Building Robust Systems
an essay

Gerald Jay Sussman
Massachusetts Institute of Technology

February 23, 2008

Abstract

It is hard to build robust systems: systems that have acceptable behavior over a larger class of situations than was anticipated by their designers. The most robust systems are evolvable: they can be easily adapted to new situations with only minor modification. How can we design systems that are flexible in this way?

Observations of biological systems tell us a great deal about how to make robust and evolvable systems. Techniques originally developed in support of symbolic Artificial Intelligence can be viewed as ways of enhancing robustness and evolvability in programs and other engineered systems. By contrast, common practice of computer science actively discourages the construction of robust systems.

Robustness

It is difficult to design a mechanism of general utility that does any particular job very well, so most engineered systems are designed to perform a specific job. General-purpose inventions, such as the screw fastener, when they appear, are of great significance. The digital computer is a breakthrough of this kind, because it is a universal machine that can simulate any other information-processing machine. We write software that configures our computers to effect this simulation for the specific jobs that we need done.

We have been designing our software to do particular jobs very well, as an extension of our past engineering practice. Each piece of software is designed to do a relatively narrow job. As the problem to be solved changes, the software must be changed. But small changes to the problem to be solved do not entail only small changes to the software. Software is designed too tightly for there to be much flexibility. As a consequence, systems do not evolve gracefully. They are brittle and must be replaced with entirely new designs as the problem domain evolves.[1] This is slow and expensive.

Our engineered systems do not have to be brittle. The Internet has adapted from a small system to one of global scale. Our cities evolve organically, to accommodate new business models, life styles, and means of transportation and communication. Indeed, from observation of biological systems we see that it is possible to build systems that can adapt to changes in the environment, both individually and as an evolutionary ensemble. Why is this not the way we design and build most software? There are historical reasons, but the main reason is that we don't know how to do this generally. At this moment it is an accident if a system turns out to be robust to changes in requirements.

Redundancy and degeneracy

Biological systems have evolved a great deal of robustness. One of the characteristics of biological systems is that they are redundant. Organs such as the liver and kidney are highly redundant: there is vastly more capacity

[1]Of course, there are some wonderful exceptions. For example, `EMACS` [36] is an extensible editor that has evolved gracefully to adapt to changes in the computing environment and to changes in its user's expectations. The computing world is just beginning to explore "engineered frameworks," for example, Microsoft's `.net` and Sun's `Java`. These are intended to be infrastructures to support evolvable systems.

than is necessary to do the job, so a person with a missing kidney suffers no obvious incapacity. Biological systems are also highly degenerate: there are usually many ways to satisfy a given requirement.[2] For example, if a finger is damaged, there are ways that the other fingers may be configured to pick up an object. We can obtain the necessary energy for life from a great variety of sources: we can metabolize carbohydrates, fats, and proteins, even though the mechanisms for digestion and for extraction of energy from each of these sources is quite distinct.

The genetic code is itself degenerate, in that the map from codons (triples of nucleotides) to amino acids is not one-to-one: there are 64 possible codons to specify only about 20 possible amino acids. [28] As a consequence, many point mutations do not change the protein specified by a coding region. Also, quite often the substitution of one amino acid with a similar one does not impair the biolological activity of a protein. These provide ways that variation can accumulate without obvious phenotypic consequences. Furthermore, if a gene is duplicated (not an uncommon occurrence), the copies may diverge silently, allowing the development of variants that may become valuable in the future, without interfering with current viability. In addition, the copies can be placed under different transcriptional controls.

Degeneracy is a product of evolution, and it certainly enables evolution. Probably degeneracy is itself selected for because only creatures that have significant amounts of degeneracy are sufficiently adaptable to allow survival as the environment changes.[3] For example, suppose we have some creature (or engineered system) that is degenerate in that there are several very different interdependent mechanisms to achieve each essential function. If the environment changes (or the requirements change) so that one of the ways of achieving an essential function becomes untenable, the creature will continue to live and reproduce (the system will continue to satisfy its specifications). But the subsystem that has become inoperative is now open to mutation (or repair), without impinging on the viability (or current operation) of the system as a whole.

Engineered systems may incorporate some redundancy, in critical systems where the cost of failure is extreme. But they almost never intentionally incorporate degeneracy of the kind found in biological systems, except as a

[2]Although clear in extreme cases, the distinction biologists make between redundancy and degeneracy is fuzzy at the boundary. For more information see [13].

[3]Some computer scientists have investigated the evolution of evolvability by simulation. [2]

side effect of designs that are not optimal.[4]

Exploratory Behavior

One of the most powerful mechanisms of robustness in biological systems is exploratory behavior.[5] The idea is that the desired outcome is produced by a generate-and-test mechanism. This organization allows the generator mechanism to be general and to work independently of the testing mechanism that accepts or rejects a particular generated result.

For example, an important component of the rigid skeleton that supports the shape of a cell is an array of microtubules. Each microtubule is made up of protein units that aggregate to form the microtubule. Microtubules are continually created and destroyed in a living cell; they are created growing out in all directions. However, only microtubules that encounter a stabilizer in the cell membrane are stable, thus supporting the shape determined by the positions of the stabilizers. So the mechanism for growing and maintaining a shape is relatively independent of the mechanism for specifying the shape. This mechanism partly determines the shapes of many types of cells in a complex organism, and it is almost universal in metazoans.

Exploratory behavior appears at all levels of detail in biological systems. The nervous system of a growing embryo produces a vastly larger number of neurons than will persist in the adult. Those neurons that find appropriate targets in other neurons, sensory organs, or muscles will survive and those that find no targets kill themselves. The hand is fashioned by production of a pad and deletion, by apoptosis, of the material between the fingers. [45] Our bones are continually being remodeled by osteoblasts (which build bone) and osteoclasts (which destroy bone). The shape and size of the bones is determined by constraints determined by their environment: the parts that they must be associated with, such as muscles, ligaments, tendons, and other bones.

[4]Indeed, one often hears arguments against building flexibility into an engineered system. For example, in the philosophy of the computer language `Python` it is claimed: “There should be one—and preferably only one—obvious way to do it.”[35] Science does not usually proceed this way: In classical mechanics, for example, one can construct equations of motion using Newtonian vectoral mechanics, or using a Lagrangian or Hamiltonian variational formulation.[40] In the cases where all three approaches are applicable they are equivalent, but each has its advantages in particular contexts.

[5]This thesis is nicely explored in the book of Kirschner and Gerhart.[26]

Because the generator need not know about how the tester accepts or rejects its proposals, and the tester need not know how the generator makes its proposals, the two parts can be independently developed. This makes adaptation and evolution more efficient, because a mutation to one or the other of these two subsystems need not be accompanied by a complementary mutation to the other. However, this isolation is expensive because of the wasted effort of generation and rejection of failed proposals.

Indeed, generate and test is a metaphor for all of evolution. The mechanisms of biological variation are random mutations: modifications of the genetic instructions. Most mutations are neutral in that they do not directly affect fitness because of degeneracy in the systems. Natural selection is the test phase. It does not depend on the method of variation, and the method of variation does not anticipate the effect of selection.

There are even more striking phenomena: even in closely related creatures some components that end up almost identical in the adult are constructed by entirely different mechanisms in the embryo.[6] For distant relationships divergent mechanisms for constructing common structures may be attributed to "convergent evolution," but for close relatives it is more likely evidence for separation of levels of detail, in which the result is specified in a way that is somewhat independent of the way it is accomplished.

Compartments and localization

There is ample evidence for modular construction based on regional differentiation in biology. We are constructed from a hierarchy of organ systems, organs, and tissues. All vertebrates have essentially the same body plan, yet the variation in details is enormous. The morphogenesis of ducted glands, and organs such as lungs and kidneys is based on one embryological trick: the invagination of epithelium into mesenchyme automagically produces a branching maze of blind-end tubules surrounded by differentiating

[6]The cornea of a chick and the cornea of a mouse are almost identical, but the morphogenesis of these two are not at all similar: the order of the morphogenetic events is not even the same. Bard [5] section 3.6.1 reports that having divergent methods of forming the same structures in different species is common. He quotes a number of examples. One spectacular case is that the frog *Gastrotheca riobambae* (recently discovered by delPino and Elinson [12]) develops ordinary frog morphology from an embryonic disk, whereas other frogs develop from an approximately spherical embryo.

mesenchyme.[7]

Every cell in our bodies is a descendant of a single zygote. All the cells have exactly the same genetic endowment (about 1GByte of ROM!). However there are skin cells, neurons, muscle cells, etc. The cells organize themselves to be discrete tissues, organs, and organ systems. This is possible because the way a cell differentiates and specializes depends on its environment. Almost all metazoans share homeobox genes, such as the Hox complex. Such genes produce an approximate coordinate system in the developing animal, separating the developing animal into distinct locales.[8] The locales provide context for a cell to differentiate. And information derived from contact with its neighbors produces more context that selects particular behaviors from the possible behaviors that are available in its genetic program.[9]

This kind of organization has certain clear advantages. Flexibility is enhanced by the fact that the signaling among cells is permissive rather than instructive.[10] That is, the behaviors of cells are not encoded in the signals; they are separately expressed in the genome. Combinations of signals just enable some behaviors and disable others. This weak linkage allows variation in the implementation of the behaviors that are enabled in various locales without modification of the mechanism that defines the locales. So systems organized in this way are evolvable in that they can accomodate adaptive variation in some locales without changing the behavior of subsystems in other locales.

Good engineering has a similar flavor, in that good designs are modular. Consider the design of a radio receiver. There are several grand "body plans" that have been discovered, such as direct conversion, TRF (tuned radio frequency), and superheterodyne. Each has a sequence of locales, defined by the engineering equivalent of a Hox complex, that patterns the system from the antenna to the output transducer. For example, a superheterodyne receiver has the following locales (rostral to caudal):

---

[7]One well-studied example of this kind of mechanism is the formation of the submandibular gland of the mouse. See, for example, the treatment in [8] or the summary in [5] section 3.4.3.

[8]This is a very vague description of a complex process involving gradients of morphogens. I do not intend to get more precise here, as this is not a paper about biology, but rather about how biology informs engineering.

[9]We have investigated some of the programming issues involved in this kind of development in our Amorphous Computing project.[3]

[10]Kirschner and Gerhart examine this too.[26]

Antenna : RF : Converter : IF : Detector : AF : Transducer

Each locale can be instantiated in many possible ways: the RF section may be just a filter, or it may be an elaborate filter and amplifier combination. The detector section may be instantiated as an AM, SSB, or FM detector. The AF amplifier section may be mutated to operate on video signals and the output transducer may be a CRT. Some sections may be recursively elaborated (as if parts of the Hox complex were duplicated!) to obtain multiple-conversion receivers.

Unlike biological mechanisms, in analog electronics the components are usually not universal in that each component can, in principle, act as any other component. But there are universal electrical building blocks (a programmable computer with analog interfaces for example!).[11] For low-frequency applications one can build analog systems from such blocks. If each block had all of the code required to be any block in the system, but was specialized by interactions with its neighbors, and if there were extra unspecialized "stem cells" in the package, then we could imagine building self-reconfiguring and self-repairing analog systems.

The structure of compartments in biological systems is supported by an elaborate infrastructure. One important component of this infrastructure is the ability to dynamically attach tags to materials being manipulated. In eukaryotic cells, proteins are constructed with tags describing their destinations. [28] For example, a transmembrane protein, which may be part of an ion-transport pore, is directed to the plasma membrane, whereas some other protein might be directed to the golgi apparatus. There are mechanisms in the cell (themselves made up of assemblies of proteins) to recognize these tags and effect the transport of the parts to their destinations. Tagging is also used to clean up and dispose of various wastes, such as protein molecules that did not fold correctly or that are no longer needed. Such proteins are tagged (ubiquinated) and carried to a proteasome for degradation.

This structure of compartments is also supported at higher levels of organization. There are tissues that are specialized to become boundaries of compartments, and tubes that interconnect them. Organs are bounded by such tissues and interconnected by such tubes, and the entire structure is packaged to fit into coelems, which are cavities lined with specialized tissues in higher organisms.

[11] Piotr Mitros has recently developed a novel design strategy for building analog circuits from potentially universal building blocks. See [30].

Defense, repair, and regeneration

Biological systems are always under attack from predators, parasites, and invaders. They have developed elaborate systems of defense, ranging from restriction enzymes[12] in bacteria, the immune systems of mammals, all the way to horns and the production of noxious fumes. Advanced systems depend on continuous monitoring of the external and internal environment, and mechanisms for distinguishing self from other.

In a complex organism derived from a single zygote every cell is, in principle, able to perform the functions of every other cell. Thus there is redundancy in numbers. But even more important is the fact that this provides a mechanism for repair and regeneration. A complex organism is a dynamically reconfigurable structure made out of potentially universal interchangeable and reproducible parts: if a part is damaged, nearby cells can retarget to fill in the gap and take on the function of the damaged part.

The computer software industry has only recently begun to understand the threats from predators, parasites, and invaders. The early software systems were built to work in friendly, safe environments. But with the globalization of the network and the development of economic strategies that depend on attacking and coopting vulnerable systems, the environment has changed to a substantially hostile one. Current defenses, such as antivirus and antispam software, are barely effective in this environment (although there are significant attempts to develop biologically-inspired computer "immune systems").

One serious problem is monoculture. Almost everyone uses the same computer systems, greatly increasing the vulnerability. In biological systems there are giant neutral spaces, allowing great variation with negligible change in function: humans can have one of several blood types; there are humans of different sizes, shapes, colors, etc. But they are all human. They all have similar capabilities and can all live in a great variety of environments. They communicate with language! However, not all humans have the same vulnerabilities: people heterozygous for the sickle-cell trait have some resistance to

[12]A restriction enzyme cuts DNA molecules at particular sites that are not part of the genome of the bacterium, thus providing some defense against viruses that may contain such sites in their genome. One could imagine an analogous computational engine that stops any computation containing a sequence of instructions that does not occur in the code of the operating system. Of course, a deliberately constructed virus (biological or computational) may be designed to elude any such simple restriction-enzyme structure.

malaria.

In our engineered systems we have not, in general, taken advantage of this kind of diversity. We have not yet made use of the diversity that is available in alternate designs, or even used the variation that is available in silent mutations. Part of the reason is that there is economy in monoculture. But this economy is short-sighted and illusory, because of the extreme vulnerability of monoculture to deliberate and evolved attack.[13] Biological systems have substantial abilities to repair damage, and, in some cases, to regenerate lost parts. This ability requires extensive and continuous self-monitoring, to notice the occurrence of damage and initiate a repair process. It requires the ability to mobilize resources for repair and it requires the information about how the repair is to be effected.

Systems that build structure using exploratory behavior can easily be coopted to suppport repair and regeneration.[14] However, it is still necessary to control the exploratory proliferation to achieve the desired end state. This appears to be arranged with homeostatic constraint mechanisms. For example, a wound may require the production of new tissue to replace the lost material. The new tissue needs to be supplied with oxygen and nutrients, and it needs wastes removed. Thus it must be provided with new capillaries that correctly interconnect with the circulatory system. Cells that do not get enough oxygen produce hormones that stimulate the proliferation of blood vessels in their direction. Thus, the mechanisms that build the circulatory system need not know the geometry of the target tissues. Their critical mission is achieved by exploration and local constraint satisfaction. Such mechanisms support both morphogenesis in the embryo and healing in the adult.[15]

There are very few engineered systems that have substantial ability for self-repair and regeneration. High-quality operating systems have "file-system salvagers" that check the integrity of the file system and use redundancy in the file system to repair broken structures and to regenerate some lost parts. But this is an exceptional case. How can we make this kind of self-monitoring

[13]This point was made quite strongly a while ago by Stefanie Forrest and friends. See [17].

[14]Lauren Clement, Jake Beal, and Radhika Nagpal have explored this part of the space. [11, 7]

[15]Regeneration in Urodele amphibians seems to be controlled locally by the "law of normal neighbors", as is the patterning of the cuticle of ciliates. This is a striking appearance of simplicity in an otherwise complex situation. [9, 18]

and self-repair the rule rather than the exception?

In both cases, defense and repair, a key component is awareness—the ability to monitor the environment for imminent threats and one's self for damage.

Composition

Large systems are composed of many smaller components, each of which contributes to the function of the whole either by directly providing a part of that function or by cooperating with other components by being interconnected in some pattern specified by the system architect to establish a required function. A central problem in system engineering is the establishment of interfaces that allow the interconnection of components so that the functions of those components can be combined to build compound functions.

For relatively simple systems the system architect may make formal specifications for the various interfaces that must be satisfied by the implementers of the components to be interconnected. Indeed, the amazing success of electronics is based on the fact that it is feasible to make such specifications and to meet them. High-frequency analog equipment is interconnected with coaxial cable with standardized impedance characteristics, and with standardized families of connectors. [4] Both the function of a component and its interface behavior can usually be specified with only a few parameters. [21] In digital systems things are even more clear. There are static specifications of the meanings of signals (the digital abstraction). There are dynamic specifications of the timing of signals. [42] And there are mechanical specifications of the form-factors of components.[16]

Unfortunately, this kind of *a priori* specification becomes progressively

[16] *The TTL Data Book for Design Engineers* [41] is a classic example of a successful set of specifications for digital-system components. It specifies several internally consistent "families" of small-scale and medium-scale integrated-circuit components. The families differ in such characteristics as speed and power dissipation, but not in function. The specification describes the static and dynamic characteristics of each family, the functions available in each family, and the physical packaging for the components. The families are cross consistent as well as internally consistent in that each function is available in each family, with the same packaging and a consistent nomenclature for description. Thus a designer may design a compound function and later choose the family for implementation. Every good engineer (and biologist!) should be familiar with the lessons in the TTL Data Book.

more difficult as the complexity of the system increases.[17] The specifications of computer software components are often enormously complicated. They are difficult to construct and it is even more difficult to guarantee compliance with such a specification. Many of the fragilities associated with software are due to this complexity.

By contrast, biology constructs systems of enormous complexity without very large specifications. The human genome is about 1 GByte. This is vastly smaller than the specifications of a major computer operating system. How could this possibly work? We know that the various components of the brain are hooked together with enormous bundles of neurons, and there is nowhere near enough information in the genome to specify that interconnect in any detail. What is likely is that the various parts of the brain learn to communicate with each other, based on the fact that they share important experiences. So the interfaces must be self-configuring, based on some rules of consistency, information from the environment, and extensive exploratory behavior. This is pretty expensive in boot-up time (it takes some years to configure a working human) but it provides a kind of robustness that is not found in our engineered entities to date.

The Cost

"To the optimist, the glass is half full. To the pessimist, the glass is half empty. To the engineer, the glass is twice as big as it needs to be."

author unknown

Unfortunately, generality and evolvability seem to require redundancy, degeneracy, and exploratory behavior. These are expensive, when looked at in isolation. A mechanism that works over a wide range of inputs must do more to get the same result as a mechanism specialized to a particular input. A redundant mechanism has more parts than an equivalent non-redundant mechanism. A degenerate mechanism appears even more extravagant. Yet these are ingredients in evolvable systems. To make truly robust systems we must be willing to pay for what appears to be a rather elaborate infrastructure. The value, in enhanced adaptability, may be even more extreme.

[17]We could specify that a chess-playing program plays a legal game—that it doesn't cheat, but how would one begin to specify that it plays a good game of chess?

Indeed, the cost of our brittle infrastructure probably greatly exceeds the cost of robust design, both in the cost of disasters and in the lost opportunity costs due to the time of redesign and rebuilding.

The problem with correctness

But there may be an even bigger cost to building systems in a way that gives them a range of applicablity greater than the set of situations that we have considered at design time. Because we intend to be willing to apply our systems in contexts for which they were not designed, we cannot be sure that they work correctly!

In software engineering we are taught that the "correctness" of software is paramount, and that correctness is to be achieved by establishing formal specification of components and systems of components and by providing proofs that the specifications of a combination of components are met by the specifications of the components and the pattern by which they are combined. I assert that this discipline enhances the brittleness of systems. In fact, to make truly robust systems we must discard such a tight discipline.

The problem with requiring proofs is that it is usually harder to prove general properties of general mechanisms than it is to prove special properties of special mechanisms used in constrained circumstances. This encourages us to make our parts and combinations as special as possible so we can simplify our proofs.

I am not arguing against proofs. They are wonderful when available. Indeed, they are essential for critical system components, such as garbage collectors (or ribosomes!). However, even for safety-critical systems, such as autopilots, the restriction of applicability to situations for which the system is provably correct as specified may actually contribute to unnecessary failure. Indeed, we want an autopilot to make a good-faith attempt to safely fly an airplane that is damaged in a way not anticipated by the designer!

I am arguing against the discipline of *requiring* proofs: The requirement that everything must be proved to be applicable in a situation before it is allowed to be used in that situation excessively inhibits the use of techniques that could enhance the robustness of designs. This is especially true of techniques that allow a method to be used, on a tight leash, outside of its proven domain, and techniques that provide for future expansion without putting limits on the ways things can be extended.

Unfortunately, many of the techniques I advocate make the problem of proof much more difficult, if not practically impossible. On the other hand, sometimes the best way to attack a problem is to generalize it until the proof becomes simple.

Architecture of Computation

A metaphor from architecture may be illuminating for the kind of system that we contemplate. After understanding the nature of the site to be constructed and the requirements for the structure, the design process starts with a parti: the discovery of an organizing principle for the design.[18] The parti is usually a sketch of the geometric arrangement of parts. The parti may also embody abstract ideas, such as the division into "served spaces" and "servant spaces," as in the work of Louis Isadore Kahn. [44] This decomposition is intended to divide the architectural problem into parts by separating out infrastructural support, such as the hallways, the restrooms, the mechanical rooms, the elevators, etc. from the spaces to be supported, such as the laboratories, classrooms, and offices in an academic building.

The parti is a model, but it is usually not a completely workable structure. It must be elaborated with functional elements. How do we fit in the staircases and elevators? Where do the HVAC ducts, the plumbing, the electrical and communications distribution systems go? How will we run a road to accommodate the delivery patterns of service vehicles? These elaborations may cause modifications of the parti, but the parti continues to serve as a scaffold around which these elaborations are developed.

In programming, the parti is the abstract plan for the computations to be performed. At small scale the parti may be an abstract algorithm and data-structure description. In larger sections it is an abstract composition of phases and parallel branches of a computation. In even larger systems it is an allocation of capabilities to logical (or even physical) locales.

Traditionally, programmmers have not been able to design as architects. In very elaborate languages, such as Java, the parti is tightly mixed with the elaborations. The "served spaces," the expressions that actually describe the desired behavior, are horribly conflated with the "servant spaces," such as the type declarations, the class declarations, and the library imports and exports. More spare languages, such as Lisp or Python, leave almost no room

[18]A *parti* is the central idea of an architectural work: it is "the [architectural] composition being conceived as a whole, with the detail being filled in later." [22]

for the servant spaces, and attempts to add declarations, even advisory ones, are shunned because they impede the beauty of the exposed parti.

A programmer should be able to lay down an untyped spare plan, very much like a circuit diagram, with the terminals of parts interconnected with nodes. Some of the primitive parts may be relations, but it will be convenient to represent functions and parts with independent local state. The wiring-diagram language must include means of abstraction so that diagrams with exposed terminals may be given names and referred to as primitive objects. Of course, for the parts of programs that can be expressed conveniently with expressions, expression syntax should be provided so the programmer will not have to express everything as explicit wiring.

The spare plan must be executable. However, we expect that programmers will want to elaborate their plans. Some parts of a system may benefit from elaborations, but they may not be necessary to impose on all parts of a system.

Declarations of types or of units and dimensions, may allow early detection of bugs. Uncertainty of some values may be probabilistically tracked by decorating them with probability distributions. Specification of representations may allow a compiler to produce more efficient code. Dependency tracking and analysis can enable programs to report on provenance, in order to defend their conclusions and simplify debugging. A truth-maintenance system can enable the simultaneous consideration of mutually inconsistent viewpoints, and can organize the optimization of combinatorial search processes.

Infrastructure to Support Generalizability

We want to build systems that can be easily generalized beyond their initial use. Let's consider techniques that can be applied in software design. I am not advocating a grand scheme or language, such as `PLANNER`, but rather infrastructure for integrating each of these techniques when appropriate.

Generality of parts

The most robust systems are built out of families of parts, where each part is of very general applicability. Such parts have acceptable behavior over a much wider class of conditions than is needed for any particular application.

If, for example, we have parts that produce outputs for given inputs, we need the range of acceptable inputs for which the outputs are sensible to be very broad. To use a topological metaphor, the class of acceptable inputs for any component used in a solution to a set of problems should be an "open set" surrounding the inputs it will encounter in actual use in the current set of problems.

Furthermore, the range of outputs of a part over this wide range of inputs should be quite small and well defined: much smaller than the range of acceptable inputs for any part that might receive that output. This is analogous to the static discipline in the digital abstraction that we teach to students in introductory computer systems subjects.[42] The essence of the digital abstraction is that the outputs are always better than the acceptable inputs, so noise is suppressed.

Using more general parts than appear to be necessary builds a degree of flexibility into the entire structure of our systems. Small perturbations of the requirements can be adjusted to without disaster, because every component is built to accept perturbed (noisy) inputs.

There are a variety of techniques to help us make families of components that are more general than we anticipate needing for the applications under consideration at the time of the design of the components. These techniques are not new. They are commonly used, often unconsciously, to help us conquer some particular problem. However, we have no unified understanding or common infrastructure to support their use. Furthermore, we have developed a culture that considers many of these techniques dangerous or dirty. It is my intention to expose these techniques in a unified context so we can learn to exploit them to make more robust systems.

Extensible generic operations

One good idea is to build a system on a substrate of extensible generic operators. Modern dynamically typed programming languages usually have built-in arithmetic that is generic over a variety of types of numerical quantities, such as integers, floats, rationals, and complex numbers.[38, 23, 34] This is already an advantage, but it surely complicates reasoning and proofs and it makes the implementation much more complicated and somewhat less efficient than simpler systems. However, I am considering an even more general scheme, where it is possible to define what is meant by addition, multiplication, etc., for new datatypes unimagined by the language designer. Thus, for example, if the arithmetic operators of a system are extensible generics, a user may extend them to allow arithmetic to be extended to quaternions, vectors, matrices, integers modulo a prime, functions, tensors, differential forms, ... This is not just making new capabilities possible; it also extends old programs, so a program that was written to manipulate simple numerical quantities may become useful for manipulating scalar-valued functions.[19] However, there is a risk. A program that depends on the commutativity of numerical multiplication will certainly not work correctly for matrices. (Of course, a program that depends on the exactness of operations on integers will not work correctly for inexact floating-point numbers either.) This is exactly the risk that comes with evolution—some mutations will be fatal! But that risk must be balanced against the cost of narrow and brittle construction.

On the other hand, some mutations will be extremely valuable. For example, it is possible to extend arithmetic to symbolic quantities. The simplest way to do this is to make a generic extension to all of the operators to take symbolic quantities as arguments and return a data structure representing the indicated operation on the arguments. With the addition of a simplifier of algebraic expressions we suddenly have a symbolic manipulator. This is very useful in debugging purely numerical calculations, because if they are given symbolic arguments we can examine the resulting symbolic expressions to make sure that the program is calculating what we intend it to. It is also the basis of a partial evaluator for optimization of numerical programs. And functional differentiation can be viewed as a generic extension of arithmetic to a hyperreal datatype.[20]

Extensible generic operations are not for the faint of heart. The ability to extend operators *after the fact* gives both extreme flexibility and ways to make whole new classes of bugs! It is probably impossible to prove very much about a program when the primitive operations can be extended, except that it will work when restricted to the types it was defined for. This is an easy but dangerous path for generalization.

[19]A mechanism of this sort is implicit in most "object-oriented languages," but it is usually buried in the details of ontological mechanisms such as inheritance. The essential idea of extensible generics appears in SICP [1] and is usefully provided in tinyCLOS [24] and SOS [19]. A system of extensible generics, based on predicate dispatching, is used to implement the mathematical representation system in SICM [40]. A nice exposition of predicate dispatching is given by Ernst [14].

[20]The `scmutils` system we use to teach classical mechanics [40] implements differentiation in exactly this way.

Extensible generic operations, and the interoperation of interpreted and compiled code, are most easily accomplished using tagged data representations. The data is tagged with the information required to decide which procedures are to be used for implementing the indicated operations. But once we have the ability to tag data there are other uses tags can be put to. For example, we may tag data with its provenance, or how it was derived, or the assumptions it was based on. Such audit trails may be essential for access control, for tracing the use of sensitive data, or for debugging complex systems.[43] Thus we can get power by being able to attach arbitrary tags to any data item, besides the tags used for determining generics.

Generate and test

We normally think of generate and test, and its extreme use in search, as an AI technique. However, it can be viewed as a way of making systems that are modular and independently evolvable, as in the exploratory behavior of biological systems. Consider a very simple example: suppose we have to solve a quadratic equation. There are two roots to a quadratic. We could return both, and assume that the user of the solution knows how to deal with that, or we could return one and hope for the best. (The canonical SQRT routine returns the positive square root, even though there are two square roots!) The disadvantage of returning both solutions is that the receiver of that result must know to try his computation with both and either reject one, for good reason, or return both results of his computation, which may itself have made some choices. The disadvantage of returning only one solution is that it may not be the right one for the receiver's purpose.

A better way to handle this is to build a backtracking mechanism into the infrastructure.[15, 20, 31, 1] The square-root procedure should return one of the roots, with the option to change its mind and return the other one if the first choice is determined to be inappropriate by the receiver. It is, and should be, the receiver's responsibility to determine if the ingredients to its computation are appropriate and acceptable. This may itself require a complex computation, involving choices whose consequences may not be apparent without further computation, so the process is recursive. Of course, this gets us into potentially deadly exponential searches through all possible assignments to all the choices that have been made in the program. As usual, modular flexibility can be dangerous.

But if the choice mechanism attaches a tag describing its state to the data it selects, and if the primitive operations that combine data combine these tags correctly, one can always tell which choices contributed to any particular piece of data. With such a system, search can be optimized so that only relevant choices must be considered in any particular backtrack. This is the essence of dependency-directed backtracking. [37, 16, 29, 39] If such a system is built into the infrastructure then exploratory behavior can be as efficient as any explicit manipulation of sets of choices, without a program having to know which contributors of data to its inputs are actually sets of possibilities. It does, however, incur the overhead of a program testing for consistency of its results and rejecting them if necessary. Of course, this is important in any system intended to be reliable as well as robust.

Constraints generalize procedures

Consider an explicit integrator for a system of ordinary differential equations, such as the Bulirsch-Stoer algorithm.[10, 33] The description of the ODEs for such an integrator is a system-derivative procedure that takes a state of the system and gives back the derivative of the state. For example, the system derivative for a driven harmonic oscillator takes a structure with components the time, the position, and the velocity and returns a vector of 1 (dt/dt), the velocity, and the acceleration. The system derivative has three parameters: the damping constant, the square of the undamped oscillatory frequency, and the drive function. The natural frequencies are determined by a quadratic in the first two parameters. The sum of the natural frequencies is the damping constant and the product of the natural frequencies is the square of the undamped oscillatory frequency. We can also define a Q for such a system. In any particular physical system, such as a series-resonant circuit, there are relationships between these parameters and the inductance, the capacitance, and the resistance of the circuit. Indeed, one may specify the system derivative in many ways, such as the oscillatory frequency, the capacitance, and the Q. There is no reason why this should be any harder than specifying the inductance, the resistance, and the capacitance.

If we have a set of quantities and relations among them we can build a constraint network that will automatically derive some of the quantities given the values of others. By including the parameters of the system derivative in a constraint network we greatly increase the generality of its application without any loss of efficiency for numerical integration. An added benefit is that we can use the constraint-propagation process to give us multiple alternative views of the mechanism being simulated: we can attach a spring, mass, and dashpot constraint network to our series RLC constraint network and think about the inertia of the current in our inductor. The infrastructure needed to support such a constraint-directed invocation mechanism is inexpensive, and the truth-maintenance system needed to track the dependencies is the same mechanism needed to implement the dependency-directed backtracking described above.

But constraints give us more than a support for generality. Constraints that dynamically test the integrity of data structures and hardware can be used to notice and signal damage. Such mechanisms may be able to encapsulate damage so that it does not spread, and trigger defense mechanisms to fight the damaging agent. Also, if we make systems that build themselves using generate-and-test mechanisms controlled by constraints that enforce requirements on the structure of the result we can build systems that can repair some forms of damage automatically.

Degeneracy in engineering

In the design of any significant system there are many implementation plans proposed for every component at every level of detail. However, in the system that is finally delivered this diversity of plans is lost and usually only one unified plan is adopted and implemented. As in an ecological system, the loss of diversity in the traditional engineering process has serious consequences.

We rarely build degeneracy into programs, partly because it is expensive and partly because we have no formal mechanisms for mediating its use. However, there is a mechanism from the AI problem-solving world for degenerate designs: goal-directed invocation. The idea is that instead of specifying "how" we want a goal accomplished, by naming a procedure to accomplish it, we specify "what" we want to accomplish, and we link procedures that can accomplish that goal with the goal. This linkage is often done with pattern matching, but that is accidental rather than essential.[21] If there is more than one way to accomplish the goal, then the choice of an appropriate procedure is a choice point that can be registered for backtracking. Of course, besides using a backtrack search for choosing a particular way to accomplish a goal there are other ways that the goal can invoke degenerate methods. For exam-

[21]This *pattern-directed invocation* was introduced by Hewitt in `PLANNER` [20] and by Alain Colmerauer in `PROLOG`.[27] This idea has spread to many other systems and languages.

ple, we may want to run several possible ways to solve a problem in parallel, choosing the one that terminates first.

Suppose we have several independently implemented procedures all designed to solve the same (imprecisely specified) general class of problems. Assume for the moment that each design is reasonably competent and actually works correctly for most of the problems that might be encountered in actual operation. We know that we can make a more robust system by combining the given procedures into a larger system that independently invokes each of the given procedures and compares their results, choosing the best answer on every problem. If the combination has independent ways of determining which answers are acceptable we are in very good shape. But even if we are reduced to voting, we get a system that can reliably cover a larger space of solutions. Furthermore, if such a system can automatically log all cases where one of the designs fails, the operational feedback can be used to improve the performance of the procedure that failed.

This degenerate design strategy can be used at every level of detail. Every component of each subsystem can itself be so redundantly designed and the implementation can be structured to use the redundant designs. If the component pools are themselves shared among the subsystems, we get a controlled redundancy that is quite powerful. However, we can do even better. We can provide a mechanism for consistency checking of the intermediate results of the independently designed subsystems, even when no particular value in one subsystem exactly corresponds to a particular value in another subsystem.

For a simple example, suppose we have two subsystems that are intended to deliver the same result, but computed in completely different ways. Suppose that the designers agree that at some stage in one of the designs, the product of two of the variables in that design must be the same as the sum of two of the variables in the other design.[22] There is no reason why this predicate should not be computed as soon as all of the four values it depends upon become available, thus providing consistency checking at run time and powerful debugging information to the designers. This can be arranged using a locally embedded constraint network.

Again, if we make systems that build themselves using generate-and-test

[22]This is actually a real case: in variational mechanics the sum of a Lagrangian for a system and the Hamiltonian related to it by a Legendre transformation is the inner product of the generalized momentum 1-form and the generalized velocity vector.[40]

mechanisms controlled by constraints that enforce requirements on the structure of the result, we will get significant natural degeneracy, because there will in general be multiple proposals that are accepted by the constraints. Also, because of the environmental differences among the instances of the systems to be built we will automatically get variation from system instance to system instance. This neutral space variation will give substantial resistance to invasion.

Infrastructure to Support Robustness and Evolvability

Combinators

If the systems we build are made up from members of a family of mix-and-match components that combine to make new members of the family (by obeying a predetermined standard protocol of interconnect), bigger perturbations of the requirements are more easily addressed by rearrangement of high-level components.

But how do we arrange to build our systems by combining elements of a family of mix-and-match components? One method is to identify a set of primitive components and a set of combinators that combine components so as to make compound components with the same interface as the primitive components. Such sets of combinators are sometimes explicit, but more often implicit, in mathematical notation.

The use of functional notation is just such a discipline. A function has a domain, from which its arguments are selected, and a range of its possible values. There are combinators that produce new functions as combinations of others. For example, the composition of functions f and g is a new function that takes arguments in the domain of g and produces values in the range of f. If two functions have the same domain and range, and if arithmetic is defined on their common range, then we can define the sum (or product) of the functions as the function that when given an argument in their common domain, is the sum (or product) of the values of the two functions at that argument. Languages that allow first-class procedures provide a mechanism to support this means of combination, but what really matters is a good family of pieces.

There are entire families of combinators that we can use in programming that we don't normally think of. Tensors are an extension of linear algebra to linear operators with multiple arguments. But the idea is more general than that: the "tensor combination" of two procedures is just a new procedure that takes a data structure combining arguments for the two procedures. It distributes those arguments to the two procedures, producing a data structure that combines the values of the two procedures. The need to unbundle a data structure, operate on the parts separately, and rebundle the results is ubiquitous in programming. There are many such common patterns. It is to our advantage to expose and abstract these into a common library of combinators.

We may use constraints when we model physical systems. Physical systems have conservation laws that are expressed in terms of dual variables, such as torque and angular velocity, or voltage and current. Primitive constraints and combinators for such systems are a bit more complex, but some have been worked out in detail. For example, the wonderful MARTHA system of Penfield gives a complete set of combinators for electrical circuits represented in terms of parts with two-terminal ports. [32]

Continuations

There are now computer languages that provide access to first-class continuations. This may seem to be a very esoteric construct when introduced in isolation, but it enables a variety of control structures that can be employed to substantially improve the robustness of systems.

A continuation is a captured control state of a computation.[23] If a continuation is invoked, the computation continues at the place represented by the continuation. A continuation may represent the act of returning a value of a subexpression to the evaluation of the enclosing expression. The continuation is then a procedure that when invoked returns its argument to the evaluation of the enclosing expression as the value of the subexpression. A continuation is a first-class object that can be passed as an argument, returned as a value, and incorporated into a data structure. It can be invoked multiple times, allowing a computation to be resumed at a particular point with different values returned by the continuation.

[23]This control state is not to be confused with the full state of a system. The full state is all the information required, along with the program, to determine the future of a computation. It includes all of the current values of mutable variables and data. The continuation does not capture the current values of mutable variables and data.

Continuations give the programmer explicit control over time. A computation can be captured and suspended at one moment and restored and continued at any future time. This immediately provides coroutines (cooperative multitasking), and with the addition of a timer interrupt mechanism we get timesharing (preemptive multitasking).

Backtracking and concurrency

Continuations are a natural mechanism to support backtracking. A choice can be made, and if that choice turns out to be inappropriate, an alternative choice can be made and its consequences worked out. (Wouldn't we like real life to have this feature!) So, in our square-root example, the square-root program should return the AMB of both square roots, where AMB is the operator that chooses and returns one of them, with the option to provide the other if the first is rejected. The receiver can then just proceed to use the given solution, but if at some point the receiver finds that its computation does not meet some constraint it can FAIL, causing the AMB operator to revise its choice and return with the new choice through its continuation. In essence, the continuation allows the generator of choices to coroutine with the receiver/tester of the choices.

If there are multiple possible ways to solve a subproblem, and only some of them are appropriate for solving the larger problem, sequentially trying them as in generate-and-test is only one way to proceed. For example, if some of the choices lead to very long (perhaps infinite) computations in the tester while others may succeed or fail quickly, it is appropriate to allocate each choice to a thread that may run concurrently. This requires a way for threads to communicate and perhaps for a successful thread to kill its siblings. All of this can be arranged with continuations, with the thread-to-thread communications organized around transactions.

Decorated Values

The ability to annotate any piece of data with other data is a crucial mechanism in building robust systems. The decoration of a value is a generalization of the tagging used to support extensible generic operations.

Let's consider some decorations that we think may be valuable in many situations.

One kind of decoration that a programmer may want to deploy in some parts of a program is the tracking of dependencies. Every piece of data (or procedure) came from somewhere. Either it entered the computation as a premise that can be labeled with its external provenance, or it was created by combining other data. We can provide primitive operations of the system with overlays that, when processing or combining data that is decorated with justifications, can decorate the results with appropriate justifications. This can be at differing levels of detail. For example, the simplest kind of justification is just a set of those premises that contributed to the new data. A procedure such as addition can for a sum with a justification that is just the union of the premises of the justifications of the addends that were supplied. Multiplication can be more complicated: if a multiplicand is zero that is sufficient to force the product to be zero so the justifications of the other operands are not required to be included in the justification of the product. Such simple justifications can be carried without more than a constant overhead in space, but they can be invaluable in debugging complex processes and in the attribution of credit or blame for outcomes of computations. However, this is sufficient to implement dependency-directed backtracking.

More complex justifications may also record the particular operations that were used to make the data. This kind of annotation can be used providing explanations (proofs) but it is intrinsically more expensive, potentially linear in the number of operations performed to obtain the result. But sometimes it is appropriate to attach a detailed audit history, describing the derivation of a data item, to allow some later process to use the derivation for some purpose or to evaluate the validity of the derivation for debugging.

For many missions, such as legal arguments, it is necessary to know the provenance of data: where it was collected, how it was collected, who collected it, how the collection was authorized, etc. The detailed derivation of a piece of evidence, giving the provenance of each contribution, may be essential to determining if it is admissable in a trial.

Of course, a programmer may choose to overlay this kind of justification when it is necessary to provide an explanation, or temporarily, to track a difficult to catch bug. A very close relative of this kind of computation can be used to implement symbolic partial evaluation, which can be the basis for doing algebra with the computations, as we have done in SICM.

Sensitivity analysis is a similar benefit that can be built with decorated data. For example, if we have a system that evolves the solution of a system of differential equations from some initial conditions, it is often valuable to understand the way a tube of trajectories that surround a reference trajectory deforms. This is usually accomplished by integrating a variational system along with the reference trajectory. Similarly, it is possible to carry a probability distribution of values around a nominal value along with the nominal value computed in some analyses. This may be accomplished by decorating the values with distributions and providing the operations with overlaying procedures to combine the distributions, guided by the nominals, implementing Baysian analysis.

An even more exciting but related idea is that of perturbational programming. Consider, for example, the problem that Google has doing a search. Given a set of keywords their system does some magic that comes up with a list of documents that match the keywords. Suppose we incrementally change a single keyword. How sensitive is the search to that keyword? More important, is it possible to reuse some of the work that was done getting the previous result in the incrementally different search? We don't know the answers to these questions, but if it is possible, we want to be able to capture the methods by a kind of perturbational program, built as an overlay on the base program.

Associations of data items can be implemented by many mechanisms, such as hash tables. But the implementation may be subtle. For example, the cost of a product will in general depend on different assumptions than the shipping weight of the product, which may have the same numerical value. If the computational system does not have a different token for each of these two equal numbers, the system does not have a way of hanging distinct tags on them.

Data-Propagation Infrastructure

We think of programs as wiring diagrams for (potentially infinite) machines. The machines have discrete terminals, which may be connected to terminals of other machines. It is through these interconnections that they communicate with neighbors. All computation is performed by machines operating on values that appear on their terminals and putting out values on their terminals.

This is not a novel view. The dataflow architecture community has had such a vision, and the constraint satisfaction community has had a related vision. We acknowledge these and build upon them. The reason why a vision of interconnected computational entities is such a compelling one is that it makes no claim about any sequential order of execution, or about direction of computation.

If we have such a network of interconnected machines we can imagine using it to compute in several different ways.

We can imagine "pushing" data through machines: a data signal that appears on some terminals of a machine stimulates that machine to perform a computation which generates results. The results are then pushed out on other terminals to other machines, propagating the results.

We can also imagine a request signal that is intended to "pull" data that we need from some output terminal of a machine: this stimulates the machine to propagate the pull signal to its neighbors, for the purpose of stimulating them to produce results that it will need to compute the value that will satisfy the pull on its output terminal. We may think of this as pushing a request signal, which ultimately is satisfied by a source pushing a response signal.

Each signal (requests and results) may be decorated with dependencies that record the reasons for the signal. Such decorated signals allow a very powerful control structure. For example, suppose there are multiple ways to satisfy a request. When a result appears that satisfies the request, that request is canceled and all requests that depend upon it are automatically canceled, thus stopping potentially useless computations. This idea was the driving force behind AMORD.

TMS

Dependency decorations on data give us a powerful tool for organizing human-like computations. For example, all humans harbor mutually inconsistent beliefs: an intelligent person may be committed to the scientific method yet have a strong attachment to some superstitious or ritual practices. A person may have a strong belief in the sanctity of all human life, yet also believe that capital punishment is sometimes justified. If we were really logicians this kind of inconsistency would be fatal. If we really were to simultaneously believe both propositions P and NOT P then we would have to believe all propositions! Somehow we manage to keep inconsistent beliefs from inhibiting all useful thought. Our personal belief systems appear to be locally consistent, in that there are no contradictions apparent. If we observe inconsistencies we do not crash, we chuckle.

A Truth Maintenance System (TMS) attaches to each proposition a set of supporting assumptions, allowing deductions to be conditional on the assumption set. So, if a contradiction occurs a process can determine the particular "nogood set" of inconsistent assumptions. The system can then "chuckle," realizing that no deductions based on any superset of those assumptions can be believed. This chuckling process, Dependency-Directed Backtracking, can be used to optimize a complex search process, allowing a search to make the best use of its mistakes. But enabling a process to simultaneously hold beliefs based on mutually-inconsistent sets of assumptions without logical disaster is revolutionary.

Since our infrastructure is based on propagation of data through independent machines it is possible to incorporate a TMS into the infrastructure in a natural and efficient way. Besides foundation beliefs, hypotheticals may be introduced by AMB machines, which provide alternative values supported by propositions that may be discarded without pain. Unlike systems modeled on expression-based languages such as Lisp, there is no spurious control-flow to pollute our dependencies and force expensive recomputation of already-computed values when backtracking.

Dynamically configured interfaces

How can entities talk when they don't share a common language? A computational experiment by Simon Kirby has given us an inkling of how language may have evolved. In particular, Kirby [25] showed, in a very simplified situation, that if we have a community of agents that share a few semantic structures (perhaps by having common perceptual experiences) and that try to make and use rules to parse each other's utterances about experiences they have in common, then the community eventually converges so that the members share compatible rules. While Kirby's experiment is very primitive, it does give us an idea about how to make a general mechanism to get disparate modules to cooperate.

Jacob Beal [6] extended and generalized the work of Kirby. He built and demonstrated a system that allowed computational agents to learn to communicate with each other through a sparse but uncontrolled communication medium. The medium has many redundant channels, but the agents do not have an ordering on the channels, or even an ability to name them. Nevertheless, employing a coding scheme reminiscent of Calvin Mooers's Zatocoding (an early kind of hash coding), where descriptors of the information to be retrieved are represented in the distribution of notches on the edge of a card, Beal exchanges the sparseness and redundancy of the medium for reliable and reconfigurable communications of arbitrary complexity. Beal's scheme allows multiple messages to be communicated at once, by superposition, because the probability of collision is small. Beal has shown us new insights into this problem, and the results may be widely applicable to engineering problems.

Conclusion

Serious engineering is only a few thousand years old. Our attempts at deliberately producing very complex robust systems are immature at best. We have yet to glean the lessons that biological evolution has learned over the last few billion years.

We have been more concerned with efficiency and correctness than with robustness. This is sensible for developing mission-critical systems that have barely enough resources to perform their function. However, the rapid advance of microelectronics has alleviated the resource problem for most applications. Our increasing dependence on computational and communications infrastructure, and the development of ever more sophisticated attacks on that infrastructure, make it imperative that we turn our attention to robustness.

I am not advocating biomimetics; but observations of biological systems give us hints about how to incorporate principles of robustness into our engineering practice. Many of these principles are in direct conflict with the established practices of optimization and proofs of correctness.

As part of the continuing work to build artificially intelligent symbolic systems we have, incidentally, developed technological tools that can be used to support principles of robust design. For example, rather than thinking of backtracking as a method of organizing search we can employ it to increase the general applicability of components in a complex system that builds itself to meet certain constraints. I believe that we can pursue this new synthesis to get better hardware/software systems.

Bibliography

[1] Harold Abelson and Gerald Jay Sussman with Julie Sussman, *Structure and Interpretation of Computer Programs*, 2nd edition, MIT Press, ISBN 0-262-01553-0, (1996).

[2] Lee Altenberg; " The Evolution of Evolvability in Genetic Programming," in *Advances in Genetic Programming,*(Kenneth E. Kinnear, Jr. editor) chapter 3, pages 47-74. MIT Press, (1994).

[3] Harold Abelson, Don Allen, Daniel Coore, Chris Hanson, George Homsy, Thomas F. Knight Jr., Radhika Nagpal, Erik Rauch, Gerald Jay Sussman, and Ron Weiss; "Amorphous Computing," in *Communications of the ACM*, **43**, 5, May 2000.

[4] *The ARRL Handbook for Radio Amateurs*, The American Radio Relay League, Newington, CT, USA (annually).

[5] Jonathan B.L. Bard; *Morphogenesis*, Cambridge University Press, (1990).

[6] Jacob Beal; *Generating Communications Systems Through Shared Context*, M.I.T. S.M. Thesis, also AI Technical Report 2002-002, January 2002.

[7] Jacob Beal; "Programming an Amorphous Computational Medium," in *Unconventional Programming Paradigms International Workshop*, September 2004. Updated version in *LNCS*, **3566**, August 2005.

[8] M.R. Bernfield, S.D. Banerjee, J.E. Koda, and A.C. Rapraeger; "Remodelling of the basement membrane as a mechanism of morphogenic tissue interaction," in *The role of extracellular matrix in development*, ed. R.L. Trelstad, pp. 542–572, (Alan R. Liss, 1984).

[9] J P. Brocks, "Amphibian limb regeneration: rebuilding a complex structure." in*Science* **276**:81–87 (1997).

[10] R. Bulirsch and J. Stoer; *Introduction to Numerical Analysis*, Springer-Verlag, (1991).

[11] "Self-Assembly and Self-Repairing Topologies," in *Workshop on Adaptability in Multi-Agent Systems*, RoboCup Australian Open, January 2003.

[12] E.M. del Pino and R.P. Elinson; "A novel developmental pattern for frogs: gastrulation produces an embryonic disk," in *Nature*, **306**, pp. 589–591, (1983).

[13] G.M. Edelman and J.A. Gally; "Degeneracy and complexity in biological systems," *Proc. Natl. Acad. Sci*, **98** pp. 13763–13768 (2001).

[14] M. D. Ernst, C. Kaplan, and C. Chambers. "Predicate Dispatching: A Unified Theory of Dispatch," In ECOOP'98. LNCS, vol. 1445. Springer, Berlin, 186–211 (1998).

[15] Robert Floyd; "Nondeterministic algorithms." in *JACM,* 14(4):636–644 (1967).

[16] Kenneth D. Forbus and Johan de Kleer; *Building Problem Solvers*, The MIT Press, (November 1993).

[17] Stefanie Forrest, Anil Somayaji, David H. Ackley; "Building Diverse Computer Systems," in *Proceedings of the 6th workshop on Hot Topics in Operating Systems*, IEEE Computer Society Press, pp. 67–72, (1997).

[18] Joseph Frankel; *Pattern Formation, Ciliate Studies and Models*, Oxford University Press, New York, (1989).

[19] Chris Hanson, `SOS` software: Scheme Object System, (1993).

[20] Carl E. Hewitt; "PLANNER: A language for proving theorems in robots." In *Proceedings of the International Joint Conference on Artificial Intelligence,* pp. 295–301 (1969).

[21] Paul Horowitz and Winfield Hill; *The Art of Electronics*, Cambridge University Press.

[22] Paul-Alan Johnson; *The Theory of Architecture: Concepts, Themes, & Practices*, New York: Van Nostrand Reinhold (1994)

[23] Richard Kelsey, William Clinger, and Jonathan Rees (editors), *Revised^5 Report on the Algorithmic Language Scheme*, (1998).

[24] Gregor Kiczales, `tinyCLOS` software: Kernelized CLOS, with a metaobject protocol, (1992).

[25] Simon Kirby; *Language evolution without natural selection: From vocabulary to syntax in a population of learners.*, Edinburgh Occasional Paper in Linguistics EOPL-98-1, University of Edinburgh Department of Linguistics (1998).

[26] Marc W. Kirschner, John C. Gerhart; *The Plausibility of Life: Resolving Darwin's Dilemma*, New Haven: Yale University Press, ISBN 0-300-10865-6 (2005).

[27] Robert M. Kowalski; "The Early Years of Logic Programming," in *CACM*, **31**:1, January 1988, pp.38–43.

[28] Harvey Lodish, Arnold Berk, S Lawrence Zipursky, Paul Matsudaira, David Baltimore, and James E Darnell; *Molecular Cell Biology*, (4th ed.), New York: W. H. Freeman & Co., ISBN 0-7167-3706-X (1999).

[29] David Allen McAllester and Jeffrey Mark Siskind; `SCREAMER` software, see `http://www.cis.upenn.edu/ screamer-tools/`.

[30] Piotr Mitros; *Constraint-Satisfaction Modules: A Methodology for Analog Circuit Design*, PhD Thesis, Electrical Engineering and Computer Science Department, MIT, (2007).

[31] John McCarthy; "A basis for a mathematical theory of computation," in P. Braffort and D. Hirshberg, editors, *Computer Programming and Formal Systems*, pages 33–70, North-Holland, (1963).

[32] Paul Penfield Jr.; *MARTHA User's Manual*, Massachusetts Institute of Technology, Research Laboratory of Electronics, Electrodynamics Memorandum No. 6; (1970).

[33] W. H. Press, B. P. Flannery, S. A. Teukolsky, and W. T. Vetterling; "Richardson Extrapolation and the Bulirsch-Stoer Method," in *Numerical Recipes in C: The Art of Scientific Computing*, 2nd ed. Cambridge University Press, pp. 718-725, (1992).

[34] Guido van Rossum, and Fred L. Drake, Jr. (Editor); *The Python Language Reference Manual*, Network Theory Ltd, ISBN 0954161785, (September 2003).

[35] http://www.python.org/dev/peps/pep-0020/

[36] Richard Matthew Stallman; *EMACS: The Extensible, Customizable, Self-Documenting Display Editor*, Massachusetts Institute of Technology Artificial Intelligence Laboratory Memo, AIM-519A (March 1981).

[37] Richard Matthew Stallman and Gerald Jay Sussman; "Forward Reasoning and Dependency-Directed Backtracking in a System for Computer-Aided Circuit Analysis," in *Artificial Intelligence*, **9**, pp 135–196, (1977).

[38] Guy L. Steele Jr.; Common Lisp the language, The Digital Equipment Corporation, (1990).

[39] Guy L. Steele Jr.; *The Definition and Implementation of a Computer Programming Language Based on Constraints*, MIT PhD Thesis, MIT Artificial Intelligence Laboratory Technical Report 595, (August 1980).

[40] Gerald Jay Sussman and Jack Wisdom with Meinhard E. Mayer, *Structure and Interpretation of Classical Mechanics*, MIT Press, ISBN 0-262-019455-4, (2001).

[41] *The TTL Data Book for Design Engineers*, by the Engineering Staff of Texas Instruments Incorporated, Semiconductor Group.

[42] Stephen A. Ward and Robert H. Halstead Jr.; *Computation Structures*, MIT Press, ISBN 0-262-23139-5, (1990).

[43] Daniel J. Weitzner, Hal Abelson, Tim Berners-Lee, Chris Hanson, Jim Hendler, Lalana Kagal, Deborah McGuinness, Gerald Jay Sussman, and K. Krasnow Waterman; *Transparent Accountable Data Mining: New Strategies for Privacy Protection*, MIT CSAIL Technical Report MIT-CSAIL-TR-2006-007 (27 January 2006).

[44] Carter Wiseman; *Louis I. Kahn: Beyond Time and Style: A Life in Architecture* New York: W.W. Norton, ISBN 0393731650 (2007).

[45] Lewis Wolpert, Rosa Beddington, Thomas Jessell, Peter Lawrence, Elliot Meyerowitz, and Jim Smith; *Principles of Development* (2nd ed.), Oxford University Press, ISBN-10: 0-19-924939-3, (2001).