Chaos and Complexity

Lawrence A. Stabile

303 Third Street #517

Cambridge, Massachusetts, USA. 02142

lstabile@alum.mit.edu

Abstract

Relations are explored and developed among the mathematical theory of chaos, models of software, and philosophical characterizations of chaos and complexity.

A generalization of sensitivity and transitivity, termed model jumping, is defined and developed, and used to characterize behavior observed in software systems and more general physical and humanistic situations. The generalizations and characterizations are tied together via a simple computational model based on graph transformation.

Introduction

Chaos theory and its relationship to complexity have been studied very deeply over the past several years, resulting in a wealth of mathematical tools with broad application in science, engineering, and mathematics itself. Yet if we let our intuitions guide us with respect to what we perceive as complex or chaotic, there is a wide range of behavior that chaos theory does not cover well.

Consider for instance a natural disaster such as an earthquake, and its effect on lives, structures, and economies. During its occurrence in a city we would likely describe the scene as chaotic, perhaps "highly chaotic." In the aftermath, as recovery proceeds, chaotic conditions will continue, but we generally perceive "less and less chaos" as time goes on.

But, if an earthquake of similar strength struck in an unpopulated mountain area, or under the sea, it would be difficult to assign a similar degree of chaos as in the city case. Yet, most of us would agree that the earthquake in the city is more chaotic.

We can also consider less disastrous, day-to-day activities. You may perceive some chaos in the work you do: One day, you have an orderly set of documents to review; the next day, a new customer requires a seemingly random set of features, and further they come from various sources within that customer's organization, with various apparently overlapping and possibly inconsistent attributes.

The term complexity similarly has at once intuitive appeal and ambiguity. A seemingly simple object may be considered complex if its internal mechanism is more intricate and entangled than its external behavior would appear to call for. One might make a kind of ball, for example, which bounces and flies like an ordinary ball, but when cracked open reveals a bizarre mechanism of springs, solenoids, sensors, and tiny computers. Our perception of complexity changes with what we know about how something works.

Another realm where we perceive varying levels of complexity and chaos is in software applications. Here we have an abundance of behaviors to study, ranging from the very simple to the dauntingly complex. Behaviors can also be observed from easily predictable to wildly chaotic and bizarre. We might experience the easily-flowing interaction with a simple game; the oddness of a word processor that shows a format correctly in one moment and changes it to something seemingly random the next; or the interactions among applications that cause unfathomable delays, crashes, or other errors. And, on the development side, studies and stories abound regarding the enormous efforts and expenses often required to get software to work – or give up trying. The world of software is fertile ground for studies of complexity and chaos.

In the intuitive cases as exemplified by the above, our perception gives us a rough guide for what we term to be chaotic or complex. But, we have few formal ways to make these intuitions more precise. If we turn

to chaos theory, we see many applications in fields with a good formal foundation already, such as physics. Importantly, most studies of chaos define it in a binary manner: a system is either chaotic or it is not.

Chaos Theory

Summary of the Elements of Chaos Theory

Several mathematical definitions of chaos may be found in the literature. Generally here we refer to the basic concepts of discrete chaos, as defined by iterating one-dimensional maps. For the most part, these are defined on $\mathbf{R}$ (the real numbers[1]) or intervals thereof, as the compositional sequence, or *orbit*, of a map f on an initial value x_0:

$$x_1 = f(x_0)$$
$$x_n = f(x_{n-1})$$

A dynamical system f: $X \rightarrow X$, $X \subset \mathbf{R}$, is *topologically transitive* if, for any open sets $U, V \subset X$, there exists an integer $n > 0$ such that

$$f^n(U) \cap V \neq \phi$$

The intuitive notion of transitivity is that the system eventually finds its way to every neighborhood of every point of X. Related concepts are *expansiveness*:[2] a map is expansive if any interval iterates eventually to the whole space; *mixing*: for any open sets $U, V \subset X$, there exists an integer N such that for all $n >$ N,

$$f^n(U) \cap V \neq \phi;$$

and *blending*:[3] for any open sets $U, V \subset X$, there exists an integer $n > 0$ such that

$$f^n(U) \cap f^n(V) \neq \phi$$

Note that while we have defined the space X as a subset of $\mathbf{R}$, transitivity can apply to the weakest of spaces; in particular a metric space is not required in its definition.

A set $S \subset X$ is *dense* in X if every open subset of X contains an element of S. The rationals Q, for example, are dense in $\mathbf{R}$. In a dynamical system X, the set of periodic points is all elements of X that belong to a periodic orbit. If the set of periodic points is dense in X, then that means that any open subset of X contains at least one periodic point.

The *limit inferior* of a sequence of real numbers is the low limit to which the sequence converges; the *limit superior* is the high limit to which a sequence converges. The sequence generally does not converge to a constant value, and we can think of sequences that have distinct limits inferior and superior as in some sense alternating.

A dynamical system f: $X \rightarrow X$, $X \subset \mathbf{R}$ is *sensitive* if there exists an $n > 0$ such that for any $x, y \in X$ there exists an $\varepsilon > 0$ such that

$$\left| f^n(x) - f^n(y) \right| \geq \varepsilon$$

Such divergence is at best exponential for all but the simplest maps (e.g., $f(x) = x + c$), and is indicated by the value of the *Lyapunov exponent*, which is defined as the logarithm of *Lyapunov number*, which is the per-orbit-step multiplicative divergence from an infinitesimal variation in initial condition. For the n^{th} orbit step of f, the Lyapunov number L_n is

---

[1] We will use the bold $\mathbf{R}$ to denote the reals; normal uppercase R has a particular use in subsequent sections.

[2] See Gilmore [10].

[3] Mixing and blending have strong and weak variants. Given here are the weak forms. See Fotiou [14] for a survey of types of chaos, including descriptions and comparisons of transitivity, mixing, blending, and other concepts.

$$L_n = \left|(f^n)'(x_n)\right|^{\frac{1}{n}}$$

If $(x_1, x_2,\ldots,x_n)$ is the n-step orbit of f, then by the chain rule

$$L_n = \left|f'(x_1)f'(x_2)\ldots f'(x_n)\right|^{\frac{1}{n}}$$

The Lyapunov exponent l is then the logarithm of L_n in the limit, thus

$$l = \lim_{n\to\infty}\frac{1}{n}\sum_{k=1}^{n}\ln\left|f'(x_k)\right|$$

A positive Lyapunox exponent indicates that a system is sensitive, and methods of calculating an approximation to the Lyapunov exponent are important in determining whether some given physical system is chaotic.

Some Definitions of Chaos

For discrete-time maps, Alligood [1] defines chaos on $\mathbf{R}$ as requiring aperiodicity and sensitivity, on a bounded orbit.

Li-Yorke chaos [4] is defined in terms of uncountable "scrambled" sets, where scrambled means that, in a discrete system, given two distinct initial points, their orbits converge by a zero limit inferior and diverge by a positive limit superior. Li and Yorke proved that a system with a periodic orbit of length three has periodic orbits of all lengths, and at least one aperiodic scrambled (i.e., chaotic) orbit.

Devaney's definition [5], which has proven popular, requires topological transitivity, density of periodic points, and sensitivity. Much work has shown that the conditions are related and redundant under various constraints on their spaces. For example, for continuous maps on intervals in $\mathbf{R}$, transitivity implies the other two conditions (see Elaydi [2]); thus transitivity in this restricted space is all that is needed to show chaos.

While all of these definitions rely on $\mathbf{R}$ for the derivation of a good portion of their interesting results, Devaney's definition seems to have spawned the most work in generalizing the concepts. Transitivity, for instance, requires no special conditions on the space; it does not even need to be a topological space.

Interpretations of the existence of chaos in physical systems often rely on methods to determine whether a system is sensitive and aperiodic.

Sensitivity is determined by calculating the Lyapunov exponent from measured data, given some definition of deviation from an idealized model. For instance, in Sussman, [12, page 434] a localized linear model is compared with actual trajectories and on this basis the Lyupanov exponent is computed.

Aperiodicity is determined by spectral breadth and density. Again, for example, in Sussman [12] we see that the planetary orbit of Neptune has a distinct line-spectrum character, whereas the spectrum of Pluto is has a more continuous appearance.

Generalizing the Intuitions

Transitivity and Sensitivity

The idea of transitivity, and related concepts like blending and expansiveness, represent a notion of *wandering*. A generalization of this leads us to ask *in what manner* the wandering occurs. We will turn to models to describe kinds of wandering and will in particular explore the idea that wandering from model to model provides a good framework on which to base further notions of the quantification of chaos and complexity.

Sensitivity, another defining basis for chaos, is deeply rooted in metric spaces. Here I refer to both the formal idea of metric space as well as its corresponding appeal to our intuition in defining what "distance" and "size" mean. In our everyday thinking and discourse we use these ideas in many contexts; they are

pervasive as analogs to many kinds of concepts. Yet we usually cannot define formally what we mean when we use these analogies. What does it mean to say a system is sensitive? We perceive it as "something small that caused something large", but normally don't explicitly quantify what we mean by those terms.

Perhaps the simplest intuitive basis for something to be sensitive is to say that, within the framework of some model, the system moves outside that model. When I turn a volume control knob by a small amount, I expect a small change in sound volume. If, say, due to material flaws, the volume suddenly increases greatly, then I am suddenly thrust into a new, unexpected model, one of annoyingly loud sound.

The descriptions above of the two ideas of general transitivity and general sensitivity appear very similar. We combine these ideas together by the concept of *model jumping*, where we examine the sequence of models of a system as it moves through its orbit, and the relationships among the models themselves. By this combination we capture the intuitions that wandering is like transitivity, a series of small steps, with varying degrees of knowledge about where the steps will end up; and sensitivity, that sometimes big leaps are taken.

The generalization also allows us to describe systems that may be decomposed into components, while nevertheless retaining notions of chaos. Transitivity and sensitivity, as formally defined in the literature, are global properties of dynamical systems.

Density of Periodic Points

While transitivity and sensitivity generalize fairly naturally to notions of set containment, density of periodic points does not generalize as easily. In particular, the spaces under consideration in this paper are of an arbitrary nature and may not be globally metric or topological (even though there might be local sub-spaces that are).

One of the qualitative hallmarks of chaos, quoting from Devaney [5], is "in the midst of this random behavior, we nevertheless have an element of regularity, namely the periodic points which are dense." The spirit of this in the work herein is the periodicity of graph structure which may be found embedded in the models we contemplate. For instance we may model day-to-day activity as a kind of periodic behavior; it may be disrupted by some unpredicted event, such as a natural disaster.

In our graph model, a recurring pattern in a time-like relation chain has these sorts of aspects of periodicity. Ideal (infinite) periodicity is conveyed by looping the time-like relation, i.e., we have a cycle in the graph. However in the real world there can be no cycles. Any *model* with a cycle must also have relationships with other models which modify the periodic model and denote that it is an approximation (i.e., a meta-model is required; see *Meta-Models*, page 12). A model which stands alone is effectively timeless.

In a model of the real world, global periodicity is not expected to be found. Even locally, objects in a recurring pattern may not be exactly the same, and the recurring pattern may not continue forever. Models of periodicity we build via mathematical tools are necessarily idealized. Nevertheless we find a great deal of utility in assuming these models apply to the real world, i.e., that they can be implemented to a sufficient degree of approximation that the predictions of the idealized model hold.

Complexity

Internal Mechanism versus Externally Observable Behavior

When we ask whether a system is complex, most of us will try to imagine the internal workings of the system, and fashion an answer based on the number of parts, their interconnectedness, homogeneity versus heterogeneity, and so on. So, for instance, most of us will say that a human is complex and that an amoeba, by comparison, is simple. Clearly they differ in terms of size -- numbers of components -- but is a human really any more complex than an amoeba? An amoeba has an intricate set of parts, all working at the molecular level. A human has many such amoeba-like parts, differentiated. Is this enough to say one is more complex than the other?

Computer systems and software also have perceived complexities. Some programs are easy to use, some are hard, even for the same kind of application. Is the hard one more complex? To a computer engineer, these programs might have the opposite complexities internally: the cumbersome-to-use program may have

a simple, elegant internal design whereas the easy-to-use program may comprise tightly interconnected components, many patches, and a hodge-podge of changes.

Interesting experiments have been performed in genetic algorithms which illustrate the internal/external complexity question. For example Adrian Thompson [13] evolved an FPGA to discriminate audio tones. Had an electrical engineer designed such a circuit, it would have likely have been fairly straightforward, and easily comprehended by other electrical engineers. However, the evolved circuit was strange and esoteric by comparison, and much study was required to fathom how it actually worked.

Our perception of the complexity of a system varies with respect to our model of its structure, which in turn we infer from its external behavior, and concepts we have generalized from observing other systems. Generalizations of complexity then, as with generalizations of chaos, depend on how we structure models and the relations among them.

Summary of Computational Model: The H-machine

The computational model used herein, a form of graph automaton, is simple but serves to ground the ideas presented. Models of this kind have been extensively studied with respect to their fundamental computational properties and limitations. A deeper discussion can be found in Stabile (H-Machine, [8]).

The computational space is a hypergraph,[4] which grows according to rules, which are themselves pattern subgraphs of the hypergraph. Given an initial graph[5] and set of rules, computation proceeds by testing each rule against each subgraph of the current graph, and applying those rules which match.

The result of an application is the creation only of new edges. We assume that an initial graph is a countably infinite set of nodes, endowed with some initial edge set. In its most basic form, this is the infinite sequence of where each node is related to the next by a sigma function, as in axiomatizations of the natural numbers. However other initial graphs are easy to imagine.

Rules are part of the system and can themselves be matched against other rules and generated. This is important for representing meta-levels.

Functions are represented literally, as mappings in the graph. Rules act to generate the mappings, i.e., rules define the function, and when the value of a function is required we assume the mapping exists and simply look up the value via the instantiated mapping.

A machine based on the above attributes is called an H-machine.

Note that although the computational universality of this sort of model has been well-analyzed in the literature, it is not addressed by the work herein.

Hypergraphs can represent broad classes of objects and meta-objects. Within this system we can represent objects and relations that allow mappings to abstract objects and relations, so that both realization and model layers utilize the same representational framework. Hierarchical construction, i.e., elements of cross products of sets, potentially special relations representing time, and an appropriately rich rule description system are all necessary elements.

A computation unfolds on a current graph G_n and driven forward by applying rules. Each step, formally defined as an exhaustive enumeration of subgraph-rule tests, is a step in the *orbit* of G. The evolution of the graph is monotonic: A time-like relation may be defined that holds between all objects of one time-step and another; nodes and edges are never deleted.

---

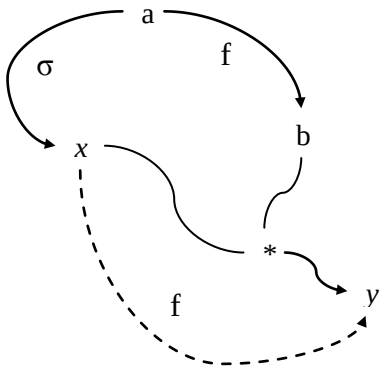
[4] We will take some liberties with the definition of a hypergraph and sub-hypergraph to simplify the exposition. A hypergraph H is a collection of sets $\{H_i\}$. The vertices of H are defined as $\bigcup_i H_i$. A sub-hypergraph is defined as $J \subset H$, with vertex set $\bigcup_i J_i$. This simplification loses no generality and admits common set notation.

[5] Herein the term "graph" will mean "hypergraph" unless otherwise specified.

This process is somewhat more formally stated as follows, where G_n represents the current graph. To support the fundamental idea that rules and data may be acted upon and generated within the system, we presume that rules are present as subgraphs within the overall graph which forms the current system state, and that we can distinguish a rule subgraph from one which is not a rule. This is the only inherent special kind of knowledge which must be encoded into the system.

$$U \equiv \text{a set}$$
$$H = powerset(U), \text{the complete hypergraph on } U$$
$$h \subset H, \text{some hypergraph}$$
$$subgraphs(h) \equiv \text{collection of all subgraphs of } h$$
$$r(h_r, h_d) = \phi \text{ if } h_r \text{ is not a rule or does not match } h_d, \text{ else a graph} (\subset H)$$
$$G_n \subset H, \text{current hypergraph}$$
$$G_n = G_{n-1} \cup \bigcup_{i,j} r(subgraphs(G_{n-1})_i, subgraphs(G_{n-1})_j)$$

Rule matching and consequent generation operate roughly as follows. A rule graph denotes both its pattern part and its consequent part, illustrated below in a rule which might be defined for the recursive segment of the factorial function. The solid-line edges are the pattern and the dotted line the consequent: a pattern solid-line match then causes the dotted-line edge or edges to be created in the matched data graph.



Matching is by subgraph isomorphism, i.e., we seek an isomorphism between the pattern graph and some subgraph of the data graph. Note that the rule graph contains no distinguished "variable" nodes; the only special notation needed is of the consequent edges. Use of the variable-style syntax *x* and *y* in the figure above is purely expository. All nodes matched by the rule test will have an isomorphic counterpart in the data graph, and those nodes are implicitly used in creating the resulting edges.

We can rephrase the example above in standard logic as follows:

$$x = \sigma(a) \ \wedge \ b = f(a) \ \wedge \ y = x*b \ \rightarrow \ y = f(x)$$

Note that although the fundamental execution model of the H-machine is with respect to orbit steps, there is inherent parallelism available, due to the monotonicity of the data and the associativity of the operations applied to evolve the graph. If the system is non-deterministic, assume as a basis that all possible avenues are instantiated. An H-machine is fundamentally an exhaustive forward-chaining rule system.

We'll use ordinary graphs as illustrations at the informal level, depending on an intuitive interpretation of how it might generalize to hypergraphs. Most annotated graph styles used as computational descriptions can be translated easily to hypergraphs; indeed we expect k-uniform hypergraphs to be the norm, where k is a small integer.

It may be noted that we specify the computational action as an enumeration of subsets of the current graph. However the initial graph is a countably infinite set. The enumeration called for is thus uncountable and hence not computable; therefore some means of approximation is necessary for any actual construction of an H-machine. Stabile (H-Machine, [8]) discusses these practical aspects of the H-machine such as approximation, representation, parallelism, and optimization.

Models

Structure

A model is, most generally, simply a mapping between two graphs. Constraints, as enforced by rules, define the semantics that make the mapping interesting. Typically the graph will be structured as objects, attributes, and relations which have some meaning to humans. However using only the idea of a mapping allows us to infer a great deal without delving widely into the semantics of the system.

True to the idea of an H-Machine, there is only one attribute that we require to be represented in the graph, that of a rule.

We define a "primitive" H machine as R, a *realization layer*. Then, define a *model layer* M as a mapping R −> M. There is a set of rules and objects associated with M, which builds the model from R. Either that machinery is not part of R or it is. If so, then we have *reified* the representation. The focus herein is on non-reified systems, although there is some discussion of reification in Stabile (H-Machine, [8]), and studies of reified systems may be found in the literature (see Nierstrasz [11] for a good summary).

Note that the realization layer is the only level at which we can say the "reality" exists; all models mapped therefrom are abstractions. An advantage of this kind of representation is that both low and high level abstractions are captured in one uniform framework.

The model layer M may in turn be decomposed into a set of interconnected models, {Mi}. Most generally, we can dispense with a special R layer and simply talk of interactions among models; adding an R is a special case.

The realization layer R is driven forward by applying rules. Each step, formally defined as an exhaustive enumeration of subgraph-rule tests, is a step in the *orbit* of R. The evolution of the graph is monotonic: A time-like relation may be defined that holds between all objects of one time-step and another; nodes and edges are never deleted.

A simple example of a system with a bottom layer R and upper abstract models M is given by basic physics. Suppose that the subatomic particles protons, neutrons, and electrons, plus any needed energy particles or fields, were the bottom layer R. Then this is all that needs to be considered as really existing and thus all that evolves over time. Upper level objects are then defined in M: atoms, molecules, rocks, cells, etc. Their existence is purely abstract and is implied by R. Of course, this is by its nature a reified system, since M must reside somewhere in R, say as bits in a computer, an electrical configuration of neurons in our brains, or as marks on pieces of paper.

Observers

As evolution of the realization layer R unfolds, rules not part of R construct one or more models of R, based on the formal definition of a model in *Model Layers* on page 11. Models of models are possible and R is simply the chosen base-level primitive model, the system driver. We'll continue to denote R as the most primitive layer and $\{M_i\}$ as the set of models beginning with R. We will use the notation M_i −> M_j to signify that a model function exists from M_i to M_j. Model functions need not be total, and in fact are probably not interesting if they are: A total model is one that characterizes its underlying mechanism completely. Capturing everything in a single model may be useful in some context, but our purpose here is to characterize relations among models – so we need more than one.

A model function is the mapping defined by the set of rules associated with the target model. As graph evolution proceeds, instances of that mapping will be created. It is possible that at some point no map instance may be created – i.e., the model function is not total.

Given models M_0 and M_1, and model function f: M_0 −> M_1, model M_1 is *active* relative to model M_0 and orbit step n iff $f(M_0(n))$ exists.

The set of all active models at orbit step n is the *model structure* at step n.

Another important relation is that of observation. There may be more than one *observer*, and the set of them also forms a structure of potentially interesting relations. Observers are themselves objects in R which in turn instantiate models; however without specifying their nature in R, nor a model function to some M,

we prescribe that their model structure remain constant at each orbit step. Observers thus anchor a sequence of models as the orbit of R unfolds.

A model structure truly embedded in R can be said to be *reify* the system. This embedding is interesting but we won't dwell on it here.

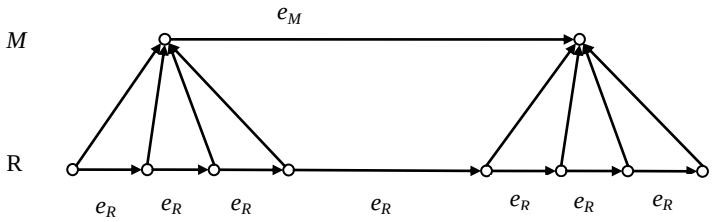
Evolution Relations

The orbit of R evolves by the addition of one or more edges into the graph G_n to produce G_{n+1}. A *deterministic, total evolution* is a set of bijective relations $\{e_n\}$ on $N = nodes(G)$, denoted by the edge label e, e_n: G_n −e−> G_{n+1}. This set of relations forms a chain of nodes; each step along the chain can be thought of as a step forward in time. When it's unambiguous that every node has a counterpart forward in time, and one and only one such counterpart, we say that the time relation is deterministic and total. We imagine that R, as a base-level representation, should be deterministic and total.

In model space, the evolution relation can be non-deterministic; i.e., branches may occur. They may also converge; in some sense we only model "reality" in which some observer selects one and only one evolution to call true for that model. The result of this in the evolved graph is that a model function will exist between one evolved branch of a model to an observer, but not between other evolved branches and that observer.

A model's evolution relation can be non-total; in this sense nodes "cease to exist." We'll call models which are total *conservative*. Note that a model can be conservative with respect to some chosen set of relations $C = \{C_i\}$. A node "ceases to exist" when it is removed from membership in all of the evolution relations, i.e., when no edge labeled with any C_i connects the node to another.

Perhaps most importantly, the mapping from evolution steps in one model to those of another need not be one-to-one. So, for example, if *M* is a model of R, then several orbit steps of R may map to the same model object in *M*.



If we think of the evolution relation as time, then only pairs of nodes linked by an evolution relation are considered to be time related. Informally this is similar to saying that if an object doesn't change then time does not pass for that object. An observer's perception of time can be gauged by the model under observation. At least one model under observation must change for time to be considered to pass. And, while we can think of the granularity of time being different at different model levels, it is just as valid to say that the very definition of time is simply different at these levels; it depends on the observer and the model under observation.

Model Jumping

A *model jump* is a change in the active structure of the set of models $\{M_i\}$.

Model jumping is an inherently discrete action, and has no continuous analog. It can be absolute, where we follow a single model sequence through the set of changes, or relative, i.e., a change in one model relative to a change in another.

It also helps apply the ideas of chaos to finite spaces, as we no longer need to speak purely of aperiodic (infinite) orbits to convey ideas of chaotic behavior.

Sensitivity

One important relative model jump characterizes sensitivity. Informally, we define this as saying that when two sets R_a and R_b in R are proper subsets of a common set R_s, and R_s model-maps to M_1, then the system

is *insensitive* if M_1 orbit-maps to the *same* model M_1 for any element of R_s, including therefore members of R_a or R_b; the system is *sensitive* when R_a and R_b orbit-map to elements of R that have a *different* model M_2.

This is illustrated in figures Figure 1and Figure 2. It can be stated more formally as follows.

First define the following sets, subsets, and orbits:

$$R_{a_1} \subset R_s$$
$$R_{a_2} \subset R_s$$
$$R_{a_1} \cap R_{a_2} = \phi$$
$$R_s \subset R$$
$$R_{b_1} = r^n(R_{a_1})$$
$$R_{b_2} = r^n(R_{a_2})$$

In an insensitive system only one model is required regardless of the orbit taken from Rs:

$$M_1 = f(R_s)$$
$$M_1 = f(R_{b_1})$$
$$M_1 = f(R_{b_2})$$

A sensitive system on the other hand requires two models:

$$M_1 = f(R_s)$$
$$M_2 = f(R_{b_1})$$
$$M_2 = f(R_{b_2})$$

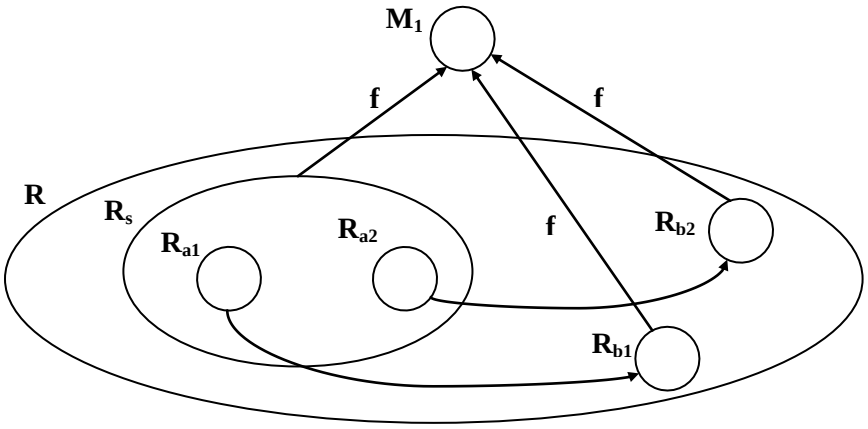


Figure 1 – An insensitive system.

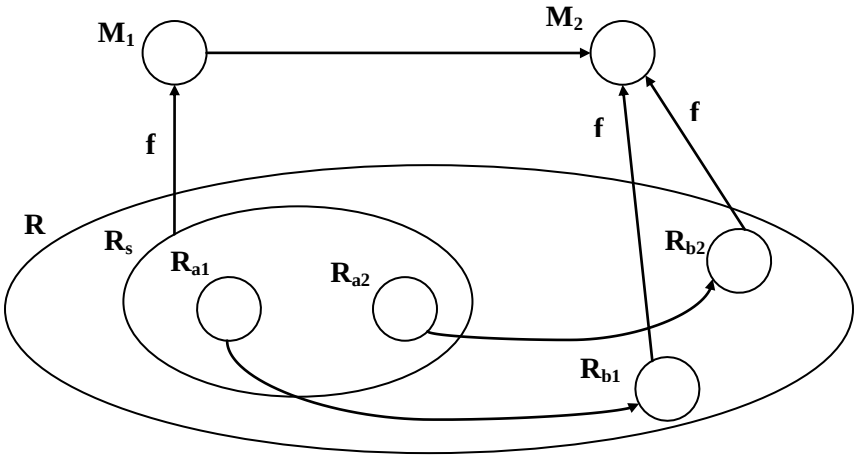


Figure 2 -- A sensitive system.

One possible generalization is that sensitivity increases with the number of models which can be reached from subsets of the same set. Zero sensitivity is $M_1 -> M_1$. This is illustrated in Figure 3.

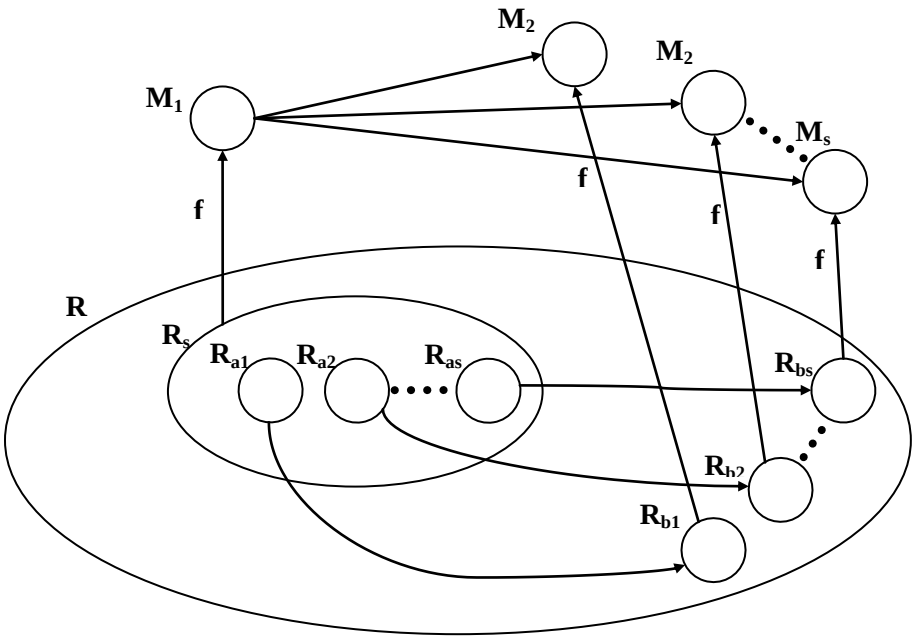


Figure 3 -- One possible generalization of sensitivity, where sensitivity increases with the number of models *s* accessible from a given initial set R_s.

Note that we need to distinguish our description of actual model behavior, e.g., "the system behaved in a sensitive manner" from statements *about* the model, e.g., "this system has sensitive dependence". The latter is the usual formulation. It is a statement about the set of models, not a particular one. See *Meta-Models* on page 12 for a more detailed discussion.

Model Layers

A *model layer* is a subgraph of the overall H machine which represents the time evolution of a particular initial graph via its orbit-inducing map.

Well-Layered Models

A model is well-layered if all aspects of the model respect the layers of the hierarchy. Informally, this means that all effects upon the upper layers of the model flow smoothly from the result of the actions at the lower layers, via the aggregation functions. In essence, this is the familiar idea that a description at some layer is complete, and fully accounts for the expected behavior.

For instance, a set of particles which cohere into a ball should act according to the simplified (aggregate) model of a spherical object, whose mechanics is well-defined and whose behavior can be predicted pretty well. We don't track the motion of every particle to track the ball.

Similarly, subatomic particles, atoms, molecules, and higher-level objects are for the most part well-layered, since the aggregational models work most of the time.

A model is not well-layered when the effects of one layer unduly cross boundaries into another. If, for example, the aggregation functions are not the only way to confer a property from a lower layer to an upper layer, but that there is also a way for a specific lower-layer object to determine the behavior at the upper layer normally conveyed by the aggregate, then a *layering violation* occurs.

A *violation function g* maps a layer M_0 to a layer higher than its immediately superior layer M_1; this means that there is model M_2 mapped to by M_1 and at some state in the orbit of M_0 an instantiation from M_0 to M_2 is created by *g*. This is more formally stated as follows:

$$\exists n \text{ such that}$$
$$f_0(m_0^n(M_0^{init})) \text{ undefined} (\text{and hence } f_1(f_0(m_0^n(M_0^{init}))) \text{ undefined}) \text{ and}$$
$$g(m_0^n(M_0^{init})) \in M_2$$

See Figure 4 for an illustration of this idea.

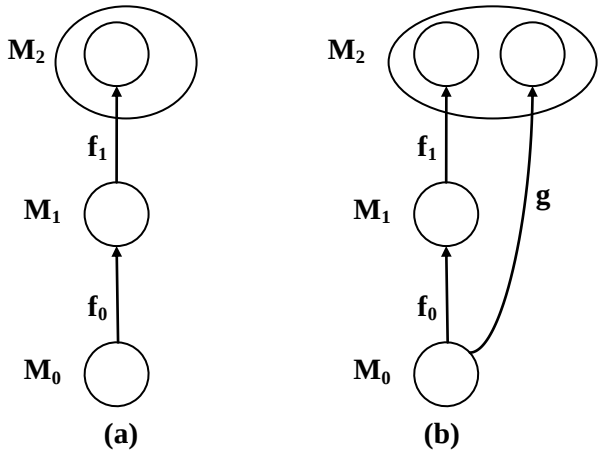


Figure 4 -- (a) A well-layered model; (b) Violation of layering

A simple example of a layering violation is nuclear radiation and its effect on chemical properties. In most matter with which we interact every day, radioactive decay is rare, and if it did not exist, atoms and molecules would happily live in their chemical hierarchy. But a single radioactive particle emission can

alter significantly the properties of an atom, and thus any molecule that contains it, and possibly other surrounding molecules as well. The hierarchical model's boundaries have thus been crossed, and the result is sometimes undesirable. Importantly, the behavior in this physical case is statistical: a relatively large number of molecules is typically required to be altered before the few alterations occur that cause disease or similar harm.

Contrast this to software systems, which use large-grain objects for which statistical approaches often don't help: if a component fails, the system typically crashes. They also often use complex, multi-layer models, and it is difficult keep them well-layered. I posit that layering violations, in various forms, are responsible for a great deal of the fragility found in software.

Sensitivity and Violation of Layering

Different kinds of sensitivity are manifested by variants on a simple model based on two parameters: the number of orbit steps, *n*, and the number of layers *l* skipped in the violation. We define four coarse-grain forms:

Form 1. When *n* is large and *l* is large, this is a rare situation, e.g., when a single radioactive particle emitted by an atom causes an earthquake. Behavior under this model is likely considered extremely interesting and unusual, perhaps catastrophic.

Form 2. When *n* is large and *l* is small, the behavior may be highly unusual, but not considered miraculous or strange, because we as humans are able to comprehend these short model skips fairly well. At the model levels of human social interaction, this is illustrated by social relationships, for example where people behave in uncharacteristic manners, or some seemingly minor aspect of the relationship between you and someone else has a major effect on your life after considerable time.

Form 3. When *n* is small and *l* is large, a great deal of confused and incomprehensible behavior may be present, such as being in the midst of a battle or natural disaster. It is difficult in this case to imagine a positive form of this kind of model skip.

Form 4. When *n* is small and *l* is small, we also have confusion, but it's likely contained within a region of model layers. As in form 2, we may perceive this in social situations, but the short-term nature of the behavior will be more like a soap opera than normal life. At lower levels diseases are an example of this form of model skip, in the sense that a relatively small change at a molecular level can induce a more significant change at the cellular level.

Meta-Models

The idea of meta-level, meta-system, or meta-language is common in many areas, particularly so in computer science and engineering. See Nierstrasz [11] for a good overview. Informally, a meta-level describes the structure of another system which implements behavior in some domain. One abstract way to look at this is that if we suppose the existence of a set *D* representing some domain, then one might define some model of that domain via a function *d*:

$$d: D -> D$$

A meta-model of that domain is thus a way to express functions on the model. We can denote an instance of such a meta-model as a function *e*:

$$e: (D->D)->(D->D)$$

Then, we close the loop by representing *e* within *D*, i.e., by defining functions of the form:

$$D->((D->D)->(D->D))$$

This expresses the idea of affecting *d* within *D*.

In the H-machine, the domain *D* is the system graph, and a rule represents a function d. Since rules are themselves members of the sets under consideration, i.e., subgraphs of the modeled objects, rules can act as the function e and instantiate new rules and thus support metalevel operations. The mechanism for this is described in detail in Stabile (H-Machine, [8]).

Sensitivity

Meta-models allow us to broaden the range of expression of the system. For example, sensitivity as described above is a differential property, i.e., it expresses possible behavior given changes in certain parameters. However, we also wish to represent actual sensitive behavior, for instance that I actually did adjust the volume by a small amount and cause a large change in sound level. To represent this, we must represent that some observer predicted one model but then got another. Hence a meta-level is required.

A schematic version of this idea is shown in Figure 5, where the meta-function **o** represents all the functional machinery necessary to model that an observer is able to simulate model **f** and observe that model **f'** actually occurred. The kind of model jump represented is the same as that associated with the definition of sensitivity illustrated by figure 2.

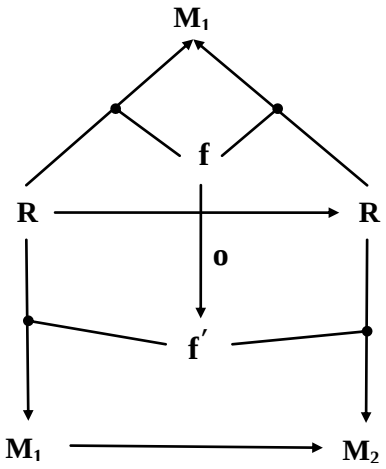


Figure 5 – An instance of sensitive behavior relative to observer o.

The Intelligent Observer

The further development of the idea of an observer is also a good use for metamodels. In particular, we can define a form of *intelligent observer* as one that compares a simulated realization layer to an actual one, where "actual" in this case is simply the R we have assigned as the independently-running model.

The key feature that makes this a simulation is that there is a mapping between rule sets such that the simulation can run independently of the realization layer. This is illustrated in Figure 6 as the object set R' and rule set r' (the simulation), under control of the observation function ***o***.

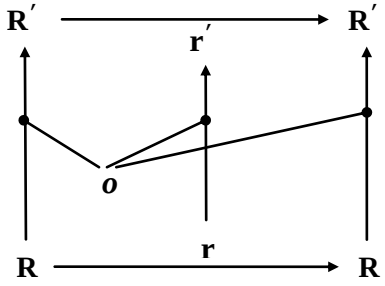


Figure 6 – The intelligent observer *o*.

We might also impose constraints on R' and r', for example that the model (R',r') is deterministic, and perhaps that model jumping in R' be minimal or zero, i.e., we confine the simulation to one or only a few levels. These two properties convey an intuitive notion of what we expect a simulation to be.

Philosophical Considerations

Below is a set of rough ideas for formalizing certain humanistic concepts in terms of the ideas presented above.

Dreams, Surrealism

Surrealism is characterized by distortions of the physical basis of reality, and a mapping of human concepts onto an idealized analogical space based in this distorted physical reality.

The distorted physical reality is often controlled by the upper level model. Dreams constitute one form of this feedback mapping in which the control is strong; effectively, the dream constitutes a model-induced reality.

Surrealism as an art form, then, finds a wealth of material in dreams and the concept of a dream. But surrealism covers more than do typical dreams, in that the physical discontinuities are of a more global nature, and are not so directly controlled by the semantic aspects of the upper level model.

For example, in a dream you may walk out your front door and step onto an ocean liner; a more extreme surreality might have both occur in a fish-eye-distorted, curved physical space.

The figures below illustrate a basic representation of a dream as compared to that of a surreality. Note that use is made here of meta-models, and it seems that meta-models are necessary to represent something like dreams.

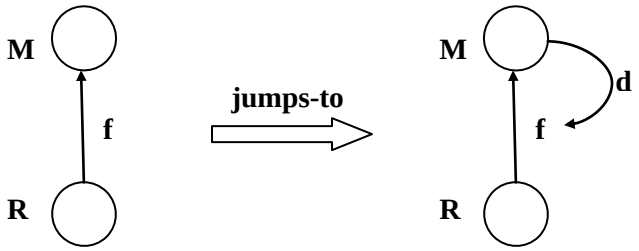


Figure 7 -- A model jump to a dream, with dream function d.

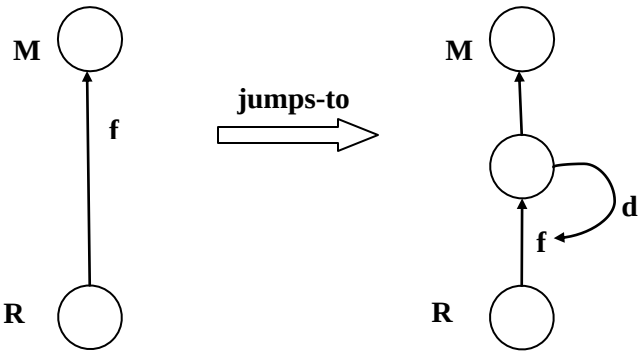


Figure 8 -- A model jump to a surreality. The feedback is via a lower level model than that of a dream.

Physics

Physics is a *reductionist science* -- at least as it has been formulated in the mainstream to the modern day – in that more has been learned by repeatedly breaking down descriptions into smaller components. There are two main ideas we need to take from this development:

1. The representations form a hierarchy, where each level describes components and relations which, when abstracted appropriately, define the next level up. The lower we descend into the hierarchy, the more regular the representations are. Thus we might have a hierarchy like Rocks −> Molecules −> Atoms −> Subatomic Particles. Or, adding levels above, Humans −> Cells −> Molecules −> Atoms −> Subatomic Particles.

   An important concept is that of *violating the hierarchy*: How does behavior at low levels in the hierarchy unduly affect behavior at the higher levels?

2. Reliably predictable behavior depends a great deal on the degree of redundancy in the system. The motion of a macro-scale object such as a rock, for example, will not be altered very much by the radioactive decay of a single atom. But such a decay in a single organic molecule in a cell can have a far-reaching effect, although there is normally redundancy at the next level up (cells) which counteracts it.

The redundancy of the system, and the degree to which the hierarchy is obeyed, form important dimensions of the sensitivity of the system to change.

Action at a distance

A*ction at a distance* is the observation of some (literally or figuratively) distant effect with an apparently local but unexplained cause. Our perception of it is generally one of surprise, at least until the cause is explained. For example, had you never seen an electric light and corresponding wall switch, you would be quite surprised at the apparent cause-effect link, until you understood the mechanism behind it. Software systems are very prone to this perceptual effect as well. Bugs can introduce surprising behavior that not only makes the system "sensitive", but causes it to behave oddly as well: perhaps, for instance, typing too quickly causes the cdrom drawer to open.

Analyzing action at a distance requires that we break down sensitivity. In characterizing sensitive behavior, we normally have some model of the mechanism involved in mind, and the model jump requires some adjustment to that mechanism. With action at a distance, however, there is no model – or at least the default model is vague and high-level.

Domain Mixing

Domain mixing may be described as follows:

1. Substitution of one object with another, where the models M_1 and M_2 of the objects are on the same level. This carries the case, for example, of substituting a cigar for a pen, perhaps in some absurdist theatre production.
2. Substitution as in (1) but where M_2 is an upper-level mapping of aspects of a lower-level model, not covered by M_1. This carries the case, in the software domain, of a bizarre error message from a lower level appearing, instead of a user-domain-level actionable message, e.g., "stack overflow, code 0xaeb09a1f3" vs "word misspelled – select correct word".

Domain mixing seems to be closely related to the philosophical term *category error*, as described in [15].

Absurdity

When we consider how the ideas of dreams, domain mixing, action at a distance, and the other concepts described above, alone or in composition, may be generally categorized, we arrive at the notion of *absurdity*, well-trodden ground in logic, literature, and art.

We can capture absurdity in general as examples of the concepts in this section, or compositions thereof.

Chaos and Complexity in Software Systems

Types of Model Jumps

In software we find at least the following kinds of model differences:

- **Domain corruption, e.g., memory overwrite**: sensitive; leads typically to upper-level model jumps which is often an inconvenience, sometimes a disaster.
- **Violation of Layering -- Domain mixing**. One of the most basic forms of violation of layering. Usually not sensitive, but can cause a wide variety of behaviors. Users see objects they weren't meant to see, but which nevertheless are at a human model level. These in turn typically lead to inconveniences; less often to disasters.
- **Functional difference**. Closely related to domain corruption, the functional difference pattern occurs when the model mapping function for a set objects in a model should follow some uniform rule, but some objects have a rule attached that is "slightly" different from the others. A conjecture is that this can be viewed as a domain crossing in the rule domains.

  When the functional differences induce small numbers of partitions on the objects, as we might have when we "clone" a function once or twice, the effects are typically immediate so this is not considered inherently sensitive. Behaviors manifested are often inconvenient and while disasters do occur it's common simply to be beset by thousands of inconveniences.

  When only a few objects out of many will be affected by a correspondingly few differences in rules, sensitivity can arise more easily, since normal behavior is likely to stay within the bounds of the bulk of the objects. When a system veers into these ill-defined regions, it's likely a disaster.

Software Systems and the Sensitivity of Violation of Layering

Software systems are constructed with fairly deep models, and layering violations of all four types above (see *Sensitivity and Violation of Layering*) can be found. However, small to medium skips in layers would appear common, with large skips as in Form 1 less so. Typically it seems that Form 1 skips will lead to crashes or similar difficult to handle situations, whereas as shorter model skips evoke absurdities and inconveniences.

Consider for example your interaction with an online book-shopping application. The domain appears to you, the user, as books, prices, shopping carts, and so forth, and the operations on these objects necessary to order a book. Suppose in the course of these actions a domain-level error occurs, such as selecting a book which is out of stock. An appropriate response, remaining in the book-shopping domain, might be to

display a message saying “Sorry, out of stock, would you like to back-order?” If instead a message is displayed saying, “Access violation, process_lock(), Plock.java, line 54, …[more cryptic stuff]…”, a layering violation has occurred, and you have to jump to a new model to explain the situation.

This case is one where *l* is reasonably large and *n* is typically small, as in Form 3 above. While not a disaster, it’s at best a minor inconvenience and at worst a significant headache.

A layering violation closer to Form 1 is a machine crash (as in “blue screen of death”) after some fairly long period of time. Both *n* and *l* are larger than in the book-shopping case, and the effect is correspondingly more difficult to deal with.

Software Construction

The world of software design and implementation seems to be particularly prone to the complexities described in previous sections. The design space of software is so large and flexible that it allows a make-your-own-physics form of design: all layers of the model space are under design control, unlike the physical world where you can’t, say, change the charge on an electron.

Given this property, it makes sense that we find domain corruption, functional differences, and violation of layering wherever we look. These are forms of chaos: they induce model jumps in the minds of users and developers with varying degrees of sensitivity and wandering.

It also helps to explain the great difficulty we have predicting the path of development of software systems, and how often we find major issues that can delay a project right up to its end. When debugging, for example, if one finds a problem at an upper layer of an application abstraction, diagnosis and repair usually proceed pretty quickly. However, if one needs to dive down to very low levels to find a problem, such as to the machine-instruction level, then prediction as to when the problem will be solved often becomes impossible.

The Absurd Behavior of Software Systems

Model jumps such as those described previously often produce behavior that is bizarre in some way. Particularly strong in the model-jumping space are violations of layering characterized by relatively short model skips and either a large or small time to violation. In either case we are often led to define this behavior as an *absurdity*, as developed above under *Philosophical Considerations*.

Examples of this sort of absurdity in software systems are easy to find:

- Saving a file in a word processor is successful, but later when you look for it, it’s mysteriously vanished.
- An innocent operation like copying and pasting text pops up an unintelligible error window, filled with hexadecimal numbers and stack jargon (illustrated by Figure 9).
- For weeks, printing from your web browser has used the printer across the hall. Today, suddenly, it’s using the one at the other end of the building.

Note that these examples are characterized by models that are relative close semantically with different times to a jump between them.

Other kinds of model jump patterns could lead us to classify further the kinds of bizarre behavior we can expect from software, for example when you think there is no way to fix your computer other than using chants and exorcising ceremonies. Virtual machines, for example, introduce an operating layer whose anomalous behavior can sometimes be considered surreal, as for instance when it acts odd with respect to time.

The reader is encouraged to think of other examples of software absurdity and similarly strange behavior, and tie it to a more formal model structure similar to those described here.

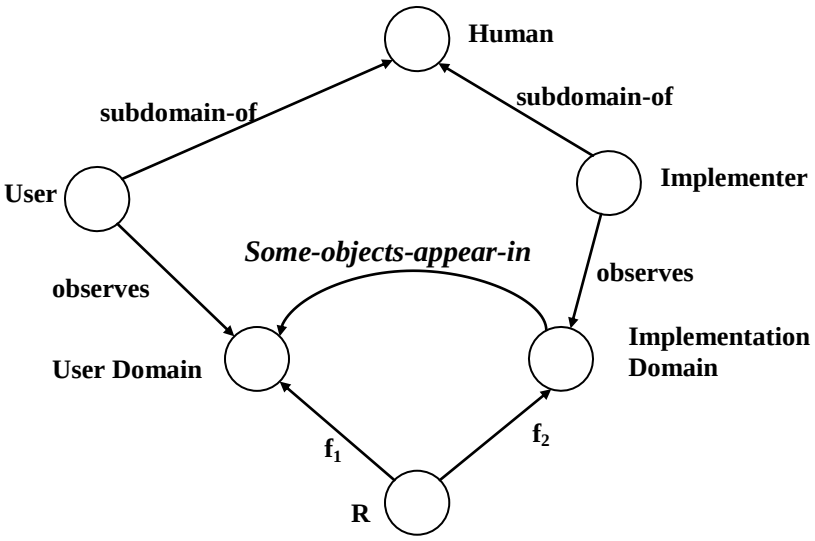


Figure 9 – A simple software absurdity.

Conclusion

The idea of unpredictability in chaos theory depends in turn on the idea of exponential divergence of trajectories, which further requires a metric space for its definition. When an obvious metric space does not present itself, we turn to more general concepts of divergence to gauge whether or not some phenomenon is predictable. It has been argued in this paper that discrete jumping among models provides a good foundation for such non-metric divergence and a way to ground ideas of unpredictability.

We have illustrated this concept by pointing to various examples, such as finding absurdity in software, models of dreams and surreality, surprises such as action at a distance, and the effect of low-level events on high-level abstractions, as may be found in fundamental physical models. We have generalized some of these ideas into the intuitively appealing concept of model layers. These ideas have been tied together using a simple unified computational framework which emphasizes structural simplicity at all levels and the fundamental use of meta-information.

In the software realm, we might find application of these ideas in evaluating software complexity. Using static analysis techniques, we can attempt to infer the model structure and judge its quality with respect to, for instance, how well-layered the model appears, and subsequently use that analysis to attempt to predict where model jumps and failures are likely to occur. This analysis need not be completely automated, and a set of anti-patterns might be developed that captures modeling concepts such as those described here.

The idea that software can behave in absurd ways, and in other manners consistent with humanistic concepts such as a seemingly supernatural action at a distance, also has intuitive appeal, and can serve as a set of concepts to bridge formal descriptions of software behavior and their more informal and colloquial counterparts.

More broadly, we can consider notations of various kinds for composing models, to cover situations consisting of combinations of model types and possible jumps between them, to characterize qualitative chaotic behavior for more complex systems than given in the examples here. Compositional notations for system descriptions are of course common, and there is range of styles and languages that would likely be suitable for such a “qualitative chaos calculus”.

References

1. Kathleen T. Alligood, Tim D. Sauer, James A. Yorke, *Chaos: An Introduction to Dynamical Systems*, Springer, 1996.
2. Saber N. Elaydi, *Discrete Chaos, Second Edition*, Chapman and Hall/CRC, 2008.
3. Herbert Simon, Architecture of Complexity, *Proceedings of the American Philosophical Society*, Vol. 106, No. 6. (Dec. 12, 1962), pp. 467-482.
4. Tien-Yien Li and James A. Yorke, Period Three Implies Chaos, The American Mathematical Monthly, Vol. 82, No. 10. (Dec., 1975), pp. 985-992.
5. Robert Devaney, Chaotic Dynamical Systems, second edition, Westview Press, 2003.
6. Daniel B. Miller and Edward Fredkin, Two-state, Reversible, Universal Cellular Automata In Three Dimensions, 2005, http://www.digitalphilosophy.org.
7. Randall Davis and Douglas B. Lenat, Knowledge-Based Systems in Artificial Intelligence, McGraw-Hill, 1982.
8. L. Stabile, The H-Machine, in preparation.
9. Stephen Wolfram, A New Kind of Science, Wolfram Media, 2002.
10. Robert Gilmore, http://einstein.drexel.edu/~bob/PHYS750_NLD/ch2.pdf.
11. Oscar Nierstrasz, Meta-levels, http://scg.unibe.ch/download/mm/Slides/01Intro.ppt.pdf
12. Gerald Jay Sussman, Jack Wisdom, Numerical Evidence that the Motion of Pluto is Chaotic, http://groups.csail.mit.edu/mac/users/wisdom/pluto-chaos.pdf, 1988
13. Thompson, An evolved circuit, intrinsic in silicon, entwined with physics, 1996, http://www.sussex.ac.uk/Users/adrianth/ices96/paper.html.
14. A. Fotiou, Deterministic Chaos, Msc Project, University of London, 2005, http://www.mpipks-dresden.mpg.de/~rklages/people/msc_fotiou.pdf.
15. Category Error, http://en.wikipedia.org/wiki/Category_error