] The world is made up of a myriad of meta-systems, both intrinsic and extrinsic. These meta-systems interlock to form a coherent world where systems simultaneously function in different meta-systems as subsystems.

[155] See Bib#.
[156] See Bib#.
[157] See Bib#.

The world is the meshing together of all the different meta-systems to form a concrete "de-totalized totality" as Satre would call it.[158] It is the totality of everything there is without any explicit totalizing principle giving it inherent unity. We experience our world as the manifestation of all that is in a semi-organized formulation we learn from childhood throughout our education. Different cultures within the global village have different twists on the same basic superstructure that governs all aspects of our lives. Historically, the divergences in “world views” was much greater, but slowly but surely a global culture is replacing those divergent views. The de-totalized totality of post-modern techno-culture is becoming more and more pervasive. The world encompasses the limits of our understanding. Multiple worlds used to exist side by side within the cosmos, but these are becoming extinct faster than our planet’s wildlife. The world is the outer limit of the vortex of manifestation.

The vortex of the unfolding of the world is a metaphor that allows us to explore the fundamental aspects the harmonic nature of

[158] See Bib#.

things. Each layer addresses a different kind of harmony, most of which were identified in the analysis of Ching Ying Chung. However, we need a way to see this harmony and analyze it in a concrete manner. This necessitates the development of a vocabulary which will allow us to talk about what is crucial at each level of the vortex. Each veil has its own necessity that must be respected if we are to deal with harmony in a concrete way rather than as a nebulous concept with no actual referents. The vocabulary to be introduced has already been shown diagrammatically. Working inward from the peripheral of the vortex, we encounter each of these terms in sequence. A brief explanation will follow.

The world’s limits are horizons within which all things appear. Meta-systems, or environments, provide contexts which to the reflective consciousness are situations. The observer has a focus on a particular facet of the system’s gestalt. Dasein is shown the truth of its own care, and the fate of the eject as they arise from pure manifestation which is the center of the vortex.

Each perspective within the meta-system

participates in these harmonic levels. Each perspective has its limits when it touches horizons. These horizons mark the interpenetrating boundaries with other worlds. Each perspective has a context with respect to the immediate environment of the system. The system has a facet that is focused on by the observer. The observer can also look within the system at objects which have attributes. We will not deal with the fated aspects of ejects since these have no psychological distance from us yet. They are transitional objects[159] which are not yet totally decoupled from Dasein. Dasein and eject merge indistinguishably like the sucked thumb. This is an important level of care for self and the fated appearance of the object as it is in itself phenomenologically. Our analysis of all perspectives really revolves around distinguishing the following harmonic levels as in Figure 52.

This vocabulary allows us to discuss for the first time the harmony of design for which we all strive. In the field of software engineering, this harmony is implicit in everything we do. It has long been recognized that simple, elegant

[159] See Bib#.

solutions are superior. Yet, we lack a way of expressing the aesthetics of design. With this vocabulary we can talk about each level of harmonization with an appropriate word, and we can see how the different levels work together in design.

Recognizing the important place of harmony in design is crucial to software methodology. Methods without a sense of harmony are empty tools without a craftsman to give them life. The aesthetic aspects of engineering are very important and little developed in this age. By building a vocabulary explicitly linked to the grades of harmonization, we can begin to explore this unexplored territory.

◆ ◆ ◆

The definition of the pure data perspective begins by recognizing the difference between “class,” “entity” and “instance.” A class is a set to which entities belong from which they could inherit attributes. For instance, the class “airplane” is a set of attributes such as wing span, number of engines, etc. This set of attributes is given coherence when mapped on to an entity. An entity is a generalized form that includes attributes as well as some other features. These features include the name, a set of relations with other entities, an assembly, a set of faces, and a set of slots. The name is a unique identifier for the generalized form. The set of relations, such as “has_one” or “has_set” shows how the entity fits together with other kinds of entities. The assembly is a set of parts that comprise the entity. Parts are other subentities of different kinds that, acting together, perform the actions of the entity. The faces of the entity are the external views that other entities may have of a specific entity. The set of slots are active equations or rules that recalculate if any values of attributes they are tied to change.

An entity is more than merely a collection of attributes. The entity is the concrete realization of an object which may exist within a system. This concrete realization is a general entity which has Being and persists through time. It has faces, parts, attributes, relations, and recalculating slots. As such, it can serve as a model of any passive object within a system. It represents the pure data view of an object within a system.

Such a pure data view sees the datum. The datum is the limit of the data perspective. It is the observed value of an attribute of the object. Klir's treatment is perhaps adequate for the observer who experiences the system as an unknown ensemble which he is attempting to reconstruct or model. However, for the software engineer who is attempting to construct a system by design, such an empty object is not adequate. Instead, we need a model of a general entity which can form the building block of our system construction. We know that within a system there are various kinds of entities in multiple interlocking relations with each other. Each of these entities belong to classes that express their kind. Kind here is expressed as a collection of attributes. Since Aristotle, attributes have been divided into essential and nonessential (or accidental). Essential attributes must be present for the entity to remain the same kind. Accidental attributes may be varied without affecting the kind of the entity. Here, kind is reduced to arbitrary sets of attributes. Kind is, of course, more than just sets of attributes. This is an external, minimal view fostered by analytic "anti"-philosophy. "Kind" has internal structure as well as being a collection of attributes. The “entity” fills this void by providing a general basis for the internal structure of any kind. The entity has relations with other entities which form a synchronic whole within the system. The entity also has an assembly of parts, giving it internal differentiation. Through its external relations and internal differentiation, the entity fulfills its role as the general model of an object within a system. Each entity may form the prototype of a whole series of similar entities of the same kind. These are the instances of an entity. An instance will replicate all the attributes of the original class of the entity. An entity may inherit from several classes at once. So the entity is the locus of inheritance of attributes. Within the particular instance, these attributes are called replicas. The slot is the active portion of the entity. The slot is the result of an equation recalculating. This equation is based on the current values of attributes. Through the slot, the entity is given a dynamism related to the assignment statement.[160] Through the equations related to slots, the inner structuring of the relations between attributes can be modeled. These inner relations are static but express the inner structure of the entity itself as

[160] See Bib#.

a data transformer.

The pure data view sees entities as nets of relations between different kinds. The entity is composed of an assembly of parts, and it expresses a structuring of relations between attributes that produce other attributes. The entity has different faces which are seen by other different entities. The entity is a general purpose construct for expressing clusters of persistent data in definite structures. These data structures will be called data “objects.” The entity is a design concept necessary for the theoretical construction of systems from the point of view of data alone. The concept of entities have gained prevalence through the emergence of object-oriented design in software engineering. Entities and their relations are modeled via Chen’s entity-attribute-relation diagrams.[161] A good introduction to this is found in Shaler/Mellor’s book on Object-Oriented Analysis.[162] However, our formulation differs somewhat from their’s because we need a more robust design representation for software systems.

[161] See Bib#.

[162] See Bib#.

◆ ◆ ◆

The event perspective is very different from that of data. Each system can be subjected to an event perspective. The key idea in the event perspective is the “signal.” A signal is a stream of pulses or a steady state of energy in differentiation from other signals. A signal implies a stream of energy whose differentiation remains fairly persistent. The other signals, along with the one under consideration, form a bundle of signals which have specific relations between each other. The signal needs the bundle of signals in the same way the “entity” needs the class to give it kind. Through the synchronic/diachronic unity of the bundle, the lacuna (blank spots) in the signal can be comprehended. The signal may either be broken by lacuna or modulated. These breaks or modulations, when they cross imposed thresholds, are events. An event is a change to a signal or group of signals that takes place at a specific point in time. Events have names and also have temporal relations with other events or lacuna. These temporal relations are represented in terms of interval logic in terms of before-after-during relations as defined by James F. Allen.[163]

X before y

x equal y

x meets y

x overlaps y

x during y

x starts y

x finishes y

These relations, plus their inverses, form the temporal logic of Allen

Interval logic allows time relations to be expressed without an objective timeline being established. This is important because an objective time scale is not always possible. In many systems, the temporal points at which things occur is not as important as the linear ordering in time. Interval logic establishes linear ordering without necessarily having to establish distance between time points. Interval logic allows us to relax distance to get

[163] See Bib#.

the pure linear ordering of time. Bundles of signals may belong to a sheaf of signals. The signal sheaf can express branching bundles of signals. This allows temporal, or modal, logic of necessity and possibility to be expressed.[164] The core of a sheaf can be interpreted as the necessary core of a set of possible worlds. A bundle of signals may branch at a particular time point to show two different routes of system evolution. The signals that have necessary modulations, or breaks, belong to the core of the sheaf unaffected by the branch which is common to all possible worlds. The sheaf contains all the signals pertinent to the system. In this way, the temporal view of the system is articulated as a series of events on signals within a sheaf of bundles of signals where there are both synchronic relations between signals and diachronic relations between intervals.

◆ ◆ ◆

This is a short introduction to the vocabulary and concepts which are necessary to understand the pure data and event views of systems. Every system can be understood as a

[164] See ??? {Temporal Logic}

kind of entity and as a sheaf of signals. Both entities and signals have the persistence necessary for Being. A system described as a sheaf of signals, or as a set of entities within the system entity, has the robustness necessary to be a full description from these particular points of view. However, at this point these perspectives are totally separate, even though they arose ultimately from the same lost source. Entities are static, and signals are dynamic. The attributes of an entity could be tied to a set of signals. Thus, the changes in attributes that cause recalculation of slots may occur. In this way, entities may appear to change over time by the changes in their attributes.[165] However, this type of linkage, which is the traditional one, is totally superficial. What is needed is a deeper fusion between the data and event perspectives.

Notice above that we linked signals to attributes. This suggests that the concepts of entity and signal are actually orthogonal to each other. In fact, when signals are attached to attributes, then these attributes may experience events. It is only the instances of an object that have attributes with values. Thus,

[165] See Figure 54.

there is formed a nexus where instances of entities reflect events in bundles of signals. This nexus is given the name “eventity.” By “eventity” is meant the same as Whitehead called the “organism” in his process-centered philosophy.[166] Its ontological foundation is the “eject” described above in relation to Dasein. The eventity combines fully the features of the data and event views into a single common element. The eventity is more like a process than a static object in a frozen world. The eventity is the nexus of the orthogonal connection of signals to the attributes of instances of entities. But the eventity brings to bare the entire structuring of the data and event views on a single nexus of transformation. Transformation becomes the key idea because the eventity is a dynamic process of unfolding where bundles of signals are actively gathered by the entities whose attributes they are linked to and control. As signals differentiate within a bundle associated with a entity, the associated attributes differentiate. Since parts are associated with sub-bundles, and since relations between entities can change after time, the eventity gives an excellent means of modeling the

[166] See Bib#.

dynamic parts of a system diachronically. The eventity is seen as a series of phases within an overall process rather than a static ever persistent object. Even though entities and signals are purely persistent, their combination can describe a dynamic nexus which is more like the “organism” described by Whitehead in Process and Reality.[167] It is bringing together the overall structure of the data and event views that allows this representation of dynamic subobjects of systems. The “eventity” contains:

grid location plus movement{spatial atributes}

start time plus extent {temporal attributes}

local timeclock

instances plus attribute’s replicas

branch and bundles of signals

signals attached to attributes

parts lists

[167] See Bib#.

classes

slots and equations

This is a fairly robust design representation yet it still lacks a crucial ingredient which is automation.

◆ ◆ ◆

The addition of the automation to the eventity is a key transformation from a means of representing the elements of systems passively to the active representation a virtual processor that can simulate the dynamics of a system. This is a move from the passive eventity to an active eventity, which is also an automation. The automation is added both to the eventity proper and to the assembly of its parts. Certain attributes are defined as input variables, states and output variables. The input variables accept a stream of signals, and the output variables generate a stream of signals. The output signals are based on the input signals and the states. This is the basic form of the finite state machine which is generally known as a foundation of computer science. The eventity becomes a transformer that transforms input signals into output signals. The slot, with its equation that fires, is the mechanism by which the automation is effected. So the need for including this mechanism in the entity becomes readily apparent.

However, the eventity also has an internal assembly of parts which must be coordinated. This is done by making the assembly an automation as well. The states of the parts together with the state of the eventity determines the reaction of the eventity and the activators of the parts. The working together of the assembly automata and the eventity external behavioral automata allows the eventity to organize the behavior of its component parts as well as to have a holistic behavior itself. The eventity contains both accepting automation (the assembly) and generating automations (the eventity state machine) in a single structure in order to effect its role as a holon. From the outside, it has the behavior of its state machine, while on the inside, it is organizing the behavior of its parts with the assembly automata. The assembly is a cybernetic mechanism which is married directly to the automation to produce a self-governing mechanism that at the same time controls its parts. This is an "autonetic," or self-controlling, mechanism. Thus, the eventity, augmented by dual finite state machines, may be called an autonetic eventity. As such, it is composed of the following parts:

AUTONETIC EVENTITY

state list {Sn}

current state {Si}

input mapping {I}

output mapping {O}

reaction mapping {R}

activation mapping {A}

automata definition {Ixs}

assembly definition {PSxs}

◆ ◆ ◆

The "autonetic eventity" is a fairly complex theoretical structure. It combines the conceptional structures of the pure data and pure event views of the system with the dual mechanisms of the automation and the cybernetic control structure. The autonetic eventity stands as a combination of the temporal and spatial viewpoints on a system. These two fundamental viewpoints arose from the always lost origin of the singularity and separated at the level of Dasein/eject to become totally reified at the level of subject/object. Within the gestalt of the observer/system layer, the entity and signal appear as the persistent focus of these perspectives. We discern the independent structuring of concepts within each perspective and how these can be recombined into a synthetic unity of the eventity which is equivalent to Whitehead's organism in the process philosophy. The eventity combines the persistence of the entity with the temporality of the events. By taking this hybrid structure that allows the perspectives to interface and adding inward and outward automation, we gain a general purpose simulator for dynamic formal/structural systems. A system may be viewed as an ensemble of autonetic eventities. This ensemble is itself an eventity with its own assembly called a "synthesis" by which the parts of the system are controlled and coordinated.

This view of the system as an autonetic eventity composed of autonetic eventities provides a foundation for beginning to attempt to understand software methods. A software system is a good example of a dynamic formal-structural system. The concept of the autonetic eventity can be used to analyze the design of this kind of dynamic system. Design analysis and design synthesis are dependent on software methods. For only with software methods can we attempt to understand the dynamics of the system at a level of abstraction higher than the execution of the code itself. Software methods are a representation at the "system" level of abstraction. This level of abstraction is above that of the higher level languages commonly in use today. "Ada" and "C" are examples of these higher level languages. They are idiosyncratic in their construction. They contain many features beyond the three constructs necessary for structured code (branch, iterator, sequence). They do not deal with the software system as a whole, but merely define each statement. The interaction of all the statements, together with inputs defines the software ensemble. Whether these statements act together as a system is difficult to discern by looking at the sequences of statements alone. This is because of the fundamental problem of non-locality.[168] Software design elements may have to be spread throughout the code to achieve a certain effect. Unless all the places in which the design element appears is known, along with how they work together, the effect of the de-localized design element cannot be known. Design methods attempt to deal with non-localization of design elements by creating a level of abstraction above the code where de-localized design entities may be conceptually localized. Localization insures that the conceptual structure will not execute. It is not executable in the sense of code statements. However, the design elements at the more abstract level make the multiple de-localizations at the code level comprehensible.

This level of abstraction is where entity and signal appear. It is the level at which the software system might be represented as a set of interacting autonetic eventities. Since the autonetic eventity is foreign to our usual

[168] See Bib#.

concept of a design element, it will remain useful to speak in terms of the kinds of design elements that normally appear in design as well in order to make our point clear. Design elements are normally thought of as ensembles of statements that work together to perform a single duty. For instance, a queue may be a design element which is used in many places in a system for different ends. The design element has a whole range of forms and levels of abstraction that it operates at to accomplish its work. The design element may be something as lowly as a pointer, counter, timer, buffer or as large as a data base or communications protocol. Like the code, the traditional software design elements form a diverse set of abstractions that are highly idiosyncratic. Thus, the concept of an autonetic eventity as a higher level design element serves as a means of finding the middle level of complexity and abstraction to refine our concept of the design element.

We expect software methods to allow us to focus in on the way design elements work together to form a single system out of diverse elements. The system is a gestalt which works by combining diverse parts in coordinated efforts to achieve common goals. Understanding the system as a whole, by looking at this cooperation of design elements, may only be done by using software methods at the proper level of abstraction. It has already been pointed out that methods appear as the relation between design points of view. Here we are dealing with only two points of view: event and data. We expect here two minimal methods to arise in order to let us see data from the point of view of event and vice versa. We are focusing in on these two minimal methods in order to get a sense of how to derive all the other minimal methods which relate all the other permutations of viewpoints.

The two minimal methods that arise at this point are called “data mutation” and “design element flow” methods. They arise from the relation of the autonetic eventity to the software system as a whole. For instance, the design element flow minimal method arises in order to formulate the relation between system states to design element states. This has two forms. One form shows the multiple transitions of the eventities in relation to system states, while the other shows the multiple eventity states that are the nodes of system transitions.

In this way, the overall coherence of the software system can be designed and maintained.[169]

These two forms of the design element flow minimal method allow designers to formulate the coherence of design elements working together. Here, event views data as flowing design elements. The events are the transitions (either eventity or system), and the data are the state values (either eventity or system). The method focuses on a crucial aspect of the eventities working together in a coherent way to give the impression of coordinated behavior. This method focuses on the coherent relations between state machines at different levels of abstraction. It could just as well be applied to the relations of the eventity automata and its parts. It is clear that this method is directed at a particular aspect of the systemic overall functions of eventities working together.

The other minimal method that arises at this point is the information flow diagram of data mutation. The changes of system and eventity states are just a special case of the changes in the data components of software systems.

[169] See Figures 55 & 56..

Most of the attributes and slots of the eventities that make up a system will be in constant flux throughout the execution history of the software system. The data mutation minimal method addresses this other very important aspect of the functioning system. The data mutation minimal method has two aspects. The first is the trace of the signals attached to eventity attributes. As signals are modulated, they cause changes in the values of attributes of the eventity. These modulations may be recognized as events by the system when a difference that makes a difference to the system occurs. These differences that make a difference are information. They are no longer pure datum because the system makes a judgement as to the importance of a particular change. Information flow within a system occurs when data about changes that make a difference is moved around the system in order to cause reactions within the system. Events are recognized by using the slot equations. A change in a particular attribute when it passes a certain value causes a certain slot value to change, and this signals reactions by the state machine of the autonetic eventity. This may cause other attributes of the system to be altered. Only by watching the traces of the signals of general eventities together, can this kind of reaction be discerned. This is one of the most basic software testing techniques. Pick a set of relevant variables, and watch how they change as their values are printed to the screen. The information flow between program variables is the most basic way of verifying that a program is working correctly. As with the design element flow method, the data mutation has two basic dual representations.

The information flow diagram shows the traces of data mutation. The information network shows the flow of information between attributes divorced from the signal traces. The information flow within a system is the manifestation of the internal temporality of the system. The coordination of information flow within the system is crucial to a system's overall functioning. In fact, software systems are primarily characterized by information flows. This is why they are called information processing systems. In many instances, it is primarily the storage and manipulation of information which is the primary concern of the non-real-time software system.

We see here that information flow mapping and state coordination are the fundamental aspects of the system that are highlighted by these two methods. These two aspects work together to give the spacetime coherence of the system. The different variables that represent both states and variables must be spatially distinguished. They each change over time, one within a finite set of state transitions on different levels which the other varies continuously over wide ranges of values. These variations are monitored by the system and reacted to so that information flows through the system are related to input values. The causality manifest in the autonetic eventity is the combination of input values of attributes and the reactions of the state machines to these inputs. Thus, the two minimal methods encapsulate very fundamental aspects of the functioning of the software system.

◆ ◆ ◆

This derivation of the two minimal methods that relate data and event perspectives has been carried out in great detail in order to show how perspectives combine to give rise to methods. As a speculation, it is possible that these two minimal methods are related to E. S.

Bainbridge’s formulations of the dual representations of automata.[170] Studied from the point of view of category theory, one arrives at the dual of a category by reversing the arrows. Arbib and others have assumed that this is an opposite automata where time runs backward. Bainbridge makes the intriguing case that in fact, that the dual of the state machine representation of an automata is an information flow representation. If this is true, then it may well be that these dual representations of automata find their manifestation in these two minimal methods. The duality of automata representations would be a very good foundation for these minimal methods. However, it is not clear whether this connection to Bainbridge’s work is justified.

◆ ◆ ◆

A general discussion of the impact of this derivation of these two minimal methods follows:

Software methods address the whole software system. Each one gives a unique view of that whole. Each perspective as subject views the

[170] See Bib#.

other perspective as object. Thus, between each set of two perspectives on the software system, there are two one-way bridges. Associated with each of these bridges is a software method that combines elements of each perspective in a unique way. Thus “data mutation” combines time and event to produce information flows and nets while “design element flow” combine time and event in a different way to produce coordinations of state machines. These minimal methods are meta-essences. If objects have essences, then systems of objects must have meta-essences. An essence is the set of attributes within certain limits which an object needs intact to maintain itself. It has been said that essential attributes are distinguished from accidental attributes. A meta-essence refers to objects embedded within system gestalts. Meta-essences govern the presentations within gestalts. In a gestalt, a series of objects are presented in a certain order. Under certain circumstances, one object is brought to the fore for presentation, while others are submerged into the background. In another circumstance, other objects are brought to the fore. Meta-essences govern the presentational sequence in a gestalt. This presentational patterning is the heart of the dynamic formal-structural system. The patterning is the meta-essence of the system. It determines what is essential and what is accidental within the context of the system. It determines what objects are necessary and their ordering within the system. If a system has a single meta-essence, then it does not have a singularity. However, a system with a singularity has multiple meta-essences which are united by the singularity. The meta-system has the singularity as its core, while the system has the meta-essence as its core. Within the meta-system, the set of meta-essences organize the relations between perspectives. Meta-essences are thus very important in the study of both systems and meta-systems. The expression of meta-essences within software systems are as minimal software methods. They serve as bridges between perspectives and thus give unity to shattered perspectival fragments. Here we have focused in on the relation between data and event views of the software system. It has been seen how elements from these perspectives combine to give rise to these methods. The methods focus in on an aspect of the system which has global significance. What is true of these two minimal methods (data mutation and design element flow) are true of the other ten minimal methods as well. Each minimal method distinguishes the perspectives it connects, while pointing to a unique aspect of the global system.

What is important, however, are not so much the minimal methods themselves, as what is seen in the area of reversibility that is discovered by oscillating between them. As we develop systems, it is necessary to iterate between pairs of perspectives. In this oscillation, we glimpse the heuristics which guide toward harmonic design. Methods are only a language for representing system-wide design issues. They are a means for communicating design ideas and representing aspects of final design decisions. Methods usually include a set of procedures for applying minimal methods in order to elaborate the design. But methods do not tell you how to pull a good design out of a wicked problem which is multiplying constrained with no obvious optima. Methods usually include some heuristics that attempt to give some guidelines in this area. The heuristics are normally few and far between. They are the words of wisdom of very experienced system designers. An example from structured design of

Yourdon-Constantine is to stress cohesion and minimize coupling. This is a famous example of a heuristic for software design. These heuristics normally appear very sparsely in texts on methods, buried as pearls of wisdom among the methodological paraphernalia (such as how to draw the right diagrams and what are the steps to follow). But these heuristics are very important pointers toward the way to instill an aesthetically pleasing and practical coherence into the designed system. These heuristics cannot be represented within the methodology, but only indicated. In designing a system, one is attempting to instill a mutual dependence among the varying parts of the system. This mutual support and internal coherence is an aspect of the meta-system. It must be instilled despite the barriers erected by perspectival fragmentation. It is only glimpsed as one jumps from one perspective to another. It is a holon that exists in the gaps between perspectives. Thus, it is not captured by any one perspectives nor by the minimal methods that attempt to bridge the gaps between perspectives. These heuristics are knowledge of what makes systems harmonic and how to instill that harmony. As holons, they are glimpses of the wholeness of the system. As knowledge, they exist at the level of knowledge, not information. Information is built into the software system. Knowledge informs the design of the software system, but is not captured by it. The holon/heuristic is captured at the next meta-level of Being -- Wild Being. It is not captured at the level of the singularity in the meta-system. Artificial intelligence deals with the application of heuristics and knowledge representation. Software engineering encodes knowledge into the design of the system, but then throws it away during implementation.

The real work of developing software engineering methods is to attempt to capture these heuristics to guide design. They are seen by passing back and forth between minimal methods in design iterations. In order to make this discussion more concrete, I will suggest a possible heuristic to describe the holon that appears between the data-mutation and design element flow minimal methods.

Heuristic 1. Do not treat design elements as clockwork mechanisms, but treat them as flowing, transforming elements with their own life cycle within the life cycle of the system.

Heuristic 2. Minimize coupling and maximize cohesion of system entities by reduction of information flow across eventity boundaries.

Heuristics such as these attempt to tell you how to build a better system in the language of the methodology. They strive to fulfill the prerequisites of mutual support between parts of the designed system. They instill the harmony of mutual support into the system from the perspective of the meta-system. This harmony should lead to simplicity, elegance and aesthetic balance in the designed system. It leads to increase in what Alexander calls “the quality with no name” called by the Chinese “the Tao” or “the Way.” Software engineering needs to discover these heuristics that indicate holonomy -- the wholeness of systems. Today our discussions focus on concrete computer science techniques, or, at best, methods. The true heart of the software engineering discipline is the discovery of the heuristics that should guide our use of methods. This should be the focus of our research, giving life to our methods and instilling harmony into our software systems. As with all other disciplines, the most harmonious is usually the optimal.

◆ ◆ ◆

Up to this point the focus has been on those points of view associated with space and time. As was pointed out earlier, these points of view are easy for us to comprehend because they are fully ordered. Cartesian coordinates and the real number line allows us to give full ordering, in terms of linearity and distance, to all points in space and time. We are taught to deal with this illusory continuity in mathematics from an early age. This mimicking continuity found in the world, mathematically seems natural to us. So natural that it goes unquestioned. Unfortunately, we discover that digital computers have some difficulty mimicking continuity. It is necessary to construct elaborate work around like floating point numbers in order to maintain the illusion of continuity. The digital nature of computers leads us to view mathematics differently. Finite mathematics takes on a new significance. Calculus begins to take a back seat. It has provided the defining framework for the Western view of the world since its invention. Now we begin to explore the granularity of the illusory continuum it offered us as a haven for so long.

With respect to space and time, we notice that our descriptions of these fundamental categories need no longer mimic the continuity we find in the world at all times. In the artificial digital universe, it is possible to relax our simulation of continuity. This is done differently for space and time. With space it is possible to relax linearity to get partially ordered distances. With time it is possible to relax distance to get linearity without distance. In non-real time systems, these relaxations cause great simplifications to occur in system design. For instance in human interfaces, it is many times only necessary to get the sequence right, and as long as response times are not too long, it is not important exactly how long the response takes. Or again, in data base systems search trees may allow efficient retrieval of information which is not sorted into a linear order. The distances between information elements are just a few algorithm steps up and down the tree which would be a great distance from the perspective of a linear search. Multiple search trees for the same data base makes the stored sort order irrelevant. In digital systems, these and other similar relaxations of space and time constraints allow many useful information systems to be built.

However, real-time systems demand the full articulation of spatial and temporal order in order for the system to adapt to the events in the world. Meeting real world time deadlines and the spatial movement of information across a computer network becomes a crucial aspect in these systems. This is unfortunate because even though space and time views of the system can expand to handle these constraints, the other important viewpoints, which will be discussed next, cannot handle this expansion. These other viewpoints are agent and function. As long as we are not dealing with real-time systems, this discrepancy does not show up. The shift to real time systems opens a rift in the relation of software methods to each other which does not appear in conventional information systems.

This is important from the point of view of methods. The space and time-oriented methods we have just described are associated with the partial ordering of linearity without distance and partial ordering with distance. Partial ordering with distance is related to the design element flow minimal method by which state machines are coordinated. The linearity without distance is related to the data mutation minimal methods by which information flow within the system is coordinated. Information datums have no distance from one another. State machines are not necessarily sequential. These two minimal methods exemplify the digital system in a relaxed state, not necessarily coordinated with external real events in space and time. However, for the digital system to mimic external continuity, state mechanisms and information flow must be coordinated together. Information takes time to flow across the network, and state machines can be made to operate sequentially. For instance, the addition of an executive cycle to a real-time system is a crucial addition that adds a basic sequential aspect to a system that might merely be interrupt driven. The minimal methods of data mutation and design element flow must be used together to gain the leverage in system design necessary to produce an adequate spacetime representation of the real-time system. Because they must be used together, the oscillation between viewpoints (event and data) is guaranteed. This oscillation allows glimpses of the holons in the chiasm of reversibility that exists when one jumps between viewpoints. These holons are fleetingly represented by heuristic aphorisms that tell how to use the methods to achieve a harmonic design. Examples of these heuristics have been offered.

Methods give us a language by which we can talk about digital software systems design. Methods, used together, provide leverage to approach the difficult problems that arise when designing extreme systems. Real-time systems are extreme in the sense that the digital must adapt to and mimic the continuous. This goes against their nature. The advantages gained by the relaxation of time and space constraints is lost. The rift between minimal methods opens up. Seeing these rifts, we gain insight into the relations between the data and event viewpoints. The holons of harmony are glimpsed. This means that the language of methods lead us to a point beyond this language. Methods are meta-essences that make visible the harmonic cores of systems ensnared in meta-systems.

◆ ◆ ◆

Now it is possible to switch from the discussion of the data event perspective to the discussion of the agent and function perspectives. These two perspectives are fundamentally different from data and event. A similar derivation of minimal software methods will be attempted in this case as well. But that derivation will take on a completely different character. This character is motivated by the nature of the inherent ordering of these perspectives. These perspectives, agent and function, are inherently only partially ordered. We are not used to dealing with partial orders. They do not fit our training that always assumes full ordering. So when we are confronted with partial orders, we think there must be some mistake.

The real question is why are different perspectives given to different levels of ordering. We would expect them all to be the same. Perhaps this difference between perspectives is an illusion? Because we expect full ordering, as if it were "natural," we suspect there is something wrong with our analysis of these perspectives. Instead we should suspect our "natural" prejudice for continuity which is misplaced within the digital world of computers.

It should be clear by now that every conceptual structure has its dual. This is a fundamental "law" of category theory in mathematics. It should follow directly from this “law” that the conceptual structure just described that related the event and data perspectives requires a dual. That dual is the conceptual structure that appears between agent and function perspectives. As we explore this “dual” conceptual structure it will become apparent just how different the duals can become while still bearing an internal relation to each other. The dual of spacetime/timespace is not just an inversion of superficial properties. It is a fundamental inversion of deep conceptual relations. Because of this, the complementarity between these duals has a profound significance for understanding systems and meta-systems.

The place to begin is a general consideration of the lopsided nature of the two sets of perspectives. In the case of the spatio-temporal perspectives, the definition of the perspective itself was quite complex, while the minimal methods which connected them were fairly simple. For these new perspectives, exactly the opposite is the case. The definition of the perspectives is simple, but the connecting minimal methods are very complex. This shift in complexity is an important phenomenon. It shows that the inner relation between the dual sets of perspectives is very unusual. It portends a deep structuring by the singularity of pure immanence. The displacement of complexity in the dual, like the displacement of images in dreams that belie the unconscious, gives us indirect evidence of the action of the always already lost source of the dual images of the interlocked perspectives.

In the technical world, it is taken for granted that everyone is asleep to the significance of images and their relations. The text of the technical is rarely read (in the sense of hermeneutics). Both because of schooling and proclivity, engineers disdain the efforts of literary critics. Because of fear of lack of competence, the literary critics are held at bay. Thus, the technical and meta-technical worlds form a closed system, seemingly immune from criticism. Yet it is precisely here that the deep structures are revealed in the most extraordinary fashion. For instance, this study finds great significance in the relations between software methods. Surely these significances will be denied. Yet the archeology of knowledge should be able to treat any cultural artifact, finding human significance in any of them. In the engineering realm, this significance is denied more strongly than any other field of human endeavor. This alone should make us suspicious. Yet Heidegger opened up the technical to us[171] in Being and Time, giving it a deep meaning for the first time. This study seeks to do the same for the meta-technical. Meta-technology is the as yet unrecognized spinoff from technology. It is technology taken to its limits where it collapses into the proto-technical. The ultimate meta-technical event is nuclear war in which the super powers cancel each other, leaving only the residue of savagery -- if any humanity at all. This is the age in which nuclear war hangs like a pall over every event giving it a changed meaning. Software makes nuclear war feasible, even if it makes nuclear defense infeasible. Thus, the post-modern age is the meta-technical era where all meanings are changed by the possibility of annihilation.

◆ ◆ ◆

Space and time comprise the field of action within which the system functions. Together they form an opening, an open place, within

[171] See Bib#.

which manifestation occurs. Space and time together give a patterning to that open place that determine what can happen there. This intrinsic patterning has been reified into our mathematical metric which describe position and occurrences as "backdrops" (Klir) against which object attributes are measured. The openness of space and time, as a stage upon which action takes place, is of primary importance for the articulation of the system. However, there is a dual to this openness which is little noticed in the literature of systems theory. That dual may be characterized as a kind of closedness, or “closure.” In this case, the closure appears as the relation between the observer and the observed. Within the openness of space and time, the observer “closes in” on the observed. Normally, we think of the system as the observed which is apprehended by an ideal scientific subject which is never specified. That observer exists in a kind of “nowhere” which is the origin of a spatial perspective that is associated with a particular vanishing point. The ideal observer becomes a position which we can all assume under the disguise of our scientific persona to view the system objectively. From that position, we feel safe from involvement in the workings of the system. However, recent quantum physics, and experiments in social sciences, both lead us to the conclusion that this separation of observer and observed is a delusion.[172] In fact, there is an intimate inexplicable link between observer and observed as idealized concepts. The observation of the system changes its operations. As a corollary, we might guess that the observer is also changed by the interaction with the observed system. Thus, mutual effectivity of observer and observed is a key concept which must be taken into account by our system’s theory. The objective description of Klir, which all but forgets the action of the observer on the system, must be replaced by a system’s theory in which the closure of the observer on the observed and vice versa is recognized.

It is not surprising to find that once this closure of observer/observed is recognized, it moves right into the understanding of the system itself at a fundamental level. It constitutes another pair of perspectives on the workings of the system that now includes the observer within it. The observer becomes part of the system, and

[172] See Bib#.

the system becomes part of the observer. Their interaction form the basis for gauging all other interactions within the system, and between the system and the meta system, or its subsystems. The openness of the clearing-in-being is balanced by the closure of the observer/observed interaction which it contains and supports. At a more basic level, the openness and closure merge into the manifestation of Process Being, witnessed by Dasein as being-in-the-world. However, after the reification of separating the nihilistic opposites of subject and object, which deny all other actions other than dominance that subject the object, then the differentiation of openness and closure become inevitable. When the reified elements become active and are seen as dynamic system/observer interactions, then openness and closure assume a primary importance as the dual means of the apprehension of these interactions. The subject/object relations are purely static delineations of territory within the openness of the clearing-in-being. The observer/system interactions allow the dynamic interplay between the territorial positions. However, for the most part, the free play of exchange between positions is limited, and no territory is gained or lost. The subject/object dichotomy anchors the observer/observed relations, so there is no loss of identity for the ego in the interactions with a particular system. This kind of anchoring may be seen clearly in Klir’s general system’s theory divorced from any phenomenological sophistication. Unfortunately, it also leaves us with a system with no internal coherence that has been reduced to a mere object. The subject is safe from involvement, but at the price of not really ever understanding the system as anything other than a random collection of attributes chosen by the subject. The interplay in which the “objects themselves” suggest their own articulation and interact with the observer, is lost. The result is ignorance that masquerades as knowledge. Phenomenology suggests we listen to the objects themselves and allow them to suggest their own coherence and the ways they are best understood. Objective science explores the terrain of nature only on its own terms. However, the will-to-power that motivates their need to dominate is based on a massive insecurity. One might be led to say an ontological insecurity because it stems from the projection of the subtle clinging of “Being” upon a world which is inherently void. The observer/system interaction exhibits a limited dynamism tied directly to the subject/object underpinnings which have reified the world into a totally static configuration. It is, however, necessary to allow the free play of these interactions in order to realize fully the character of systems that have been released from the prison of objectivizing dominance by the subject. In such systems, the observer/ observed interaction becomes a fundamental relation upon which all other systemic relations are based. The closed borders of the subject and of the object are thrown open. In the interaction, observer and system merge into a new dynamic whole. The system is affected by the observer, and the observer is affected by the system. The observer becomes part of the system, and the system becomes part of the observer. In this merging, there is a closure in which these two, at times, become indistinguishable, or at least closely bound up with each other, so different aspects of the meta-system that embraces both, appear as one at one time and as the other at another time. Observer and system-as-observed form subsystems of a single meta-system which have dynamic interchanges with one another. The closure of the observer on the observed system, and the counter closure of the observed system on the observer might be called a “cloture.” The cloture is like an in-folding or mixing, which is the opposite of the unfolding of the openness within which the observer-system (ob-sys) occurs. Because the nature of cloture is closed, interlocking, and obvolute, it is not as well articulated as its dual of spacetime/ timespace. Yet it is spacetime/timespace that ultimately bind the observer and system together, if nothing else. Yet we know that “spooky action at a distance” also exists because of the proof of Bell’s theorem. Thus, the intertwining of observer/observed is inherent in the nature of things.[173] This intertwining is called “cloture.” Space-time unfolds from the singularity. In the singularity, (the always already lost origin) there is a single limit. This limit splits to give rise to the interval in which space and time are mixed. The meta views of spacetime and timespace arise as two ways to look at the interval, and from these arise the concepts of separated time and space which are then artificially merged again in our metric conceptions. This opening out of the interval from the singularity is matched by an opposite movement which is

[173] See Bib#.

here called cloture.

By Bell's theorem's proof, we now know that particles that are once related never cease to be related, no matter how far apart they become. Somehow the separation of things in timespace is inessential, whereas relatedness is essential. The openness of the interval of spacetime/ timespace is somehow superficial. This means the universe, no matter how big it expands, is somehow fundamentally interrelated because it all started by means of a single event -- the Big Being. This interrelatedness pervades everything. The singularity from which the universe arose is, in some way, identical with the whole of the universe itself. The common origin ensures a deep interrelatedness that Chung has called "interpenetration." This is the highest form of harmony. Beyond this fundamental interpenetration, due to common origin of everything in the universe, there is the further interrelation of mutual dependence which occurs whenever two things interact in the universe. The interaction of observer and system is no exception to this. Once interaction occurs, then mutual dependence follows, which ultimately is based on interpenetration. This interlocking occurs when one system impinges on another. The trace of the interaction is carried away with the two parties. No matter how far they get from each other in spacetime, there is a basic interrelation through the medium of their interpenetration which is the trace of the singularity (or their common origin within). The interlocking of observer/ observed-system is a cloture by which interaction leads to mutual dependence based on interpenetration. In the cloture, there is a relation of belonging together or “sameness” like that described by Heidegger.[174] A web is formed by everything that interacts as they leave their traces on each other. This web grows stronger with prolonged contact until the web becomes a seamless fabric of mutual interrelation. Mutual interrelation taps the source of the interpenetration that ultimately unites everything. The web of cloture is just as important as the opening in which it occurs. In fact, the web that fills the opening is in some ways more basic. Both the opening and the web arise from the singularity. The opening is an unfolding from the single limit of the singularity into the multiple limits of the intervals within spacetime/timespace. The web of cloture is the creation of mutual dependence

[174] See Bib#.

through interaction based on the traces of the singularity left on everything which appears as the interpenetration of all things. From the web of cloture arise ecological systems and cultures. Rupert Sheldrake has begun to explore phenomena related to cloture in The Presence of the Past.[175]

In fact, all meta-systems are webs of cloture. Meta-systems that do not contain singularities are merely environments. In environments, different systems interact and form mutual dependences. These mutual dependences tap the interpenetration of the subsystems to produce a deep harmony. In some cases, the meta-system will represent to itself its own always already lost origin as a singularity. When the singularity appears, the perspectives on the system fragment. This reification and reintegration of perspectives allows the singularity, or lost origin (the impossible), to haunt the meta-system without ever being observed. Cloture exists not just between observer and observed within the meta-system; cloture exists between each element of a system. Any element of a system may be considered as observer or observed by any

[175] See Bib#.

other element. The elements of systems take account of one another by mutually affecting each other in their interactions. They react to each other. It is the sum total of these natural reactions that make up a system. This is, in fact, mutual observation. Observation is not a passive perception. Observation is a reaction to the other. It is the sum of these reactions to others, and the internalization of this set of reactions, which defines the self of each entity within the system. G. H. Mead made this position clear in his analysis of society in terms of symbolic interaction.[176] Each entity is defined in terms of its multiple cloture with all other elements within a system. Multiple cloture is the key for understanding how the two other perspective (agent and function) on a system arise. Think again of the difference between essence and accident. The essence is the set of attributes and their interrelation which are necessary for anything to be "what" it is. The accidental attributes are those that can change without causing a thing to change kind. Any eventity within a system may be considered with respect to its essential or accidental characteristics. By considering its essential attributes, we assign it to a class.

[176] SeeBib#.

Normally, accidental attributes vary over time, while essential attributes tend to vary less over time. When we are attempting to pin down what remains of a thing, it is the essential attributes which are sought. The essential attributes give a thing its Being or persistence. An instance of an eventity is distinguished more by its accidental properties than by its essential properties. Accidental attributes distinguished instances of the same kind. Kindness is dependent on persistence of essential attributes.

When an eventity is thought of as an observer of all other eventities in a system, it becomes clear that the multiple cloture defines each eventity instance, giving each its unique position within the system. This diacritical definition, whereby the meaning of each element in the system flows from its relation to all other elements, obviously gives a new twist to the accidental/essential dichotomy. It is clear that any system is made up of a certain set of different kinds of eventities. These eventities have specific relations based on the interaction of their persistent attributes. But also, each eventity probably has multiple instances that all define each other diacritically by their accidental properties. Thus, diacriticality occurs on two levels. It occurs on an essential level by the relation between eventities of different classes, and on an accidental level by the accidents that define each unique instance. Essential diacriticality may be viewed as independent and orthogonal to accidental diacriticality regardless of the spacetime positions of the instances. The dynamic nature of systems are dependent on the interplay between these two forms of diacriticality. The system can be recognized as the same over time because of the essential diacriticality between kinds of eventities. In can be recognized to change because of the accidental diacriticality between different instances. Actual movements in space and time may be irrelevant here, but usually these form a third level of important change. Changes in essential diacriticality are called the evolution of the system. Changes of accidental diacriticality are called the system dynamics of the system. Change of levels of instantiation of different eventities is called population fluctuation. Changes in spacetime position of instances are called demography.

The essential diacriticality exists because within a system there are persistent and changing relations between the essential attributes of eventities. Accidental diacriticality exists because within a system there are persistent and changing relations between the accidental attributes of instances of eventities. These two types of diacriticality are different from the essential and accidental attributes of the eventities themselves. These types of diacriticality are two complementary aspects of the web of multiple cloture between all elements of the system. These two aspects are, in fact, the basis of the other two perspectives on the system called hitherto “agent” and “function.” The difference between agent and function has been characterized by the difference between what and who. Who you are is almost entirely dependent on your accidental attributes. Your date of birth, address, social security number, name are all things that are inessential, but they are the very things that identify you. These accidental attributes answer the question as to who you are. On the other hand, what you are distinguishes your kind from all other kinds. For instance, what you believe, what you do for a living, your education, your genetic make-up. These are all essential to what kind of person you are. Perhaps they were originally random as well, but if any of them were changed, you would be a different kind of person. The difference between “who” and “what” is fundamental to our way of thinking about any population. Klir mentions space, time and population as basic backdrops for attributes of systems. But population is a complex concept. Separated from concerns of demography or numerical ratios, population becomes a combination of who and what. And these separate components of population actually separate into two independent perspectives on the eventities that make up a system. This is similar to the way space and time are separated from spacetime or timespace.

With respect to systems, just reducing these perspectives to “who” and “what” related to the attributes of entities and their instances, is too simplistic. Actually, it is not the essential and accidental attributes that determine these perspectives, but essential diacriticality and accidental diacriticality. Essential diacriticality is related to system evolution, and accidental diacriticality is related to system dynamics. System evolution is related to functional process of transformation. System dynamics is related to a locus of action as an identifiable agent.

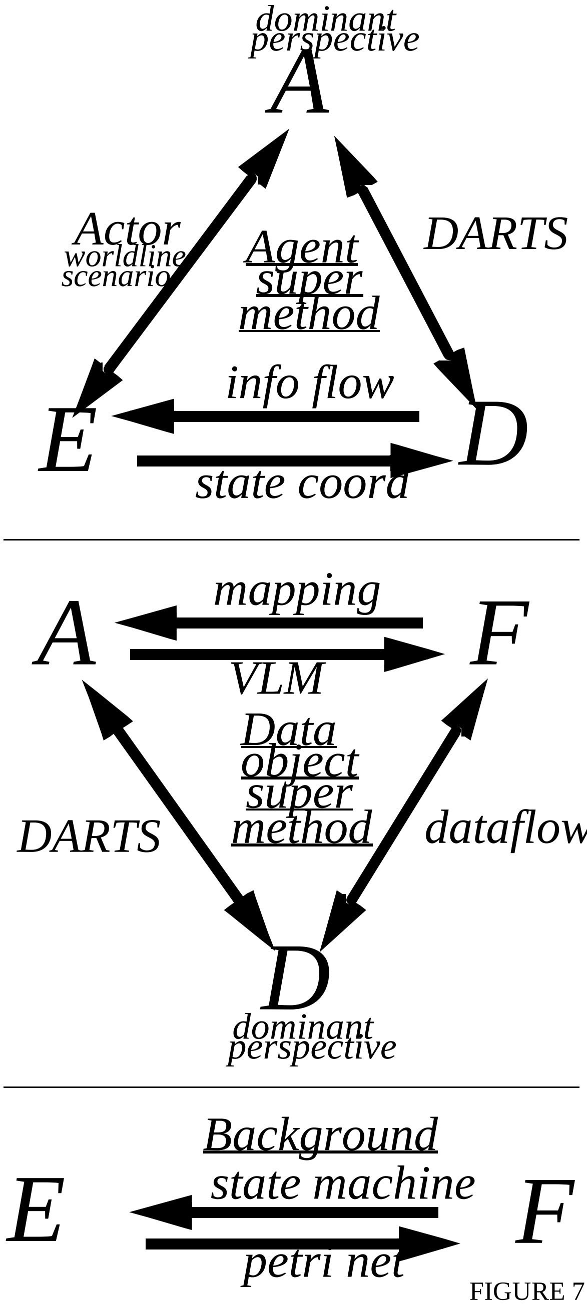


FIGURE 7 3

The diacritical essential relations in systems express themselves as transformations. These processes of transformation within the system appear as functions that take a certain input and produce a specific output. Eventities cooperate to produce these changes. When transformations affect the whole system, because functional relations change, the system is said to evolve. Essential diacritical relations are not static. If they were static, then there would be no system. Systems are normally characterized by the chains of functional relations that constitute it. These chains are called processes. A process is a specific set of transformational steps that lead to a specific end. All systems contain these chains of transformations made up of functional steps. The chain of transformations is held together by the persistent essential attributes of the different kinds of eventities that make up the system. On the other hand, each system will be made up of different vortexes of action where different kinds of eventities work together. A vortex of activity is not necessarily the same as a chain. Different vortexes of activities, called virtual processors, may perform separate functional steps in a single process, or all the functional steps of a single process may occur in a single virtual processor. Activity vortexes and chains of transformations depend on eventities and their instantiations. However, these are completely different concepts from that of the eventity. The eventity takes inputs and produces outputs. But the differences between inputs and outputs do not necessarily constitute a transformation. Nor is an eventity necessarily an activity vortex, a transformation or an activity vortex may be in the interspace between several eventities that cooperate together to perform the transformation or make up an activity vortex. In some sense, the transformation and activity vortex are absences rather than fullnesses. It is precisely because they are absences that an eventity can step into that space and fill the role. Yet the role was there even if there is no particular concrete thing that can be identified to do all the work. It is better to think of transformations and virtual processors as two types of operating areas between groups of cooperating eventities. Associating them with a positive structure is a mistake of misplaced concreteness and leads to misunderstanding the nature of systems. Systems affect chains of transformations called processes through the cooperation of different kinds of eventities working together. Systems are composed of many different virtual processors which are vortexes of activity by cooperating eventities. Virtual processors and a chain transformations may or may not overlap. The voids that are vortexes of activity, or functional processes, may or may not be filled by a single eventity affecting these changes by itself. More likely, in complex systems, vortexes do not overlap chains, and there is no functional or processing eventity. Groups of different kinds of eventities cooperate to form vortexes or chains. This is why we need the concept of system over and above the concept of object. Systems are precisely a set of cooperating eventities that form vortexes of activity that are virtual processors or chains of functional steps that form processes. In processes, things change kind. The process transforms its inputs from one kind of thing to another. In activity vortexes, things change who they are. If nothing else, who is active at a given time changes. Activity is the fluctuation in the accidental attributes of one or more instantiations. Transformation is the change in the essential attributes of one or more instantiations.

We have grave difficulty thinking about

anything that is not a concrete thing. Vortexes and chains of transforms have no substantial thing to pin our hat on. Yet systems theory that concretizes everything within a system is in grave danger of misunderstanding the whole system. We will use terms like nexus, locus, vortex and chain to describe these sets of changing diacritical relations within the system. We will attempt to avoid false concretization. Vortexes of the actions of virtual processors are seen from the agent perspective. Chains of transformations that form processes are seen from the function perspective. These two perspectives appear as two aspects of the diacriticality of multiple closure within the system. The “who” of an eventity within a system is determined by accidental diacriticality between instances of different kinds of eventity. The “what” of an eventity within a system is determined by the essential diacriticality between eventities of different kinds of eventity. But who and what need not be tied to a particular eventity. An agent may be a group of interacting eventity instances. Likewise, a chain of transformations may be affected by a group of interacting eventities. These locuses of transformation or activity can change the what and who of other eventities within the system. An eventity may either be acting, acted upon, or both, in respect to a particular locus. The locus has a function, or an identity as an agent, which is independent of whether it is associated with a single or a group of eventities.

◆ ◆ ◆

Once the agent and function perspectives have been clearly identified, it is possible to begin the derivation of the minimal methods associated with the bridge between these two perspectives. We will begin by differentiating how these perspectives view the different levels of harmony. At the horizon, the agent sees peripherals in the world which are either sensors or actuators. These are normally specialized hardware equipment that the system interfaces to in order to be connected to the world. At the horizon, the function sees the externals which are either sources or sinks for its transformation resources or products. The context for the agent is a network of processors which form a cluster interconnected by communication channels. The context for the functions are the skin of the system that forms an envelope around its chains of functions.

This is normally denoted by a context bubble in structured analysis. The facet focused on through the agent perspective is the virtual processor. This is a locus of coherent activity. The facet focused on through the function perspective is the transformation. This is a chain of functions that lead to a particular result by a set of specific steps. The attribute that the agent looks for within the processor is the task. The attribute that the function sees within the transformation is the function. The task, or the function, are the threads of coherence within their respective locuses. A virtual processor may have several threads of coherence within its activity vortex. A transformation may have several threads of coherence within a transformation. These threads of coherence are very important. They make it possible for the system to be coherent as a whole. Weaving together all these threads of coherence within a system is what differentiates a system from an object. Having the locus of activity and transformation is not enough. These are only theaters of operation in which the coherence of the system may be expressed. Good design, using the language of minimal methods and guiding heuristics, together with insight, make it possible to confer coherence on the software system. Eventities are the warp of the system, while the threads of coherence form the woof of the system.

◆ ◆ ◆

The entity-relation description of these two perspectives, as shown, is very simple, and has already been explained for the most part. A detail is that processors normally have a physical substrate in a central processing unit. However, this physical hardware substrate may be many levels removed by layers of virtual processors. Tasks time share the resources of this hardware substrate. In this essay we do not use the word “process” to signify an independent task as it is used in Unix. Tasks may or may not communicate or share memory. Tasks are independent threads of execution within a virtual processor. Tasks may contain a hierarchy of sub-tasks.

Each transformation within a system has a transformate and a transformand. The former is the operator, and the latter is the operand. Each transformation may contain independent functional threads that are interwoven functions that perform a specific step within one of these threads. They have specified inputs and outputs. Functions have implementations in standard programming languages like C or Ada. Functions are normally algorithmic. All functions or transformational chains may be expressed by the control structures of structured programming, i.e. iterator, selection and sequence. A transformation chain may contain multiple functions performed at each step. So chains may have subchains or branching chains. Functional chains are always expressed as action verbs. All functions must be executed by a processor within a task to become actualized. The activity of the action verb is performed in the vortex of activity. Outside that vortex it is static. Real transformations only occur when the vortex of activity in the virtual processor is brought together with the functions that are implemented. Implemented nonexecuting functions are static code. Implemented executed functions perform active transformations in time and space. A processor that is not executing implemented functions is idle, even if tasks are running. Pointing and grasping must work together for real work to be accomplished. The agent embodies pointing, and the function embodies grasping. The processor is pointing at the currently executing instructions of the implemented function. The transform grasps and manipulates the inputs, turning them into outputs. A processing transform does actual work with concrete results in space and time. A processor without a transform is using time but not producing any spatial changes. A transform without a processor takes up memory but does not produce any changes of inputs to outputs in time. The processing transform executes the threads of coherence in the system. These are threads of coherence with the locuses of action and the locuses of transformation.

Within the system, eventities participate in processing transforms. All four perspectives from the meta-system onto the system are brought together in this statement. Eventities combine the event and data perspectives. Processing transforms combine the agent and function perspectives. When eventities participate in the processing transform, an executing software system is the result. The system viewed statically would discover several independent elements. There is the eventity, its autonetic substrata, the processors and the transformations. Processors and transformations only touch in a single moment in time. The processor plays the music of the transformation chain of functions. As it plays the music, the keys of the autonetic state machines are touched, producing different effects. The eventities are the different instruments in the orchestra of the system.

Notice that processor and transform fall apart when not held together in time and space. On the other hand, eventities persist to hold both space and time together, whether or not the system is executing. Openness produces unity, whereas closure produces separation.

◆ ◆ ◆

Next it is necessary to explore the minimal methods that combine these perspectives and act as a bridge between them. In this case, the methods that are minimal are complex. This complexity stems from two sources:

1. Agent[177] and function are only partially ordered, and

[177] See Bib#.

2. Agent and function do not combine easily but must be forced together unlike event and data. A processing transform is an overlapping of locuses, not a thing like an eventity.

Both agent and function are described independently as nested tasks or nested functions. Other than the partial ordering of that hierarchy, they have no internal structure of their own. This structural poverty allows them to express meaning. The structural richness of data and event cause them to not be able to express meaning very well. Processors and transforms are merely diacritical markers related by partial ordering. They have no real structure of their own. They simply reflect important points in the systems structure.

These hierarchies of agents and functions must combine to produce minimal methods. When the two hierarchies are played off one another, there is enough information to produce either partial order with distance or linearity without distance.

In this context, linearity without distance can be associated with the “virtual machine instructions” minimal method. Partial ordering with distance is associated with the "mapping between functions and tasks" which is the other minimal method connecting agent and function. Virtual machine instructions are the embodiment of functionality within the task. The "distance" between the instructions has no meaning. The instructions are executed in a linear sequence by the virtual processor. The virtual machine instructions are the implementation of the function which allows it to be executed by the virtual processor. Both virtual processors and virtual machines may exist on many levels within the software system. By turning a function into a set of virtual machine instructions, the processor is then able to execute the function. In this way functions are executed one by one in a chain to complete a transformational process. The processor does its processing by executing the virtual instructions that comprise a virtual machine.

A virtual machine is a composed set of instructions. Each instruction is a lower level virtual machine. Nested virtual machines are executed by nested virtual processors. The term "virtual" here indicates that the processor is not necessarily a hardware implementation.

The processor could be emulated in software which runs on a lower level hardware or software processor. Virtual machines perform transformations. They are implementations of state machines. The instructions are fired in a certain order, given particular inputs to produce specific outputs. Part of the instructions functioning is the reset of the state of the machine for the next invocation of the macro instruction.

The relation of macro to micro instructions may be shown by a structure chart diagram, as those that Yourdon-Constantine used in structured design. The sequence of calling micro instructions is determined by the state of the macro machine and the inputs (or parameters) given to the macro machine. Virtual machines are used to implement the state machines of autonetic eventities. They are abstract implementations which make functionality processible in a way that also embodies the automata. An automata is a mapping from inputs to outputs and states. This mapping may be a table lookup, or more commonly it is an implementation of some functionality which makes the transformation from inputs to outputs. In this latter case, outputs are not already available but must be made. Making outputs from inputs and available data is a transformation performed by the virtual machine. That virtual machine also resets its own state to be ready for the next invocation. By resetting its own state, it is able to participate in the external functioning of the system which expects different responses in different circumstances. When a virtual instruction has its own state, it is a virtual machine. It is multi-functional. On the other hand, an instruction may not have its own internal state, in which case it is just a straight implementation of a function. Functions transform inputs into outputs regardless of external changes in the system as a whole. They always do the same job. Many times virtual instructions are merely functional implementations.

When virtual instructions are associated with particular eventities, they are called operations or “methods.” In this case, the operations change the values of attributes or slots within the eventity. In object-oriented design, “methods” are the only means of changing data encapsulated by the eventity. We will make the distinction here between operations and workers. An operation (or “method”) is a virtual instruction that changes an internal attribute of an eventity. A worker is invoked by the state machine of the eventity to transform input to output and change the state of the eventity. Operations are visible outside the eventity, while workers remain hidden. In constructing virtual layered machines, the lowest level instructions ideally all end in operations (or “methods”) on eventities. The word “method” for operations in persistent data will be avoided.

However, some virtual instructions are pure transformations of inputs to outputs with no persistent data being changed. These pure functions, which exist to some extent, cannot be associated with a persistent object. These tend to be garbage leftover when a purely object-oriented design method is used. This problem with purely object-oriented methodology has been identified by Vaclav Rajlich[178] among others. The problem is solved by using the Object-Oriented Design/ Virtual Layered Machine (OOD/VLM) method proposed by Nielson and Shumate in the book Designing Large Real Time Systems with

[178] See Bib#.

Ada.[179] This book, and their article with the same name, should serve as a major source on this methodology. The methodology gives concrete implementation to functional transformations for a particular level of virtual processor. Functional transformations are accomplished by the processor executing the virtual instruction. Virtual instructions may or may not be virtual machines with their own state values. A virtual instruction with no state values is a simple transformation. A virtual instruction that is also a lower level virtual machine is multi-functional, reacting differently to external system states at different times. The virtual processor and the virtual machine intersect at the executed instruction.

◆ ◆ ◆

The other minimal method that serves as a bridge between agent and function is the mapping between tasks containing virtual machines and functionality. This mapping allows us to see how the total system functionality is to be carried out by the threads of coherence (tasks) executing the virtual instructions of virtual machines. This

[179] See Bib#.

traceability, from the functional decomposition of the system to the specific elements that implement that functionality, is very important. In order for this traceability to be carried out, two other methods must exist. These methods map both function and agent perspectives onto the data perspective. The method connecting agent and data perspectives was defined by Gomaa, and is called DARTS.[180] A new version developed in conjunction with the Software Productivity Consortium is called ADARTS[181] and uses Ada. The method connecting function to data is the Data Flow Diagram used in structured analysis developed by Yourdon and DeMarco. These two methods are each composed of minimal methods which combine to produce a single macro method or two-way bridge between perspectives. There is a high degree of parallelism between these two macro methods. This parallelism allows isomorphic mappings between tasks and functional bubbles. That mapping illustrates how the macro functionality of the software system is accomplished by the threads of coherence (tasks) in the virtual processor. Without this kind of mapping, there would be

[180] See Bib#.
[181] See Bib#.

no way of seeing if the entire system would function properly. This is different from the perspective that looks at the individual virtual machine instruction within a particular task to see what micro function it embodies. Functionality is both system-wide and narrowed to the particular instruction executed now. Both types of functionality must be coordinated for the software system to work properly. The two minimal methods that connect agent and function assure this coordination by giving a means of representing it.

The mapping minimal method is not as complex as the virtual machine minimal method. The mapping is a web of traceability links between functions and tasks. The method is explained best in the third volume of the Ward-Mellor series called Structured Development for Real Time Systems.[182] The process of creating the mapping is called "functional allocation." In functional allocation one first allocates functionality to processors, then to tasks and finally to sequential modules (virtual machines) within tasks. At each step of this crucial system

[182] See Bib#.

architectural design process, the essential model of system functionality as a data flow diagram should be modified to reflect implementation decisions. Functional bubbles may be split between processors or tasks, so new bubbles and associated data flows must be added. The resultant transformed essential model is called the implementation model. This implementation model is isomorphic to the processor-task-module implementation decomposition of the system into process-able units. Functional allocation is the process by which this mapping is produced. Functional allocation may be described as the transformation of the functional description of the system into an implementation description which results in traceable links. This transformation is evolved through a mental processing of the functional essential model. That mental processing results in an implementation model and a system architecture in terms of processor-task-module descriptions. It is here that the main effort of design work is done by mental simulation. Mental simulation in design is a virtually ignored aspect of the design process. There are almost no books or articles on how to perform mental simulation in design work. Yet this is exactly the most difficult part of the design process. Design simulations are necessary during functional allocation in order to make the system work properly. Without mental simulation during design, there can be no correct processing of a system at execution time. Without methods, these mental simulations are too complex for the mind to handle. Methods are the language in which mental simulations are executed by the mind of the designer. But the actual work of how to do mental simulations is not described anywhere. As a result, software design is a very difficult type of mental work. Software design is mentally exhausting. Each set of virtual instructions must be simulated mentally many times before that is ever executed by a virtual processor. In fact, the mind is the first virtual processor for all virtual machines. Mentally simulating virtual machines is different from ordinary thinking. There is a rigor to mental simulation that is not necessary in thinking, which goes against the grain of the mind and is particularly difficult to emulate. During this process, the designer is not just emulating the paths through the virtual machine under various conditions; he is, at the same time, changing the virtual machine that is being emulated so it will function correctly. This is a very complex task that is very difficult to perform, particularly because only a small part of the virtual machine (seven to nine chunks) can be kept in short-term memory at a single time. One is constantly losing one’s place, or realizing the relevance of some other aspect of the system to the problem at hand. Thus, mental simulations tend to be diverted and interrupted in mid-stream as long-term memory suggests other relevant connections within the system being designed. How to do mental simulation needs to be a high priority research area. Tools and meta-methods need to be developed to support this work and take the drudgery out of it. The definition of a framework of methods, such as that developed here, is a good start. But the methods and heuristics do not touch the heart of the problem of system design, which is the mechanics of mental simulation.

◆ ◆ ◆

Having said that the virtual processor and the chain of transformations are locuses, it is still possible to substitute the eventity into these locuses. When the eventity is substituted into the place of the virtual processor, it becomes an "actor" after the meaning of Agha and Hewett. Here the eventity becomes an agent and is identified with either the processor or the task thread of coherence. Likewise, the eventity may be substituted into the chain of functional transformation. In that case, operations are associated with the eventity that perform these functional transformations which may be independent of the modification of persistent data. The addition of these operations allow the eventity to be identified with either the virtual instruction or with persistent data objects.

By making these additions, the eventity comes to serve as a generalized computational emulator. It is a generic design element which can perform many different roles within the system. Yet this connection with the virtual processor and transform should not lead to the fallacy of misplaced concreteness. Processors and transforms are independent of their linking to an individual eventity. They are more likely to be represented as cooperating groups of eventities. However, the ability to represent processors and transforms as eventities does serve the purpose of giving us a single multifaceted design element that can be the basis of system design. The eventity is fairly complex. But this complexity allows faithful representation of the “systemic” nature software aggregates. The problem with using less complex design elements such as queues, pointers, counters, timers, etc., is that the systemic character of the mechanism they embody is hidden. In order for these systemic features to be presented in the design, the design element must represent the complexity of the spacetime backdrops as well as being an automation. In this case, the automation is dual in order to represent outward and inward control over parts. Thus, I argue that the eventity is just complex enough to represent the systemic features of software designs. And this is very important. Unless the systemic features are represented, then there is no way to hone in on a harmonic design. Harmonic design depends on the representation of the systemic figure/ground dynamics. Different eventities may serve as both elements of the figure or the ground within the system. Eventities give a generic design abstraction which is at the right level of complexity to capture the whole gambit of system functioning. Below the threshold of the eventity, which is robust enough to represent a subsystem, the systemic interdependencies are submerged within the panoply of lower level design elements. It is as if the system were out of focus. When the granularity of the design elements is wrong, the system is blurred. Eventities are at the right level of design element complexity to bring the system into sharp focus and thus ease the burden of design work.

◆ ◆ ◆

Now it is necessary to consider the relation of the minimal methods associated with the agent and function perspectives, i.e. mapping and virtual layered machine, to the metrics of system supports. As has been mentioned, both agent and function perspectives are radically different from the fully ordered space/time perspectives. The agent/function perspectives are only partially ordered. This means that within either of these perspectives that exhibit cloture, only partial order relations between elements may be adduced. Consider the agent perspective. An agent is an independent actor or center of activity. Our natural models for independent agents are organisms. For independent centers of activity, or actors, the concept of linearity and distance have no meaning. By their nature, actors are processing simultaneously independent of each other. They only take turns in a linear fashion if they are forced to through some synchronization mechanism. The distance between processors is also irrelevant to their processing capability, although it may be important with respect to their ability to communicate as a network. Now consider the function perspective. A functional transformation also lacks comprehension of distance and linearity. Like processors, transforms operate in space-time. But functionality, when abstracted from the space-time milieu, has no inherent linearity or distance component. Consider the data flow diagram. The distance between bubbles is irrelevant. The bubbles fire when all their inputs are available. Thus each bubble is really like an independent processor. The network of bubbles is ordered by the data flow arcs. But data flow arcs are a spatial feature flowing from the data perspective. A data flow diagram with no arcs would be static. Without inputs, there would be no firing. Without arcs the dataflow nested bubbles are merely a partial ordering of system functionality. There is no linearity because there are no data flow arcs, and the distance between bubbles is irrelevant.

This should suffice to show that linearity and distance do not have meaning for the pure agent and function perspectives. Yet obviously for the minimal methods of mapping and virtual machines, these concepts do have meaning. Mapping connects the hierarchy of agents with the hierarchy of processes. Through this connection between the two hierarchies, the notion of distance becomes meaningful. One can think of distance as the number of boundaries crossed in the background hierarchy. One hierarchy is used as foreground, usually function, and the other forms the background, usually agent. The background forms a grid against which foreground objects can be measured. For instance, a single function bubble might be split between two agents. The separation of the functionality by the agent demarcations induces distance. This is why we call this assignment of functionality to multiple agents the “spreading” of the functionality out within the system. In this case, two partial orders, when connected by mappings, gives rise to meaningful demarcations of distance. Distance is measured by counting the boundaries crossed in the background grid by objects in the foreground grid. On the other hand, virtual machines exhibit linearity. The instructions of a virtual machine are fired in sequence. The idea of distance between instructions, however, has no meaning. Each instruction is an implementation of some functionality that can be executed by a virtual processor. The instruction is the intersection of the functional map of the system with the processor map. These single points of intersection are active, one at a time in sequence and their firing constitutes the execution of the software program. Here again, it is the relation between agent and function hierarchies at the single point of execution linearity through time that allows two partial orders to emulate a linear order without distance.

As a general principle, agent and function are inherently partial orders when viewed separately. When used together, they can emulate linearity without distance (virtual machine minimal method) or partial order with distance (mapping of functionality to agent minimal method). However, agent and function cannot emulate full ordering in the way event and data perspectives can. This means that there is no equivalent of cartesian coordinates for agent and function. In cartesian coordinates, both linearity and distance may be seen at once. Event and data may be represented within cartesian coordinates or via a timeline using real numbers on the x, y, z and t axes. There is no equivalent to this for agent and function perspectives. Processes and agents may be seen as moving in time and space, but they do not themselves give rise to fully ordered supports against which system variables might be measured. Agent and function only give rise, at most, to supports with linearity but without distance, or supports with partial order with distance. This seeming defect in the function and agent perspectives is a major barrier for software system modeling. In information systems, the barrier is not readily apparent. In these systems, the space-time constraints are relaxed. Thus, all the minimal methods suffice to work together to model the system. However, in real-time reactive systems, the event and data perspectives must interface with events in the world and move information through the network to meet the system needs. In that case, a gap appears between the fully ordered capabilities of the event and data perspectives, and the lack of that capability in the agent and function perspectives. This gap is the hiding place of the singularity of pure immanence. It is the unbridgeable gap between openness and cloture. It is what makes the design of real-time reactive systems so difficult. This is because ultimately the methods do not help at this extreme point. Ultimately, the methods break down, and there is no full coordination between system supports. There is only a partial coordination. The system must be designed according to the minimal methods as they overlap when space-time constraints are relaxed. Then the space-time constraints must be tightened to discover performance as the system is executed. The performance where the system is executed must be analyzed, and then the design is modified to attempt to solve performance problems. The gap between methods design and system execution performance is a fundamental element in the software system. It is this gap which gives software science its important role in observing executing systems. It is this gap that gives software its quasi-experimental nature. It is this gap which is the point of cancellation of Hyper-Being.

In fact, the gap is not a defect. It is an important indicator of the nature of software. Agent and function embody meaning in a way event and data do not. The poverty that agent and function have from the point of view of systems supports is made up for by their ability to serve as markers for meaning. Data and event have no internal dimension. They represent complete exteriority. Function and agent, on the other hand, have a dimension of interiority in which semantic value may be invested. The expressiveness of agent and function, in terms of semantic content, balances their poverty with respect to metric ordering as system support variables. This fact shows that software methods can be vehicles for semantic expression. They have a natural interface with language that appears whenever we say “what” or “who.” We anthropomorphize agents and project organism (or even human) qualities upon them. This is not correct, but it shows the viability of the semantic interface of these methods with language. This interface is even more pronounced with respect to saying what a function is. We speak of functionality in the same way we confer meaning on things in everyday life. By this means, the functioning system becomes a particular kind of thing --

i.e., it comes to inhabit our world along with us. The bank teller software with its hardware does the job of the human teller. It becomes an "automatic teller." The emulation becomes the thing it is emulating. Our ability to say what a software system is, and thus incorporate it into our world flows from the interface of the function perspective with language. The function elements in the system serve as bearers of meaning, as do the agent elements. This connection between software systems and linguistic reality is very important. The fact that software methods have this interface shows that their structure in relation to the gap between openness and cloture is no accident. In this we see an essential extension of technology into meta-technology.

The derivation of the minimal methods that directly relate agent and function perspectives, or event and data perspectives, has led to an appreciation of the difference between these two sets of perspectives. They can be contrasted in terms of the ordering metrics associated with them. Event/data perspectives are fully ordered, whereas agent/function perspectives are only partially ordered. We see clearly how the minimal methods derive from relaxation of ordering in the case of event/data and a tightening of constraint in terms of agent and function. The tightening of constraint occurs when partial orders are combined to attempt to mimic fuller ordering. There is, however, a limit to this mimicking, and full ordering is unobtainable. This impossibility that lurks at the heart of the matrix of software perspectives is an important phenomenon that determines the nature of software. It bears out the ontological analysis already performed in the earlier part of this essay. The gap between event/data and agent/function is the fundamental locus of differing/deferring within which the singularity of pure immanence resides. The gap itself is like the event horizon of the singularity. It only shows up through a detailed rigorous analysis which seeks to understand fully the interrelation of minimal methods that spring from the perspectives on software design.

The four minimal methods explored thus far are crucial to understanding the nature of the gap between openness and cloture. What has been learned by this analysis and attempted derivation may now be applied to the other minimal methods. There are five other methods that relate the perspectives (agent, function, event and data):

EA	Worldline/scenario	Two-way bridge
PD	Data flow	Two-way bridge
AD	Tasking and comm mech (DARTS)	Two-way bridge
PE	State machine	One-way bridge
EP	Petri net	One-way bridge

Interestingly enough, three of the remaining methods are two-way bridges composed of two complementary minimal methods which fuse into a single macro method. Of these three macro methods, two are very similar. Those are the data flow method of Yourdon and DeMarco, and the DARTS method of Gomaa. In these methods, agent and function are almost interchangeable. There is a direct parallel between all the features in these two macro methods. This shows that the data perspective exerts a powerful organizing force which sees agent and function in very similar ways. Attempts have been made to synthesize these two macro methods into a single overarching representation. For instance, Mitchell D. Lubers of MCC developed "A General Design Representation"[183] which attempts to combine the data flow and tasking/communication mechanisms into a single super method based on a Unix processes and pipes motif. This kind of collapse of two macro methods into a single super method has the danger of confusing the perspectives which are intrinsically distinct. From the separation of data flow and DARTS, is opened out the mapping minimal method and the virtual layered machine method. In fact, it is possible to see that the viewpoints from which two macro methods diverge, i.e., the data and the agent perspectives, have the greatest organizing capacity with respect to the software design. Within each pair of viewpoints separated by the gap, there is a dominant viewpoint (agent dominates function) (data dominates event). The dominant viewpoint has diverging macro methods (two-way bridges) which opens up the arena in

[183] See Bib#.

which the methods connecting the opposite set of viewpoints exist.

The inferior perspectives from each pair (event and function) are connected by two one-way minimal methods which are state machines and petri nets. These minimal methods are important, but do not have the overall organizing capacity that the cones of methods arising from the dominant perspectives have.

DATA FLOW AND DARTS ORGANIZE MAPPING AND VLM

AGENT WORLDLINE AND DARTS ORGANIZE STATE COORDINATION AND INFORMATION FLOW

From this, we can see why the object-oriented design is beginning to achieve dominance over the functional orientation. The methodological field itself has an internal structure that would shift the balance in the favor of data-oriented minimal methods. We would expect the same sort of ultimate emphasis on “actor” like systems such as that described by Agha. The function/event axis would apparently have to be subservient to the two major triangles of super method groupings.

This does not mean that any of these methods should be neglected in system design. It means that there are intrinsic organizing principles in the field of the methods which dictate the perspectives from which the most powerful organizing forces will naturally radiate.

◆ ◆ ◆

Partial order		**Linear with**
with distance		no distance
	Actor	
Scenario		worldline
	DARTS	
Monitors and tasks		Tasks and comm mechanisms
	Data flow	
Methods and data stores		Data flow arrows
	Background	
State machine		Petri net

Each of the eight minimal methods shown above can be divided into those which approximate linearity without distance metric, and those which approximate partial ordering with distance metric. The concept is that all minimal methods approximate one or the other of these two intermediary metrics. All of these minimal methods are combinations of the fully ordered and the incompletely ordered perspectives. Each of these methods operate across the gap between perspectives with similar metric orders and thus combine, in various ways, dissimilar ordering elements. However, in this case, the approximation to the intermediary metrics comes from the collapsing (relaxation) of the event perspective toward linearity, and the collapsing (relaxation) of the data perspective toward distance. The interaction of a partially ordered perspective with a fully ordered perspective that is related causes the fully ordered perspective to dominate. Thus, worldline and scenario have a dominant linearity, as does state machines and petri nets. Whereas in DARTS and data flow, their submethods have a dominant distance component. For this reason, the direct derivation of the two intermediate orders is not straightforward as it was in cases previously studied. Yet for the most part, the paradigm of which sees each minimal method as an incarnation of an intermediary level metric construct (L-D or PO + D) can be applied directly.

For example, the “actor” macro method which combines worldline and scenario minimal methods, has a mapping to the two intermediary level metrics.

Scenario	Worldline
Partial Order + Distance	Linearity-Distance

Linearity is dominant because the event perspective’s full ordering collapses into L-D. Thus both of these minimal methods are apparently linear. The worldline may be seen as a fully ordered timeline, or as a sequence of events for a single agent. Scenarios cut across multiple timelines with a linear ordering of events performed by multiple agents. However, because worldlines may be relativistically related in the absence of a global clock, this linearity of scenarios is not what it may seem at first glance. The distancing of agents in different inertial frames (associated with nonsynchronous clocks) plays a role in scenarios not present in worldlines. Worldlines normally collapse from full-time ordering using the real number timeline to a linear sequencing of events and actions. Scenarios involve the slicing across worldlines in non-orthogonal planes that allows causality between events and actions to be seen. When scenarios become involved with relativistically related worldlines -- usually this is not obvious -- then there is interval distancing between events and actions. The “actor” model of concurrent processes is designed to handle this kind of interval distancing which allow nonsynchronous clocks for multiple worldlines. Here it is clear that the dominance of linearity contributed by the event perspective to these two minimal methods is not straightforward. Distancing insinuates itself within the scenario minimal method in ways that may not be obvious. Thus, though the analysis is clouded, it is not impossible. Ultimately, it is possible to understand how the intermediate metrics show up within the two minimal methods being studied.

Another case is the DARTS and dataflow macro methods. These are very similar. In both, distance is emphasized through the relaxation of the space (or data) perspective. For this reason, it is clear that the data flow bubbles or tasks boxes strung together by data flow or communication mechanism arcs is a linearization metric. Whereas the relation of function bubbles or task boxes to data stores or monitors is primarily a distancing metric. The linear sequence of bubbles strung together by arcs is contrasted with the mapping of methods to data objects. The linear sequence of tasks strung together by communication mechanisms (queues mailboxes, etc.) is contrasted with the mapping of tasks to monitors. In these cases, linearity and distancing appear clearly in the different related minimal methods. Distance appears in the mapping from active to passive elements. Multiple active objects are related to a single passive object. In object-oriented design, this mapping is made such that there is a close binding between the method and the data object. In functional design, the mapping is less direct, so that there is greater distance between the function and the data object. With respect to tasks and monitors, the distance appears as contention of tasks over the same resource.

The final example is the state machine and petri net minimal methods. Petri nets are like refined control flow charts which are augmented with states and control flow tokens. The petri nets emphasize the linear flow of control; whereas the state machines emphasize the relation of state places to each other. Distance appears as the number of intermediary states exist between any two states. Linearity appears as the sequential path as the control token. Petri nets and state machines are intimately related. The emphasis on state diagrams is the static representation of the state places and their interrelationship. The emphasis on petri nets is the linear sequence of states that a control token moves through as the transitions fire. The former is static, while the latter is dynamic. The former emphasizes distance, while the latter emphasizes linearity. It is interesting that it is the state machine which must be added to the eventity to make it essentially autonetic. One state machine governs the coordination of parts while the other governs externally observable behavior. The fact that the state machine/petrinet minimal methods are left over as outside the agent and data super method complexes is very significant. It shows that the state machine/ petrinet methods have a special place in the field of minimal methods. This special place has been characterized as the background because they are not included in the super method complexes. But they could just as well be characterized as the jewel that stands out on the background of the super method complexes. It is state machine and petrinet methods which are used to model the structure of automata. Automata are either active or passive, depending on whether they are parsing input or defining output. When the active and passive state machines are combined in a single two-part complex, they become autonetic or self governing. The self-governing autonetic automata can be described in terms of state machines or petri nets, depending on the static or dynamic emphasis. When the state machine/ petrinet is combined with the eventity, then the autonetic eventity is the result. The autonetic eventity is the key to advanced design that takes into account all four perspectives on software design. It combines all four perspectives into a single multi-purpose design element. This integration of design perspectives by advanced knowledge of methods is the crucial work of software meta-methodology.

◆ ◆ ◆

This essay began as an exercise in meta-methodology. Meta-methodology seeks to understand the interrelationships between methods. Software design methods derive from the four fundamental perspectives on the software system. These perspectives (agent, function, data and event) have been identified with what Klir calls “backdrops” or “support variables.” The minimal methods define the relationships between these supports. The support relationships form a substructure that serves as the foundation for the superstructure of the designed software system. Normally, the substructure is invisible because its main purpose is to provide something for designed system variables to be measured against. In this essay, an attempt to make the substructure visible has led to an appreciation of the complexity of the field of intra-method relationships. This substructural field is the backdrop against which all the relations between designed system variables must be understood. If a variable can only be measured against one perspectival support variable, then only the method associated with that perspective may be used to define that designed system variables in relation to other designed system variables. If a variable can be measured against two perspectival support variables, then the relevant minimal method may be brought to bear to define the place of that designed system variable within the whole designed system. The more perspectival supports that the designed system variable can be measured against, the more central that variable is to the internal structure of the designed system. The key system variables can be measured against all the perspectival support variables. As the design progresses, more and more of the designed system are brought within view of all the perspectives. In a fully defined system, a good portion of the superstructural variables should be, in principle, situated in relation to the total field of the substructure of support variables and their bridging methods.

The development of design perspectives as support variables allows us to plug our design methods that flow from these perspectives into Klir's overall definition of the epistemological levels of the formal-structural system. It is, however, necessary to read Klir's overall definition of the formal-structural system in a particular way in order to see its significance for software design. Klir's general systems theory should be considered the basis of software engineering formalism. Having defined the generic backdrops or perspectives applicable to all software systems, Klir’s general system theory then becomes a general theory of software systems. We need only look at it in terms of systems design instead of systems inquiry. This may be accomplished by realizing that every designed system must interface directly with its environmental anti-system. The designed system and the anti-system form two subsystems of a meta-system. In the design process, technical source systems for both the designed and the anti-systems must be defined. The external system boundary separates the designed source system from the anti-source system. Both the designed and the anti-source system variables are measured against the fundamental perspectival support variables. These fundamental software design perspectives mediate the interchange between the two source systems. Once the source systems have been defined, then the data flows across the external system boundary are isolated. In relation to these data flows, the anti-data system is defined. Its reciprocal designed data system is postulated on the basis of the needs of the anti-data system. The designed data system is the inverse of the anti-data system. The next step is to define the designed generative behavior system. This is the software capable of generating the required data streams. This generative behavior system may require further development of structural and meta-models which drive the generative behavior (software) system. Support invariant constraints between variables necessarily change overtime. These changes from one model that generates data to another are structural relations. Meta-models record the changes in structural relations. Each level above the generative behavior system records the embedding of more and more sophisticated constraints into the software design. Structural systems record changes in design system variables relations. Meta-systems record changes in the support variable set in relation to the designed system variables. In the process of design, variables measured against a single perspective are extended to be measured against two perspectives, then three perspectives and then four perspectives. Each extension of a designed system variable to another support perspective allows another meta-system level to be defined. What is invariant in relation to two perspectives may be variable in relation to a third perspective. The addition of support variables other than those required by the software design perspectives, allows this process of generating meta-systems to become an endless regression like that of the endless levels of meta-structures. The two infinite regresses of meta-structures and meta-systems approach infinity, ending in the singularity of pure immanence.

Klir's definition of the formal-structural system gives a firm basis for understanding the embedding of structure into the software design. Formal methods in software engineering normally apply logic and other discrete mathematical formalisms to the definition of software. These formalisms must be augmented by an appreciation of meta-structure and meta-system articulation. Software design must be defined by structural as well as formal relations. The formal relations are the basis for understanding these structural relations. Klir's model of the formal-structural system is the only known model where these structural relations are clearly defined. Thus, formalisms such as logic, set theory and axiomatic definitions should be applied within the framework of general systems theory to achieve a proper formalization of software design. These formalisms are independent of the software design methods.

The field of software engineering must necessarily draw from several sources which work together to get full coverage of the problem areas being addressed. The unifying discipline is mathematical category theory which is applied to abstract automata. To this is added Klir's definition of a formal-structural system with its various epistemological levels. The formal components of the formal-structural system may be controlled using various other mathematico-logical formalisms such as logic or set theory. For software systems, the four fundamental perspectives must be added, along with the bridging minimal methods. These perspectives define the substructure upon which the designed system superstructure is built. To these sources must be added domain analysis[184] which adapts the software design methods to a particular domain. Domain analysis is explained by S. Shaler and S. J. Mellor's article "Recursive Design."[185] In this article, they identify four types of domain:

[184] See Bib#.

[185] See Bib#.

1) Application Domain Specialized Architectures

2) Service Domain Database, Operating System, etc.

3) Architecture Domain: Agent, Event, Function, Data

4) Implementation Domain: How architectures are incarnated as programs.

The design methods articulated in this essay describe in detail the architecture domain. Implementation domain includes languages, operating systems, data bases and other tools at various levels of abstraction which allow the who, what, when and where of the design to be expressed as “how.” This is implemented concretely so that it runs as an executing software system. Service domains are various specialized architectures which serve the application which, in turn, has its own architecture. Each of these domains consist of objects that have data and states as well as sets of operations that modify the data and states. Shaler and Mellor suggest using their object-oriented analysis method to delineate these domains. Within the architecture domain, however, the concept of autonetic eventities is a more powerful organizing construct. Within each domain, various mathematico-logical formalisms may be applied to organize and keep consistent the design information. But these formalisms must remain subservient to the general systems theory concepts which are applied to each domain in order to discern the systems that give the objects within a domain coherence. Object-oriented analysis lacks these basic systems' theoretical constructs. Architectural design methods must eventually be applied to the various domains in order to render the software design which may then be implemented. The steps of meta-design might appear as follows:

1. Identify domains and their relations

2. Identify objects in domains and their operations

3. Apply formalisms to all domain objects and operations

4. Apply design methods to formally constrained objects

5. Embed meta-structures and meta-models in domains

6. Implement designed system

7. Study implemented design behavior.

Meta-design takes into account the place of the design methods in the world as they are applied to different applications. Design methods, by themselves, are merely an abstract view of how all embedded software must be structured. When combined with domain analysis, formal mathematico-logical methods, and Klir's definition of the formal structural system, these formal methods find their application within the world.

◆ ◆ ◆

This essay must finally turn to considering the significance of the substructure of software methods in relation to the superstructure of the designed system. Meta-design posits that these software architectural methods must be understood in relation to other disciplines. But fixing the place of the field of software methods among other disciplines is not enough.

We strive here to understand the meta-technical in some fundamental way. It is clear that the meta-technical is the substructure added as a supplement to the technical formal-structural system as defined by Klir. This supplement dovetails with Klir’s definition by fitting into the slot of “backdrops” and “support variables” that ground all measurements within the software source system. Yet, the supplement adds unwanted elements. The perspectives do not all have the same metrical format. The division between fully ordered and partially ordered metrics causes the field of software methods to be distorted. This distortion, when surveyed carefully, reveals a serious gap between the two kinds of metrics. It has been posited that this gap hides the singularity of pure immanence associated with Being^3. Software derives its inherent nature from this kind of Being^3. Software is a strange kind of writing. It is automated writing which shares weird characteristics with the automatic writing of the explorers of the paranormal. In automatic writing, some other force from another realm is said to control what is written. The Ouija board is often used as a tool in the seance. The fingers are moved by spirits from another world to spell fateful words in answer to questions. This phenomenon is sometimes explained as unconscious direction of the motor system. It is usually dismissed as self delusion by those who are too easily impressed by tales of the supernatural. Whatever the explanation, automatic writing is a strange, borderline phenomenon set outside the boundaries of the domain of order and reason. In automatic writing, the question is “who” or “what” is controlling the writing. Some other agency other than the subject is posited which is speaking to us -- from the unconscious; -- from the spirit world; -- from our need for self delusion.

Within the realm of technology, well within the domain of order and reason, safe from all accusations of pseudo scientific tendencies, there is another kind of automatic writing. In this case, it is the written text of software executed by the central processing unit of a computer. At first glance an altogether different situation. Yet, on further analysis and reflection, software becomes more and more strange. Slowly it is realized that software is not like other mundane things. It has a special kind of presence in the world. That kind of presence has been named Hyper Being. It is the cancellation of Process Being of Heidegger with the Nothingness of Sartre. It indicates the existence of pure immanence, unseeable, behind the showing and hiding of temporalized, i.e. pure transcendence, presence. By analogy with the physics of Blackholes; pure immanence is like the singularity surrounded by an event horizon, i.e., outside the laws of physics and things beyond reasons and normal order. In physics strange things occur at the hypothesized event-horizon. What occurs at the event-horizon where Process Being and Nothingness cancel, becoming crossed out, is described by Derrida as differing/deferring of Differance:

> Derridian difference is movement which permits the spacing required by difference while necessarily deferring fixed relations between signifier and signified.[186]

This event-horizon has been identified with the gap between fully ordered and partially ordered perspectives on software design. The gap between these two pairs of perspectives is a fundamental difference. This difference is opened up by the more fundamental differences between pure immanence and pure

[186] William Corlett, Community Without Unity, Pages 157-158

transcendence, and between the two cancelling forms of pure transcendence: Process Being and Nothingness. The dynamic of the presence/non-presence of Process Being, set against the dynamic of presence/absence of Nothingness, gives rise to the phenomenon of cancellation. In cancellation, the present becomes a nonpresent absence. What was present just vanishes through annihilation. Not even a trace is left. This means pure transcendence in its two poles, turns into pure immanence -- that which can never be made present and which is inherently nonpresent and absent. That singularity of pure immanence is then in one sense identical with pure transcendence. In fact, the “purity” of immanence and transcendence slowly becomes suspect. Perhaps these also are nihilistic antimonies of pure reason which also ultimately collapse in the cancellation. What is left after the dust has settled from this cancellation of nihilistic meta-physical opposites is the world -- not even a leaf has rustled as a result of these momentus metaphysical events -- which lies undisturbed before us, wild and primordial, which existed before the edifice of ideation was erected. Differ*a*nce is the opening up of the fundamental metaphysical differences which creates an opening which all other differences can operate. This opening up of the difference between pure immanence and pure transcendence, which in turn flowers into the difference between Process Being and Nothingness, creates the external/internal horizons of subjectivity. When the subject looks out, it sees the infinite horizon of Process Being as temporalized manifestation of things in the clearing-of-being. When the subject looks inward, it sees the infinite horizon of Nothingness where all its searching for an inward core gets lost in an endless desert of nowhere. The subject gazes on the world from this nowhere. Corlett defines the subject as . . .

> any being capable of maintaining continuity across time. Accordingly, a (reasonably collected) person can be a subject, but so can a neighborhood or village.[187]

The subject has been defined by Corlett in a way which separates this concept from the individual, but what is deeply interesting, if not thought provoking (in a Heideggerian sense), is that agency has been projected by computer technology outside the human organism.

[187] Page 22, Community Without Unity.

Computers driven by software can be seen to fall under this definition of subjectivity. They maintain continuity as agents across time/space as CPU cycles and digital memory. Thus, the human organism becomes replaceable by the automation as a locus of experience in terms of a space/time nexus. This new subject does not just maintain continuity passively, but acts as a transformative agent as well. In some way, it is possible to see that the definition of the four fundamental perspectives is some minimal definition of subjectivity, and that the “gap” of the event-horizon of cancellation opens up directly in the middle of this minimal definition of the subject. Whether the subject is a single processor or a cluster (neighborhood or village), is not relevant. Fragmentation of agency is not as important as maintaining the illusion of continuity. The illusion of continuity is precisely what the notion of the subject guards. This illusion is generated by the mechanism of ideation. Ideation repeats the image fast enough so that the gaps in the projection cannot be seen. The differences between repetitions are glossed so that differences within the projected image can catch the eye. Within the human organism pretending to be “subjugated” these images are his simple ideas. Within the automata, these images are the primitive operations of software commands.

Human beings have projected their carefully designed subjectivity out beyond themselves into the world. The ideational mechanism has taken on a life of its own in a nonorganic realm. But the crucial question becomes "who is the designer." We fashion our robots as we attempt to refashion ourselves through genetic meddling. Strangely, we discover that software is the key in both cases. The genetic software of DNA allows us to reprogram ourselves, while the software that controls the automata allows us to mimic ourselves in silicon representations. The "designer" in both cases is not easily pinned down. The organic foundations are changing, while the ideational locus of subjectivity is projected outward. Suddenly we wonder who is writing the software. It is certainly not the subject, because that is what has been thrown out into the world as automata. It is not the human organism, because that is changing under our feet. When we search for the "designer," we look into ourselves and see only nothingness. We don't know who we are. We discover only viewpoints from nowhere. When we look outward, we see an infinite horizon of temporality full of beings we project as we fall towards death. Some of those beings are automatons animated by ourselves. Some of those beings are organisms we alter genetically. Suddenly we realize that perhaps the nothingness of our own viewpoints from nowhere is perhaps identical to our projection of manifested beings. These two horizons of transcendence in that moment collapse. The "designer" as subject in a flash disappears. Pure immanence and pure transcendence merge. All at once we are savages again. The ideational edifice has collapsed. Reason has become the same as madness. Order is indistinguishable from disorder, yet the human species has been altered irrevocably. The human species has been redesigned to adapt to the externalized agents that now control it. From the unconscious, or the world of spirits, or the realm of self-delusion the automatic writing has been done. The "designer" is elusive. But that elusive "designer" has used the meta-technology of software to rewrite who we are.

Differ*a*nce is the name of the movement that opens up the spacing wherein differences are seen. In that spacing -- between pure immanence and pure transcendence -- between Process Being and Nothingness -- between subject and Klir's object system -- there occurs the deferring of fixed relations between signifier and signified. There, in that spacing, the temporality of nihilistic artificial emergence and erratic change constantly turns over the soil, so the seed of a non-nihilistic distinction may be planted and grow into the tree of genuine emergence. In Old English, the words "tree" and "true" are related by vowel shifts. The world was envisaged as the tree of Yaggrasil. When the Kurgan people roamed the great forest that was Europe, the tree seemed to be the dominant structure in the world connecting heaven and earth. The straight tree did that best, and was called true. In our times, only those genuine emergences rooted in all four kinds of being stand out in the nihilistic landscape that has replaced the great forests of Europe and the Americas. "Software" as a concept is one of those great trees within the nihilistic landscape. Software is a kind of entity whose roots go deep. It reaches down to tap the ground waters of Hyper Being, and becomes a means of reaching even deeper to the level of Wild Being. Then knowledgeware becomes embodied in software. Software reaches through the cracks in the metaphysical bedrocks and taps the primordial writing that expresses differance. The automated/automatic writing of difference is the face of the “designer” who redesigns the subject. The origin of the forms that the “designer” imposes on the subject are always already lost. By definition, the origin cannot be made present. As we look into the face of the meta-technical phenomenon, we see the fragmentation of our subjectivity. As designers of automata, we eventually have to allow them to become nondeterministic (as they embody knowledgeware). That is no longer controlled by us. As designers of ourselves, we begin to evolve as organic artifacts. As designers of the bionic combination of these two evolutions, we become the lost origin of a different kind of creation -- ‘the ubermann’.

> When Zarathustra arrived at the nearest of towns lying against the forest, he found in that very place many people assembled in the market square; for it had been announced that a tight-rope walker would be appearing. Zarathustra spoke this to the people:
>
> “I teach you the superman. Man is something that should be overcome. What have you done to

> overcome him?
>
> All creatures hitherto create something beyond themselves; and do you want to be the ebb of this great tide, and return to the animals rather than overcome man?"[188]

In the Gameplayers of Zan, M. A. Foster imagines the “Ler” as the next step in human evolution. They are totally involved in playing the game of “Zan” which is their interface to the ultimate machine which is an autopoetic spaceship. This wedding between the creatures we may create from our organic basis and the autopoetic automatons we may create from inorganic projections of subjectivity, is brought to life in Foster’s classic science fiction. But this deep possibility owes its potential to the genuine emergence of the archetype of software as a fundamental means of exercising power. Given this tool, our will-to-power enters a new realm in which meta-technology takes over as the dominant theme from technology. Whether Foster’s vision is realized or not, there will be profound changes from the fragmentation of Being. And the intensification of technology as meta-technology is not the end. After the full flowering of meta-technology, comes the final

[188] Nietzsche, Thus Spake Zarathustra, Page 41 See Bib#.

state of proto-technology in which we return to savagery and confront chaos directly.

> “Very well. What does the game have to do with piloting? I know the game, thanks to Krisshatem, but I fail to see...”
>
> “The hard question; thus the hard answer. Let me build a dynamic identification series for you: consider vehicles. You make a cart, a wagon, hitch it to a pony, and off you go. Its purpose is to go, but it can be stopped, and it doesn’t change, or stop being a cart. Yes? Now consider a bicycle, which must be in balance to go. Yes? Now an aircraft; it can only be stopped when it is finished being a functional airplane, yes? You can’t stop it just anywhere, and never in the air, unless you have rotary wings, which is just cheating the system. Yes? Just so the leap to the ship. It is a quantum leap into a new concept in machines. If indeed that is the proper word. Before, we had machines that could be turned off. The more complex they became, the harder to turn off. With the ship, we enter the concept-world of machines that can’t be turned off -- at all. They must be on to exist. Once you reach a certain stage in the assembly of it, it’s <u>on</u>, and that’s all there is to it. And when you build it, you are building something very specific; that is the law of multiplexity. The more developed the machine, the more unique it becomes.”
>
> “So then,” she continued, "this machine can only operate, be on, exist in one mode only. A spaceship which can’t be turned off. Now in the true-mode of that existence, the laws we limited living creatures perceive about the universe are so distorted that they may as well not apply. One doesn’t look out a window to see where one is going! The kind of space that the ship perceives, operates in, is to creatures such as you and I, chaotic, meaningless, and <u>dangerous</u>, when perceived directly, if we can at all. To confront it

> directly is destructive to the primitive mind, indeed the whole vertebrate nervous system. At present, Dragonfly Lodge thinks this underlying reality -- universe is destructive to all minds, whatever their configuration, life and form or robotic. Basic to the universe: that is, its innermost reality cannot be perceived. A limit. So we interpose a symbolizer, and that translates the view into something we can perceive and control. And we must control it, for like the sailing ship our ship emulates, it cannot exist uncontrolled, and there can be no automation to do it for us (the Ler). It is flown manually, all the time; even to hold it in place relative to our perceptual field. For at the level of reality we are operating at here, to perceive is to manipulate. As you go further down into mystery, they become more and more similar; even the forerunners (the last men) know that. But at the ship they converge!![189]

The automata ceases to exist when turned off. Software needs the executing hardware. When we have projected ourselves totally into virtual space via our cybernetic creatures like "knowbots" and the system goes down, what will be left over? The overman will probably be totally locked into his narcissistic interaction with the automata like the Ler. Perhaps he will have already discarded his creator "homo sapiens" when the system goes down. Once genetic engineering has taken place, there is no going back. Little improvements here and there finally lead to something quite different, even barring intentional reengineering. The

[189] Gameplayers of Zan, M. A. Foster, Page 370, See Bib#.

technical adaptation of the human, so that man no longer confronts technology, occurs through meta-technology. Hardware and organic wetware merged through software and became symbiotically merged. Knowledgeware governs the synthesis. What cannot merge becomes the discarded and the savage -- the proto-Ler or last man.

In software, the destiny of humanity is written. Visions from science fiction are not needed to tell us that important events for the fate of humanity are happening in our times. Software meta-methodology has a pivotal role to play in these events.

◆ ◆ ◆

Dedication: This paper was written over the period from September of 1989 to June of 1990. It was just in this period that my son, aged eight, Shuaib Ibrahim Palmer suddenly grew ill and finally died. In November he fell ill and was discovered to have a brain tumor. On January 22 he had his first operation to remove the tumor. The tumor turned out to be an epidermoidal cyst in the third ventrical of the brain attached to the Hypothalmus. This first operation was a success, but within a few months the cyst had grown back. A second operation was attempted on May 18th, and Shuaib died on May 25th from a massive stroke after showing initial signs of recovery. In the brief period between his two operatons, Shuaib spent a great deal of time drawing and painting and attained unusual proficiency for a boy his age. Shuaib, the inventor of many wonderful things, was created with an extremely rare flaw which suddenly manifested and claimed his life. As with all the creations of God, both he and his flaw expressed the perfection of God’s work. This essay, with all its inherent imperfection, is dedicated to my son, whom I love, in death as in life, with all my heart.

■ ■ ■

KEYWORDS:

Software Engineering, Software Design, Software Methodologies, Ontology, Theory, Being, Philosophy of Science, Technology, Formal Systems, Structural Systems, Systems Meta-methodology.

ABSTRACT OF SERIES:

Software Engineering exemplifies many of the major issues in Philosophy of Science. This is because of its close relation between a non-representable software theory that is the product of design and the testing of the software. Software Engineering methods are being developed to make various representations of the software design theory. These methods need to be related to each other by a general paradigm. This paper proposes a deep paradigm motivated by recent developments in Western ontology which explains the generally recognized difficulties in developing software. The paradigm situates the underlying field within which methods operate and explains the nature of that field which necessitates the multiple perspectives on the software design theory. Part One deals with new perspectives on the ontological foundations of software engineering based on recent developments in Western philosophy. Part Two deals with the meta-methodology of software engineering by laying out the structural relations between various software architectural methods.

OVERVIEW OF PART TWO:

In this second part, the ontological results of the first part will be extended by a study of specific software engineering methodologies and how they work together in the design of real-time embedded software systems. The fitting together of methods into a coherent whole is the domain of systems meta-methodology as defined by Klir. Software engineering uses general systems theory as a foundation for describing structural systems. Structural systems provide the architectural framework for defining software systems. Software systems design requires the development of four specific points of view. These points of view require methods as bridges that allow movement between viewpoints. This section explores the structural infra-structure of the method bridges between design viewpoints.

[SEFpart2j/draft6/930908kdp (X90-999/601)]

Apeiron Press

PO Box 4402
Garden Grove,
California 92842-4402

714-638-7376
714-638-1210
palmer@think.net
palmer@netcom.com
palmer@exo.com
Dataline 714-638-0876

Draft #1 950710 Editorial Copy.
Not for distribution.

This book was set using Framemaker document publishing software by the author.

Electronic Version in Adobe Acrobat PDF available at http://server.snni.com:80/~palmer/homepage.html

Library of Congress
Cataloging in Publication Data

Palmer, Kent Duane

WILD SOFTWARE META-SYSTEMS

Bibliography
Includes Index

1. Philosophy-- Ontology
2. Software Engineering
3. Software Design Methods

I. Title

[XXX000.X00 199x]
9x-xxxxx
ISBN 0-xxx-xxxxx-x

Keywords:

Software, Design Methods, Ontology,
Integral Software Engineering
Methodology, Systems Theory