Rapid Software Evolution

From Construction to Computation and Back

Borislav Iordanov
Kobrix Software, Inc.
biordanov@acm.org

Abstract

Software has quantifiably reached levels of complexity beyond human understanding. The divide and conquer principle breaks for large-scale systems for clearly identifiable reasons. To overcome those barriers, the process of software construction should be modeled after the process of evolution in the living world. We propose a concrete, practical platform for evolving software programs through natural selection and continuous human participation. The presentation is informally structured as a dialog.

Categories and Subject Descriptors D.2.m [*Software Engineering*]: Miscellaneous

General Terms Design, Theory

Keywords evolutionary engineering, software construction, complexity, distributed computing, knowledge management, live-system, hypergraph

1. Prolog

Chewbacca is a young, hairy, tireless and clever dog. Josefina is a thoughtful, conservative, aristocratic lady dog. While not in the field themselves, motivated by the noble desire to participate and understand their masters' livehood, they like to discuss ideas about software engineering while said masters are not around.

The following dialog begins with a systems theory perspective on software complexity and an argument for the theoretical and practical limitations of traditional divide and conquer approaches such as modularity and abstraction. A biological evolutionary process for the creation of software is argued for and a conceptual framework for software construction inspired from the idea of evolutionary engineering ([4], [5], [9]) is proposed. Finally, an ongoing implementation of a platform based on this framework is presented. The platform is founded on a distributed memory model structured as a generalized hypergraph, offering facilities for programming language interoperation and management of rich semantic information.

2. A Chat About Complexity

It is early in the morning. The masters have left the house rushing to work as usual. Chewbacca kindly escorted them to the door and went back to wake up Josefina in an unusually excited mood.

C: Good morning, Dear! Wake up! We have an exciting day ahead of us and that melancholic look on your face could almost have me believe that I'm alone in my enthusiasm.

J: It's a deep look, not a melancholic one. I know better, for it is I who produce it.

C: Sure, sure. . . anyway, remember last night I told you about this idea of rapid software evolution? If you still had a few active neurons, you'd recall that I conjectured that people could, and indeed should, be creating software programs by evolving them, rather than by designing them *ab initio, ad infinitum*.

J: Oh, my! A good night's sleep didn't help it, hm? I guess that explains all the giggling and jumping around. So let me see. You want to make a program that evolves by itself, yet towards some sort of purposefulness. A rather strange proposition, I must confess. And I'm curious what drove you to that idea.

C: Well, people are creating those things called computer programs, that they very much like to conceive of as intentional. And as time passes by, they are a great deal inclined to imbue them with behavior that is, quite frankly, going out of proportions with anything resembling a predictable, well-engineered beauty of a system.

J: You mean, they want to implement more and more functionality in order to meet ever increasing business needs in a competitive global market?

C: Something like that. So, they want software that is autonomous, adaptive, contextual, resilient, that deals with new and unexpected, complex situations, that is optimized for frequent tasks, while still capable of carrying the less frequent ones, perhaps not so efficiently. It seems like the stated challenge is a rehash of the historic goal of creating truly smart machines!

J: Listen, then why don't you go chat with AI guy and leave me alone.

C: Are those guys still around? But you are right, that's the point. Many of the conceptual underpinnings of modern programming originated with AI research and symbolic models of cognition. For instance, object-orientation was heavily influenced by Marvin Minsky's frame theory ([34]). Now that the focus of research has shifted towards the construction of the so called "complex systems", maybe AI researchers could give the starved for innovation software engineering discipline another push. But, allow me to continue.

J: Please, go on.

C: Good. So, we all agree that software is getting way too complex. That term is somewhat vague in an engineering context, and I believe we can narrow down a more precise meaning than the rather subjective "something complicated".

J: I'm listening.

C: First, think of a software system as an object of study for a minute, instead of as an engineering problem to be solved. Say, you'd like to quantify how complex that system is. What would you do?

J: I could count the number of lines. Or, the number of classes. I could also count the number of dependencies between modules. I could do some code analysis and count the number of branches in its control flow.

C: That reminds me of an interview question where the candidate had to optimize a fairly long and complicated C program. After thirty minutes of sweating and nail biting, he managed to conclude that the program was invariably printing the string "Hello" on its output. So, while valid those sorts of measures do not reflect the functional complexity of your system. For the latter is context-dependent upon the environment within which the software operates and only looking at internal structure ignores the environment altogether. A programmer ignorant of the system's context becomes a black box like the program they code and this just shifts the difficulty elsewhere.

J: Agreed. And what would you propose as an alternative, if I may ask?

C: You certainly may and I would regret to disappoint you, but the measure that I'd propose is something that you already know. There's a very simple, yet informative way to look at the complexity of something and that we're very familiar with: we measure the length of the shortest possible description of that something. It's called algorithmic complexity ([46], [29], [18], [19]). Now, to account for the full functional complexity ([3], [6]) of a system, one needs to characterize the environment (the set of possible inputs) as well as the system's responses to each of the different inputs. If it takes at least $C(E)$ bits to describe the environment and at least $C(A)$ bits to describe each possible response action of a system then the functional complexity is

$$C(F) = C(A)2^{C(E)}$$

J: How did you calculate that?

C: Well, there are $2^{C(E)}$ possible inputs, right?

J: Right.

C: And for each of them, you have to specify the corresponding action, right?

J: Yeah…

C: So you need $C(A)2^{C(E)}$ bits to fully specify the system's behavior, in terms of algorithmic complexity that is.

J: Got it.

C: Good. Now, let's pause for a bit and see what that formula tells us. Firstly, it quantifies a system's complexity from an observer's point of view where both system and environment are taken into account. Judging from your mind set in regards to measuring complexity, you could argue that a more truthful approach would be to examine the system's internals only and calculate the algorithmic complexity of the program itself. To which I reply that a progammer has no choice but to take the observer's stance because the software construction process, an utterly creative process, amounts to a *constant folding of program and environment, a continuous rubing of code against the human world so to speak* ([15]). Incidentally, it is this process that makes programs "intentional by extension". As Brian Smith puts it ([44], p.360):

> ... we are expanding our registrational capacities – building instruments and other devices that mediate our full participation in the world

Moreover, current programming tools, namely programming languages for the most part, are by their nature ultimately means for expressing functional descriptions. In other words, for obvious reasons programming does not involve creating a set of interacting components out of which behavior emerges in an unpredicatable way. To the contrary, the relationship between input and output is rather explicitly encoded in a program. This is yet another reason why, from an engineering perspective, a functional approach to the complexity of software systems is more relevant than a structural one.

J: Relevancy granted.

C: Next, note that increasing $C(A)$ by one bit, increases the system's complexity by $2^{C(E)}$ bits. This means that adding new functionality while maintaining the operational context constant may still raise the overall complexity by a factor depending on that operational context's complexity. Worse, changing the environment while maintaining the same exact behavior may increase the system's complexity exponentially!

The point of all this, my dear Josefina, is that this is the complexity that programmers must confront. This is the sort of thing that Fred Brooks so eloquantly describes in [10].

Programmers cannot eliminate or reduce that complexity, because it is what it is. Instead, they must match it with copious amounts of code. And programmers, being themselves human beings with their own limitations, for all practical purposes quantifiable by the same equation above, can only handle that much, some more some less, but essentially the upper bound is quickly reached.

J: Your argument is very powerful. However, I must go back to what I know, to my foundations that have been so thoroughly explored an rehearsed over the years, that the simple trick of divide and conquer does wonders to apparently intractable problems. For instance, I can modularize and encapsulate, stratify, abstract and instantiate, and so forth.

C: Modularization isolates pieces into self-contained and autonomous parts. When looked upon statically, in the logical space of program constructions, they make perfect sense. The assumption is then that they can be glued together to make a working whole. That assumption takes the whole as a literal sum of its parts, the modules. But the running program is rarely a completely faithful realization of the static context within which modules are created. While at coding time we strive for low coupling, at run-time the opposite must happen, by magic - the indepedent parts must come together as a unified whole. The magic works most of the time, but I contend that this is so only because our programs are still deceptively simple. The problem is not that designing by divide and conquer is absolutely wrong and should be dropped. It is still the only way to produce meaningful components and put systems together. For instance, human perception and cognition are in a way based on information hiding, i.e. filtering input for relevancy. The problem is that when looking at the system's totality at runtime, the components become irrelevant and hidden interdependencies resurface with their ugly horns. As pointed out in [17], *the purpose of information hiding plays no role in the execution of a program.* Further, the reason those dependencies crop up is very fundamental: without them, the system wouldn't be able to reach the variety required by its environment and defined by its function. So they are inevitable. As all metaphors, "divide and conquer" goes only so far. After all, Caesar wanted each part weaker, while engineers would like each part to be stronger (at runtime). So modularization fails as a weapon against complexity.

J: I strongly disagree. Modularization fails only when the abstractions weren't properly defined. If the design of the program is grounded in sound, solid abstractions, all potential dependencies between components would be already accounted for. Sound abstractions come to be by following proper modeling, good logic and description of the world as it is. If your model is logically sound, in the sense of having the right properties, the right relationships with the right cardinalities, you won't have suprises along the way. Low coupling is to be combined with high-cohesion, may I remind you.

C: I agree that this attitude seems beneficial from a practical point of view. It gives you a sense of confidence and it allows you to make progress. Also, I too shared the intuition that there is a sense in which an abstraction can be qualified as "right" or "wrong" and that it is not just an arbitrary construct whose value stems purely from usefulness in such and such context. After all, agreement in human communication could have never been reached if abstractions were so arbitrary.

J(ironically): Thank you.

C(ignoring irony): However, I would conjecture that this view is partially successful not so much due to the correctness of those abstractions as handles to some platonic entities, but because of the internal coherence constantly enforced by the programmer, the compiler and the runtime system. Enforcing this sort of coherence, even only at the syntactic level, is what makes software so resistant to change. As time passes, a system becomes less and less able to absorb new perturbations (in terms of modifications), again because the law of requisite variety ([1]) demands a never ending expansion of those abstractions and an ever increasing coupling between modules.

Now, observe that the software abstractions you are talking about are the ones that get transformed into artifacts, source code to be precise, and therefore they are in actuality always reified. And as soon as they get implemented, they take on a life of their own, and a very concrete one at that! The reification process strips them from their potential fluidity and malleability as models and that is why you demand an impeccable logical correctness from the start. Creating an abstraction is a big commitment! Even when some logical correctness is achieved, the exact details that are left during the abstraction process are, to begin with, themselves arbitrary. But this arbitrareness does not guarantuee that the precise set of details left out is irrelevant to other parts of the system. Just think of the number of times one wishes to specify this extra parameter to override such and such default behavior.

J: Of course, to specify what is to be left out, one must assume how the abstraction is to be used.

C: And since one cannot assume such a thing, one defines usage instead. But one cannot define how the world is to be, so abstractions ulimately fail in new contexts. Logical faithfullness of world models has only limited applicability.

J: I see your point. Especially in object-oriented modeling, design follows a rather linear process of differentiation, starting from that which is common to all and proceeding, in one direction, to that which is distinct from all. This linear process of reducing degrees of freedom is too rigid to make abstractions as context-sensitive as one would hope.

C: Precisely! So, no silver bullet there either. I mean, it all works well for systems of moderate complexity, but...

J: That is a rather gloomy perspective. You seem to be insinuating that programmers have reached, or will soon reach, the limits of what's possible to achieve in software engineering. And that the availability of more, better, faster hardware resources becomes pointless because software has attained its top complexity level.

C: On the contrary! It is an opportunity to do something completely new and exciting!

J: Oh! Enlighten me then please!

3. Growth vs. Evolution

C: I think we should have some lunch first. But let me conclude the argument. Lost in yet another evaluation of the state of affairs in software development, you seem to have forgotten where we started.

J: Biological evolution?

C: Right. To sum up: have I managed to convince you that there is intractable complexity in what people want to do in order to take software to the next level?

J: Yes.

C: And that while the traditional methods of dealing with complexity in engineering may have localized success, on a large scale and over time they break down miserably for very fundamental reasons?

J: I'm sort of more in line with your thinking in this respect now, yes.

C: Good. My conclusion from all this is, so to speak, "natural": to attain those ever increasing levels of complexity, an evolutionary process in software construction must be embraced. A process that mimics biological evolution in all its glory. This means a changing population of individuals, with variation and natural selection. No engineering effort was ever capable to even get close to producing something similar to a living system. Evolution, on the other hand, has done miracles. Call me copycat, but this is the only known process that is capable of generating systems of versatile and complex behavior such as yourself. The difficulty, of course, lies in the realization of such a process in a practical and fruitful way.

J: That's precisely my concern. There is a certain paradox in biological evolution that seems to invalidate it as an engineering process, namely all living organisms have one and only one readily recognizable purpose as individuals and that is to remain the same, to sustain their life and identity. Yet they change. On the other hand, a system is constructed with a desired end goal, something very different from the initial substrate out of which it is built. And in a way software is already being built by evolving it from something simple into an entity with progressively richer behavior and wider applicability. In fact, what is generally meant by "software evolution" is this process of maintaining and growing a system to satisfy its users.

C: Well, first of all remember that in the living world change operates at the population level! But let's look at orthodox software evolution more carefully. This process has been extensively studied and refined. See, for example, the FEAST project reports ([30], [31]). Software growth is characterized as a multi-feedback loop involving all participants from developers to users. But isn't it striking how Lehman's first[1] and sixth[2] laws of software evolution are collectively reminiscent of a living organism's goal of survivability. The only difference is that the context within which such evolution occurs is rather limited. The result is a single product that must meet everybody's expectations. It is very rare that a company manages to produce several variations of the same product in order to satisfy competing and incompatible needs. Variation is introduced later, at deployment time, and the burden falls on the user, or on highly paid consultants to tailor/customize the product through cryptic configuration files or a myriad of option dialogs, or macro systems. In a sense, this type of evolution balances two sorts of pressures: the pressure to change coming from the market and users, and the organization's resistence to change coming from engineers' desire to keep the complexity low and management's desire to keep the cost low.

J: That's debatable. I don't think that change is seen as evil in software development circles. On the contrary, people love to see their programs grow and incorporate more features.

C: True, but as long as that doesn't mean rewriting a substantial chunk of the program. The only kind of change engineers accept without restraint is addition of functionality that does not impact the existing system. Otherwise, there is resistence, regardless of extreme programming and the like, programmers strive towards making things generic, immutable, rock-solid, reusable, whatever. And the reasons for that are very well-known.

Thus, the most fundamental aspect of this sort of feedback-only driven growth of software is the lack of variability. Uniformization is seen as an inviolable principle, perhaps mostly because of economic considerations, perfectly valid considerations I must confess. However, this has several unfortunate consequences:

1. Users must pay for and then endure the "gorilla" when all they want is the "banana".
2. Programmers must often aggregate and account for conceptually incompatible functions in order to forcefully maintain unity.
3. Users must operate in what is the most common denominator. More sophisticated users are not given the ability to adapt the software to their way of doing things, while less sophisticated users get lost into what seems overly complicated for their simple task.

[1] An E-type program that is used must be continually adapted else it becomes progressively less satisfactory.

[2] Functional content of a program must be continually increased to maintain user satisfaction over its lifetime.

In addition, and perhaps even more importantly, a rather unfortunate consequence of all this is **the pogressive divide between the solution space and the problem space**. This is opposite to what intuition tells us and to the overall engineering mandate, which is to align both spaces.

J: I'm losing you here...

C: On one hand, you have an open-ended and ever changing problem, which is to meet real world user requirements. This requires matching ever changing user practices, business processes and expectations, taking advantage of better and cheaper hardware. On the other, you have the solution, a reified abstract construction - the program - that, through growth and increasingly unmanageable internal complexity, reduces the set of possibilities in alternative behavior and therefore adaptation. This divide gets deeper and deeper because the variability of the environment (i.e. users, other systems) cannot be matched by the variability in the program as an abstract construction.

J: This is a very strong position. I must remind you that many programs have remained untouched, yet in use for decades!

C: Of course. Likewise in nature - organisms that have not found compelling reasons to adapt have remained the same for billions of years. There are further parallels. For instance, the standard explanation of the origin of multicellular organisms is that single-cell organisms cooperated together against an adverse environment. The same goes for individual programs that get integrated/bundled with other programs to present more attractive packages.

J: Ok, but the fundamental problem with your proposal remains! Engineering means constructing systems with strict, predefined function and purpose. Now, I understand that the evolutionary process that you propose is more of the Lamarckian kind[3] so it's not completely driven by randomness and natural selection.

C: That's correct.

J: Nevertheless, the core aspect of evolution as a creative process is the lack of a definite end-goal. And engineering is about construction of systems with definite behavior. This makes a top-down strategy the rule.

C: A top-down strategy demands that one be at the top, a very presumptuous requirement when speaking about large-scale systems. When you can't start from the top, and when you don't have a clear path towards it, the best option is exploration. But let me give you a different angle then. Classical physics describes the world in terms of causes and effects. Engineering has grown out of this mechanistic world view. There are forces and the action of a force, exerted by something, causes such and such effect on something else. There is no effect without a cause and vice-versa. Mechanics thus enjoys a rather clean, and easily graspable logical structure. The classical theory of computation also is entirely concerned with causality. And our programming practices are heavily dominated by that view. But think in terms *computation in the wild*, as coined by Brian Smith ([44]), instead of algorithmics. This is computation as it occurs in the real world, all of it, computation as it participates in our lives. It is a complex phenomenon embedded within the human cognitive web beyond the graps of a single individual. Now, evolutionary theory is recognized to have a comparable explanatory power to mechanics when it comes to complex adaptive systems, the type of thing that we would want to create in software. When trying to explain how complex systems come about and why they do the things that they do, one needs evolution or at least nobody has yet found a satisfactory alternative framework. **Now, you just need to make the same shift from the descriptive, explanatory role of a theory (classical physics) to a prescriptive endeavor (engineering), but with complex systems and evolution.** The "only" difference is one of scale.

J: Let's grab a bite. I'd like some time to digest this.

4. HOWTO Evolve Software

After lunch, Chewbacca seems sleepy and exhausted. Josefina gives her an amused look this time.

J: I must confess that your argument about using biological evolution to build software seems a fair bit convincing. And indeed, I started wondering what you have in mind. The crux of your argument seems to be that the process of biological evolution would provide for enough variability in program construction.

C: Yes. Recall the essential ingredients of an evolutionary process:

1. A growing population of individuals.
2. Means for creating variation amongst individuals.
3. Selective pressure to eliminate unfavored individuals.

When such a process is put in place for program construction, I believe a lot of the design headaches that programmers face would vanish. Because this process is exploratory by nature, it has a much better chance of yielding success.

J: I'm eager to hear about the specifics of your proposal. Is it something like GAs (genetic algorithms, [23])? Are you proposing to build programs through some sort of gigantic genetic algorithm?

C: Not really. But your intuition is correct. That would be a theoretically valid approach, but it would be going to the other extreme: from complete upfront planning to no planning at all. I think that programs should be seen as points in a solution space satisfying some fitness criteria. However, in a GA the units of variation are too fine-grained, too narrow. This makes GAs appropriate for well-defined optimization problems where fitness can be evaluated as some function computable in a few milliseconds. In other words, GAs operate at a time scale much smaller than that of human beings and can hardly be incorporated into an engineering process, except for specific narrow domains.

[3] See, for example, http://en.wikipedia.org/wiki/Lamarck

And I must point that this is actually being done already in electronic chip design precisely for the reasons of intractable complexity that I've explained.

J: You are right. Building software by running some genetic algorithm would simply be too slow. Software today is different. It operates at a fairly high cognitive level. It must interact with and be meaningful to humans.

C: And as I said, it is "intentional" in some sense for it participates in our daily lives, a trend that can only increase from this point on. The environment within which software evolves, its evolutionary context if you will, IS our daily lives. What defines success is how well a program participates in the human cognitive web, whether it is used, whether people are happy with how it works, how it looks, how fast it is and so forth. A program is evaluated by people, changed by people, reproduced by people.

J: I see. And in fact this is why we have markets and competition between programs. This is why we have so many open-source projects, each striving for attention. Isn't that then what you are talking about? Programs created by different organizations or groups of people competing for adoption?

C: Not really. But your intuition is correct again. I think that programs should be evaluated for their fitness by end-users as much as by programmers and I think that usage is what should drive change. When you view this software market process from a global perspective, you realize that it is hopelessly inefficient as well, since it operates on too big of a time scale. Furthermore, the granularity is too coarse. A whole program can "win" over another, but specific features, specific parts of it might be worst that the corresponding parts of the "loser" program, and users are stuck with them.

J: And you are arguing for a middle ground?

C: Bingo! We need a middle ground. Software must be evolved through units of information understandatable by humans, such as abstract data types, data structures, algorithmic processes/functions, inference rules, parameter sets, help text even and what have you. In other words, everything that's produced to construct software but looked at more granularly, above the bit, below the shrink wrapped product. When a change is introduced at such a level of granularity, it should be given a chance to live for a while, and not decided against by a designated architect, code owner or a committee of committers. For such decisions are made in the face of uncertainty and lack of complete knowledge, and as argued in ([48]), they are best made by crowds rather than single individuals.

J: I don't understand how the dynamics of such a thing would work. How is change validated? How and when does it reach users for it to "live for a while"?

C: Well, dear friend, let's flesh it out step by step. What would a platform for evolutionary engineering look like? As I mentioned a while ago, humans make up the environment where computer programs evolve in the same way a natural habitat is the environment in which living species evolve. All humans have a participatory role in a program's evolution. While in a biological world, the constitutive elements of a system are composed of physical matter, in the silicon world the elements are drawn from the cognitive landscape of human knowledge, interaction and information exchange. A key point that emerges from this perspective is that all forms of participation must be incorporated in the evolutionary process, from the core programmer to the end-user - they should be woven together into the same computing medium.

J: Hmm . . . users and developers sharing the same computing medium. That reminds me of environments like Smalltalk ([22]) or Self ([42]).

C: And rightfully so. An open, live-like system where a user interacts directly with completely visible and editable software artifacts that are persisted across sessions. So, let's stipulate:

Characteristic 1: Open, live programming-interactive system.

This yields a tremendous development speed boost, the benefits of continual testing etc. What else?

J: Speaking of a "computing medium" and the idea that the programs' habitat comprises all humans, aren't you also assuming individual systems are interconnected somehow?

C: Indeed, I am. For changes to propagate, for software elements to diversify, specialize, be selected for or against, a platform for evolutionary engineering must be comprised of networked live systems. The larger the scale of this networking, the better! And we have:

Characteristic 2: A decentralized network of individual, local systems.

C: Thus, the vision would be for a distributed prototyping environment. Because of the much finer level of granularity and immediate visibility of change, many pairs of eyes can contribute and validate a new piece of code or data. What else?

J(looking suspiciously): Not sure at this point. However, I'm wondering what kind of programming language would people use? Smalltalk or Self or some derivative seem natural candidates for a live system. Or, something completely new and tailored to this paradigm?

C: It's an interesting question and a logical one at this stage. Any thoughts?

J: As I said, why not adopt an existing language suitable for a prototyping environment. Some dynamic language that yields itself easily to data persistence. The Squeak Smalltalk implementation comes to mind ([45]). But then, it seems confined to a fervent, yet relatively isolated community. Perhaps a more modern language with an aura of practicality, like Perl 6, would be best. On the other hand, why not create a language specific for this paradigm. A new paradigm calls for a new language to express it after all!

C: I hardly see anything in a programming language that makes it so special for an evolutionary engineering paradigm.

J: That it must be interpreted seems like an absolute requirement to me.

C: Insofar as some sort of late binding is available. But I would consider that a detail. In broader terms, the most important differences in programming languages are the so called "programming paradigms" - imperative, functional, logic and so forth. In any event, why do you assume that there must be a single language in use? On one hand, every person has a different background and talent, and on the other every problem yields itself best to a different programming paradigm. Large modern systems are a hodge podge of components implemented in several languages mostly because each language is particularly suitable for some task or the other. So it seems to me that choice must be given and no one language could be argued to best fit the bill. Mutliple paradigms must be able to interoperate naturally and the environment must be of immediate practical use.

Josefina kindly raises her voice in an effort not to stiffle Chewbacca's ongoing enthusiasm

J: Wait, hold on! You can't just mix languages arbitrarily like this. Do you know how hard it is to get languages even within the same paradigm (e.g. object-oriented) to interoperate and share data? Let alone languages that are conceptually at odds.

C: It's certainly a difficult and very interesting problem, but not undoable. The key to interoperability is shared data representation and I will tell you the specifics of my thoughts on the subject in a bit. For now, we are fleshing out requirements. So:

Characteristic 3: Mutli-paradigm, language neutral programming with a shared information representation.

J: This sounds like a grand vision! Since you've mentioned the practicality aspect, I am beginning to wonder what could be really achieved in practice. The feeling I'm getting as I begin to envision such a platform myself is, to put it politely, an incorrigible mess. You are essentially leading towards a process whereby a large population of participants is encouraged to produce a potpourri of software artifacts, small and large and surely yielding a maze of dependencies with hidden, implicit intents.

C(simulating a smile to the best of her abilities): And producing surprising results!

J: Well, I would say most likely unsurprisingly producing no results at all.

C: That sort of scepticism is boringly typical of you. Incidentally, Andreas Wagner, comparing living and engineering systems in his comprehensive analysis of robustness and evolvability mentions ([49], p.311) that, contrary to man-made systems, ... *the parts of living systems behave much more eratically* and *many organismal systems perform their function in complicated, apparently inelegant and outright byzantine ways*. Both aspects are associated with the robustness of living systems at every level of organization.

J: Precisely my point! This poses an important question as to the strength of the biological metaphor for systems engineering, namely should people be living with engineered systems that perform their function in "apparently inelegant and outright byzantine ways".

C: They already are! They just can't stop complaining about it. Well, some don't complain so much ([35]). But note that this apparent messiness is modulated by proper management. To pursue the analogy a bit further, note that as more of the mechanisms through which DNA yields a living organism are uncovered, it is being realized that, especially for multicellular organisms, the majority of the processes and pathways involved deal purely with *management and regulation* of what we would label *core functionality*. Specifically, it is mostly regulation that increases with an organism's complexity rather than the number of functional components ([11]). To state the obvious, coordination is an inherent aspect of any complex system.

J: Yes, there are layers of organization in the expression of DNA. And the so called junk DNA turned out not to be disposable after all.

C: Right, to the contrary, it is now given primacy status by many! The conclusion for us then is that software incurring major coordination overhead shouldn't be considered so bad for it would be no worse than us living beings. However, I would like to propose a different perspective to the "coordination vs. core function" divide. For starters, the notion of function and purpose in biology is a heavily debated philosophical problem, for when one speaks of the function of some part of an organism, one cuts the system through a very subjective, preferential prism, always *post factum* ([32]). But the organism itself is a whole with a set of systemic properties and the function of a given part shifts depending on the perspective and interest of the observer. Frequently, individual parts are found to fulfill many functions at once.

J: But in an engineering endeavor function is the basis of the artifact and precedes it. Your point please?

C: Right. The point is that as the software artifact matures and makes its way into the world, it detaches from its original creator and intent. The detachment of the software artifact from the single programmer's idealistic, limited and restrictive cognitive realm is unavoidable.

J: What sort of detachment? That it gets appropriated by the programmer's peers and eventually applied in a different context? That is gets reused?

C: Kind of, yes. Note that "software reuse" is an oxymoron because it usually refers to artifacts being used *as is* over and over. A "reusable component" actually means "hopefully widely applicable component". But in reality reuse is a social phenomenon ([47], p.714), not a technical achievement and when it occurs it amounts to repurposing the artifact for something different than the original intent. This requires one type of management. Another type of management is exemplified by the need for fault tolerance, or performance monitoring and optimization. Yet another by the problem of resource utilization. Semantic dependencies and inconsistencies are all-pervasive and a huge management problem of their own. And so on and so forth. In sum, most of the aspects that make a component/process perform a given function are related to "external" management, an apparent overhead outside the scope of main intent.

J: Your point please.

C: My point is that I don't agree with my last sentence. **Management and regulation align the function of a component or a process to its context of operation.** From a systemic perspective they are constitutive of said function since they ensure the invariance of the systemic properties that permit us to attribute the function in the first place. Note that this resonates with my previous assertion that programmers are as much observers as enablers of the system they produce for it is preposterous that they ignore the operational context of their creation[4].

J: If I understand correctly, you are suggesting that there be no significant conceptual division between the functional role of a software piece, and everything around it that contributes to that role being fulfilled.

C: Indeed, this is precisely my point. Think of it as an argument against the black box dogma. On the other hand, it is less clear what the practical implications are. For one thing, I seriously doubt recent proposals ([17], [41]) that strive to "isolate and encapsulate" as black boxes the sorts of management aspects that I just outlined.

J: Surely, then, you must have an alternative strategy in mind. What you refer to as the black box dogma revolves around notions such as well-defined behavior, roles, responsibilities, preconditions, postconditions, input-output mappings, all things that make up, pretty much, all of programming. What makes you so uncomfortable with that?

C: Let me give you an example . Say you have a component X, and you want to make X robust. The black box strategy mandates that you make a component R such that $\forall x\ :\ R(x)\, is\, robust$. This, I contend, is impossible because *robustness* has nearly as many different meanings as there are Xs around (unless you define a robust program as one that has 99.99% uptime, which would be trivially useless). The approach suffers from all the shortcomings of the abstraction-modularization tandem we discussed before.

J: But, what else is there?

C: There is the age old trick of going meta - being able to talk **about** the system, do something **to** the system. I stated before that one of the main problems with abstractions in computer programs is that they are reified. Meta turns that into an advantage. To align the function of X to the context of operation C, one could align $meta(X)$ to $meta(C)$ instead.

J: Meaning?

C: Meaning that part of the software would bring together knowledge about the system and knowledge about its environment in order to align desired behavior and context. Meaning that meta-models should be part of the system, not external to it. Meaning that behavior should be *knowledge-driven*. The approach draws its strength from the fact that $meta(X)$ and $meta(C)$ are categorically the same and therefore comparable.

J: Ah, ok. You are proposing a knowledge-driven approach as the framework for tackling the management vs. function divide. Hmm, you know what? I just had an idea about another approach to evolving software.

C: Great! Would you be kind enough to share it with me?

J: Sure. Since you are talking about modeling, meta-models and relying on knowledge representation, why not synthesize programs *in silico* based on those models? Then let the programs run and only create variation between the models and select them based on the fitness of the programs they produce. Research on model-driven architectures, code generation, round-trip engineering and the like would bootstrap the process, but then the meta-models' translation mechanism could be evolved as well. This amounts to the following correspondence with evolution in nature:

$$\begin{aligned}&DNA \Longleftrightarrow (Meta-)Models\\&Phenotype \Longleftrightarrow Program\\&Growth\&Development \Longleftrightarrow Model\, Translation\end{aligned}$$

C: Sounds like a good idea! And in the same way the developmental process in living organisms is itself a DNA program, the mapping of knowledge models to programs is encoded as knowledge. But I have something else in mind, something closer to actual programming practice. I'm thinking of the concept of *light semantics* ([38]) proposed by Dewayne E. Perry and used in his work on the Inscape IDE, back in the 80s. Perry distinguishes the complexity of algorithms as mathematical creations from the complexity that *"...arises from the sheer wealth, or mass, of details"* where *comprehension is hindered by the problem of scale*. It is this latter type of complexity, which is the dominant one, that had you worried about the potential messiness of my proposal and it is this type of complexity that, according to Perry, calls for *a much lighter form of semantic approach...than full automated theorem proving but which goes beyond the current available forms of type checking*.

J: Dismissing my idea so quickly is impolite, but ok, interesting. So by knowledge-driven, you mean making use of some sort of 'light' semantics to assist computation.

C: I'm not excluding 'heavy' semantics for when the ends justify the cost. As I alluded before, performing a function should be seen as maintaining a systemic invariant across environment contexts. The idea is to keep just enough in-

[4] Except for the rare, privileged instances of closed domains.

formation about a software element that is needed for it to maintain function now. The information can be augmented on a need by need basis, through the evolutionary process. Furthermore, maintaining semantic information is needed both by people for documentation, refactoring, reuse as well as computers for validation, interpretation/execution, inference. Note that a rarely stated, implicit requirement in engineering is human understandability and system capabilities are sacrificed solely for that. Abstraction and modularization often serve mainly that requirement, but there are other ways for it to be fullfilled. And we are at one more characteristic of our evolutionary engineering platform:

J: And I suggest that we stop there because I'm hungry again.

C: So here it is:

Characteristic 4: A knowledge representation framework as part of the shared information representation.

In sum, knowledge representation and semantic information are part of the system. Knowledge about the system is maintained together with knowledge about its environment. Both knowledge and processes are evolved in the same way, through the same interface, obeying the same rules as all the familiar programming artifacts.

J: That puts an extra dimension on the problem of getting languages from multiple paradigms to share data.

C: I know. The schema for interoperability would call for a full-fledged, highly-structured, yet flexible common memory domain. And I am ready to talk about that, but let's postpone it for after dinner.

5. Getting Real

It is almost time to go back to bed now.

J: You are talking about a common memory domain within which languages can interoperate. I don't quite follow you. On one hand, computer memory as a linear sequence of cells is a perfectly valid common ground. On the other, each programming language has its own way of organizing memory that is tied to its semantics and particular implementation.

C: It's a real challenge, I admit, to find a better lowest denominator for organizing memory across languages and paradigms. And I suppose there could be several equally valuable proposals. My intuition tells me that just going one step beyond the hardware level, acknowledging a handful of fundamental operations in memory organization that pervade computer languages would already provide a better foundation, and by better I mean closer to cognitive software design processes, for interoperability between artifacts produced by different teams in different contexts.

J: The most fundamental principle of memory organization is linking two or more things together. This provides a basic means for aggregation. So the memory model must at minimum offer good support for associating entities. In addition, all important programming languages have a type system of some sorts and rely heavily on it. Types can be thought of as the first meta-level of semantic organization of raw data, and a required one. Further, the model should be general, flexible and extensible since it is to serve as a generic knowledge representation framework allowing arbitrary levels of semantic organization.

C: Indeed. What I'd propose can be characterized mathematically as a generalized hypergraph. A hypergraph is a graph where edges may point to more than two nodes. The generalization further allows edges to point to other edges. So edges and nodes are unified into the single notion of a hypergraph atom where each atom has an arity - the number of atoms it points to - which is a number ≥ 0. Thus atoms are interlinked in completely arbitrary ways, but linkage is explicit. This structure was invented and proposed as a cognitive model for artificial general intelligence by Ben Goerztel ([21]).

J: What about types? The Types!!

C: Each atom is typed and has a value as payload. Values and types are managed by a type system embedded within the hypergraph structure itself. However, the connection of an atom with the rest of hypergraph is independent of its type and value. Thus, typing and interconnection are two orthogonal pieces of semantic information that each atom carries. The type system itself is completely open and able to accomodate virtually all computer languages.

J: Hmm, tell me more about that? How can you really unify typing mechanisms accross languages?

C: Well, obviously there's no magic, but a good conceptual model will certainly help. In programming languages, one thinks of types in terms of computational constraints imposed on the data (e.g. the range of values a given data variable may assume). Nearly every typed language offers a type system based on a set of primitive types and means to build new ones out of the primitives. The complexity of a type system generally reflects the number of ways one can construct new types (unions, intersections, functions, records, inheritance) and the elements out of which those new types can be constructed. In polymorphic languages for instance, one is allowed to extend the means of building new types. Types can be parameterized over other types so that we end up with ”type constructors”, entities whose instances are concrete types much as data values are instances of types. We then have a set of predefined constructors: the one for records, the one for functions and perhaps some others, sometimes we also essentially have means to define ”custom constructors”. In general, neither programming languages, nor formal treatments go beyond this meta-level of type constructors to, for example, type-constructor-constructors. Mathematically, however there is no reason to stop at level 2, and go to a potentially infinite tower of types (see for example [43]).

Now, sub-typing seems to be usually always analyzed in an ad hoc manner in formal treatments of programming languages. First, there is a nice mathematical type formalism with type checking rules and type inference. Then subtyping is stipulated with special rules for each separate type constructor: for records, the sub-type relationship means something, for functions it means something else. But the rules are always tied to the idea of substitutability.

J: So there is no clear cut, definitive formal notion of a type that could be easily implemented as is, without being bound to a particular programming language or a family of such.

C: Well, Hindley-Milner's algorithm exemplifies such a notion that seems fairly standard, but even that would be too restrictive. The French logician Jean-Yves Girard very rightly points out that types can be seen as plugging instructions. A term of a given type T is both something that can be plugged somewhere as well as a plug with free, typed variables ([20]). This is an operational view very much in the spirit of B. Russell's original type theory whose motivation was that of formal (i.e. syntactic) constructability. I think this is as far as one can go in unifying what types really are in computing.

J: So what do you propose?

C: That types be simply hypergraph atoms with a minimal set of operations for managing the persistence of values in their range and implementing a substitutability predicate for the values that they manage. Then type constructors are just types whose values are types and type constructor constructors are possible at as many levels as one wishes. Inheritance is derived from substitutibility as well.

J: I still don't see how different languages will work with that.

C: Each will need to bootstrap with a set of predefined types. Then, more complex types are simply hypergraph atoms with full information for their runtime reconstruction available. Some of those predefined types will be what we know as primitive types, i.e. numbers and the like. Some will be type constructors that can create a runtime instance of a type based on structured information in the hypergraph. Thus, each language will have a different runtime binding for a given type, but they all share the same storage layout for values. Incidentally, this architecture opens the door for pluggable type systems ([8],[2]) since several different type schemas can be overlaid on the same graph of values.

J: That sounds like a very elaborate organization of data. How does addressing work?

C: Each atom is referred to by a globally unique and persistent handle. At runtime, however, speedier handles can replace the persistent ones. Linkage in the graph relies on handles. Values and types and raw data are all identified with unique, persistent handles. Here is an explicit definition of the structure in a grammar-like format:

$$Atom \to (Type, Value, TargetSet)$$
$$TargetSet \to (Target1, Target2, \ldots, TargetN)$$
$$Type \to Atom$$
$$Value \to (Part1, Part2, ...)$$
$$Value \to Raw\ data$$

J: So support for this would include searching, traversing the graph, perhaps some pattern matching of graph structures, handling of semantic relationships between entities?

C: Yes, but think of it as a memory model. At the implementation level it is simply a very flexible, pluggable, language and platform neutral database for storing hypergraphs. But conceptually I see it as a replacement of the random access memory as the foundation for program interoperation.

As I mentioned, it originates as the data structure of a model for cognition and it is very suitable as a generic knowledge representation framework within which management of an artifact at all levels of granularity can be implemented. In other words, this hypergraph connectivity + types model merges the attempt to model cognition at an abstract representational level with the concrete data management needs of mundane programming!

A key aspect of hypergraph as a way of organizing data is that a single atom in the graph can carry a huge semantic import in virtue of its connectivity role while another atom can simply hold a specific value suitable for a single user.

J: Can you give me some examples of atoms? Perhaps that'll help me get a better picture of where you are going.

C: A string is an atom. A named record is an atom. The source code of a class is an atom as is the compiled class itself, perhaps related by a *source-of* relationship. A closure is an atom. A document is an atom. A script in a dynamic language is another atom. A function, an inference rule, a predicate, a predicate logic fact, the relationship between several entities are all an atoms. The type of a function is an atom. A UI live component is an atom. The relationship of a UI component as the viewer of instances of a type atom is also an atom. An instance of a class, the result of a function as applied to a set of atoms is also an atom. A full-blown user interface is an atom. The set of keybindings for a particular UI component is an atom.

J: This looks like yet another "everything is an X" conception, I must confess that I'm rather suspicious of those. In fact, you started with a requirement to explicitly avoid this trap in your characteristic 3!

C: Think of it instead as "nothing is not (conceivably) an X" conception. Or equivalently, "everything can be an X". An important shift is in escaping what some have referred to as the *tyranny of objects*.[5] Thus while you could record every single object or class or function as an atom in the hypergraph, you are not required to do so. You could have a set of those stored in a *module atom*, perhaps in source

[5] Sadly, the author does not remember where he got that locution from. Help in tracking down the originator will be appreciated.

code form, and have semantic dependency relationships of other entities on this atom. I do not think that any meta-model of programming whatsoever should be imposed, for a meta-model is a model and models should be malleable, replaceable and emerging during the software construction activity. This is a fundamental flaw in proposals inspired by similar considerations as we have here such as [24]. What an evolutionary computing platform should provide is facilities and encouragement for **the process** of evolving and sharing software artifacts at granularity level N, where N is a natural number.

J: Aham. Well, even though this fog of interlinked atoms begins to clear up, I still can't get a handle on how programs are constructed.

C: It depends on what your handle is on programs themselves. If you need an executable file, then forget it. An executable file is a packaging convenience which can be dispensed with. Think more in terms of processes rather than programs. Recent work from this perspective sheds some light on the kind of execution model envisioned ([13], [12], [37]). All those works share the desire to eliminate the compile-time vs. runtime distinction which is a promising venue. But the benefits of eliminating this distinction can be achieved without abandoning familiar software practices altogether. A good way to understand my proposal can be found in the work of Yinliang Zhao and termed, rather appropriately, *granule-oriented programming*: *an evolvement metaphor in which programs are "ground" into code ingredients in order to localize unfitting parts of a program as explicitly as possible, and then "compound" them into granular output code* ([50]). In a simple, practical setup, processes are composed out of executable atoms (the ones whose value is code, compiled or in some programming language). In a more refined setup, execution can be driven by low-level representation entirely based on the graph of atoms where atoms would represent operators and operands for example. In fact, a portion of the hypergraph can be viewed as a combinatory logic expression and executed by applying graph reduction techniques from functional programming. The level at which computation is represented in the graph is left at the programmer's discretion. The crucial aspect is that the composition of processes out of atoms is knowledge-driven.

J: What do you mean by composition being knowledge-driven?

C: Putting things together in some sort of syntactic/semantic relationship can be done in many different ways, either visually or textually by using a programming language. But ultimately that's irrelevant. What's relevant is the representation (in the hypergraph) of the composition as knowledge. Now, knowledge representation is often thought of as "semantics", but it is purely syntactic in its raw form. Semantics comes from the enactment of this knowledge as a computational process. In my view, $semantics \equiv computation$. Thus by knowledge-driven I mean, enacted through the interpretation of formalized knowledge. This is a defining concept within the logic programming paradigm and can be thought of as generalized data-driven programming.

J: Yes, in logic programming computation is driven by formal rules and facts expressed in some logical formalism. So you are proposing that program construction be defined through logical rules?

C: Again, that would amount to postulating a meta-model and a method for computation. And we are against method, aren't we? Rather, there is no construction *per se*. There is an encoding of knowledge about atoms, itself being represented as atoms (remember, in the generalized hypergraph edges can point to edges as well) and there are means to enact that knowledge resulting in a runtime process. The interpretation may come in the form of rules (inference rules, rewrite rules, whatever), but also as imperative constructs or as event-based, reactive triggers.

J: Interesting... when you are speaking of formal knowledge about software artifacts, I'm thinking mostly in terms of verification, validation, consistency enforcement. And your model doesn't seem to preclude those engineering "goodies". I'm reminded of the long forgotten work of Dewayne E. Perry's that you mentioned before dinner where the programming environment continuously enforces consistency based on declared semantic constraints on procedures and data ([39], [40]).

C: Exactly. There is no *a priori* reason for a live, prototyping environment to forbid static validation of constructs.

J (smiling with complicity): The validation process being itself created within the same paradigm, I presume.

C: But of course, my dear. Software construction and computation are interwoven as one and the same, and so as to not lose all we've learned in more than half a century of software engineering practice.

J: You know, Graham Nelson has created the wonderful IF[6] Inform 7 environment ([36]). There, fictional stories are created as computer programs. All this conversation makes me think that programs should be seen as fiction stories as well. So what he says about Inform 7 becomes of general relevancy:

> ... I suggest that the activity of programming IF is a form of dialogue between programmer and computer to reach a state with which both are content, and that it is not unlike the activity of playing IF, also a continuing dialogue in which the computer rejects much of what the user tries.

C(smiling with content): I guess that summarizes it beautifully.

J: Well, to be honest I'm getting quite tired already. Perhaps we could continue tomorrow, but before that I'd like to hear how you envision two non-trivial aspects of your evolutionary computing schema - variation and selection.

[6] Interactive Fiction.

I'm guessing that the population of individuals are those hypergraph atoms.

C: That's correct.

J: Then each user is working with their own instance of the hypergraph. And each instance is connected to other instances so that atoms can be shared, replicated or distributed in a peer-to-peer fashion.

C: Yes, your understanding is right. Clusters of hypergraphs may exist within a single organization, or span geographic locations and serve as a collaborative medium for teams and programmers. A user may be connected to such a cluster solely for consumption purposes without ever participating in code.

J: I see. Now, I'm guessing variation in the population of atoms is created by programmers.

C: Yes, or by more sophisticated users! The simple act of configuring a certain aspect of the system in some context is a valuable contribution to the whole.

J: Sure. Now, tell me what varies. More specifically, how does an atom vary?

C: Recall that an atom is essentially a triple of (type, value, target set). All three constitutive elements are allowed to change while the atom preserves its identity (because it is identified by the same handle) so that links pointing to it remain valid. Thus an atom can simply have its value replaced with something else (note, however, that values themselves are immutable) and/or it's type be replaced by a more appropriate one. Or, it can be redirected to link a different set of target atoms. Each such variation normally changes some aspect of the behavior of the system.

J: So consistency would be enforced through the subsumes predicate that you mentioned before?

C: Right. But note that when an atom is not modified in isolation, dependencies can be easily tracked since they would normally be explicit in the graph structure. In such cases, variation would be at the sub-graph level. When atom types change, approaches such as [14] could assist.

J: I see. But there are risks in such a setup, for when a dependency is not directly represented in the graph, inforeseeable inconsistencies will occur.

C: Indeed they will. In that case the change will break and it will have to be reverted back or fixed *in situ*.

J: Hmm, fair enough. In that case, it must be possible to backtrack to older versions. I assume change history must be kept. Then how are atom versions identified?

C: They are not. There is a distinction between modifying an atom and updating it from another hypergraph location. Modifications are not considered special. Updates on the other hand are, and bear a unique identifier. When an atom is updated, its current triple is recorded as a different atom (i.e. under a different handle). The new an old version are linked and tagged with the update. When a whole sub-graph is updated, all atoms are tagged with the same update identifier. An update can thus be rolled back regardless of how many atoms it is comprised of.

J: Sounds like a sensible approach. Now, I'm guessing that selection happens when a certain variation of an atom gets replicated widely in the network of hypergraphs.

C: Yes, and it doesn't have to be replicated everywhere. Different versions can exist in parallel and evolve separately at different locations.

J: That's understood. However, I'm curious as to what exactly triggers an update from one location to another? Is it random? Is it explicitly requested by the user?

C: That's the tricky part! I'm guessing that different negotiation protocols could be implemented. For instance, a team closely working together towards a single definite goal would like synchronization to be almost automatic with the relevant part of the hypergraph identically replicated everywhere. On the other hand, if a user is connected to a large peer-to-peer network composed of many strangers they would want strict isolation and manual triggering of updates. Policies for proposing updates based on previous history, or some measure of similarity between two parties can be implemented. It's an interesting problem. And perhaps it should be solved through evolution within the same platform! All hackers are welcome!

Josefina yawns and her glassy, red eyes give her the slightly dreamy look so typical of her. Meanwhile, Chewbacca keeps staring at her in expectation of an eventual reaction. Josefina heads for her "lair" slowly, but stops and looks back halfway one last time before turning in.

J: Dear friend, this is all very promising, I will sleep on it for many nights to come and I hope that we can continue the discussion. One last question, if I may?

C: Shoot!

J: How would this type of open environment work as a market for software?

C: Hmm, don't know. You tell me.

6. Epilog

The main point in the discussion above is that to take software to the next level of complexity, what is *mostly* needed is not a new and better set of abstractions, not a library that models some seemingly relevant aspects of living systems, but the re-enactment of the process by which living systems come to be. This doesn't require abandoning "software as we know it". There is nothing wrong with it. It doesn't require a software revolution, but the evolution of software.

An implementation of the architecture outlined in the above dialog is currently under development on the Java platform. We have made great emphasis on the practical aspect of the environment and its usability for day to day work. The hypergraph memory model implementation is at a very advanced stage, in the form of a free, open-source product called HyperGraphDB ([27]). In addition to serving as a persistent layer of live objects and knowledge representation for the evolutionary computing platform endeavour, it is heavily being used in semantic web projects. HyperGraphDB is accessible through a Java implementation, but the storage layout is independent of Java and a C++ access layer is planned. At the time of this writing, HyperGraphDB is not yet distributed. A programming environment called Scriba ([28]) holds sway of the interactive computing live system. At the current stage, Scriba is a scripting environment integrating languages implemented on the Java platform. The integration is loosely based on the JSR 223 specification, to which some extensions are being experimented with. Three paradigmatic languages have been incorporated: BeanShell[7], JScheme[26] and JLog[25]. Parts of the current user interface are “hard-coded” as a Java application, a bootstrapping step towards an environment implemented entirely within the proposed platform. It is inspired from the Mathematica system ([33]) where the REPL is replaced with nested cell structures, each cell holding anything from an expression to a live Java component; it is also free, open-sourced and currently in use for prototyping and testing Java and J2EE application.

Acknowledgments

Thanks to Assia Alexandrova, Yaneer Bar-Yam, Ben Goertzel and Juliette Colinas for their numerous constructive comments on earlier drafts of this paper.

References

[1] Ashby, Ross W., ”An Introduction to Cybernetics”, Chapman & Hall, London, 1956

[2] Andreae, Chris, Noble, James, Markstrum, Shane, Millstein, Todd, ”A Framework for Implementing Pluggable Type Systems”, Proceedings of the ACM Conference on Object-Oriented Programming, Systems, Languages and Applications, Portlan, Oregon, 2006

[3] Bar-Yam,Y., ”Dynamics of Complex Systems”, pp. 752-757, Perseus, Reading, MA, 1997

[4] Bar-Yam, Y., ”Large Scale Engineering and Evolutionary Change: Useful Concepts for Implementation of FORCEnet”, Report to Chief of Naval Operations Strategic Studies Group, 2002, available at http://necsi.org/projects/yaneer/SSG_NECSI_2_E3_2.pdf

[5] Bar-Yam, Y., ”When Systems Engineering Fails — Toward Complex Systems Engineering”, International Conference on Systems, Man & Cybernetics Vol. 2, 2021-2028, IEEE Press, Piscataway, NJ, 2003.

[6] Bar-Yam, Y., ”Unifying Principles in Complex Systems, in Converging Technology (NBIC) for Improving Human Performance”, M. C. Roco and W.S. Bainbridge eds, Kluwer, 2003.

[7] http://www.beanshell.org

[8] Bracha, Gilad, ”Pluggable Type Systems”, Proceedings of the ACM Conference on Object-Oriented Programming, Systems, Languages and Applications, Vancouver, British Columbia, Canada, 2004

[9] Braha, Dan, Minai, Ali, Bar-Yam, Yaneer (Eds.) ”Complex Engineered Systems: Science Meets Technology”, Springer, 2006

[10] Brooks, Frederick, ”No Silver Bullet: Essence and Accidents of Software Engineering”, Information Processing, 1986

[11] Colinas, J., ”Non-coding DNA sequences and gene regulation”, Ph.D. thesis, Duke University, 2006

[12] Edwards, Jonathan, ”Subtext: Uncovering the Simplicity of Programming”, Proceedings of the ACM Conference on Object-Oriented Programming, Systems, Languages and Applications, San Diego, California, USA, 2005

[13] Elliott, Conal, ”Functional Programming by Interacting with Tangible Values”, draft paper under revision, available at http://conal.net/papers/Eros

[14] Evans, Huw, Dickman, Peter, ”Zones, Contracts and Absorbing Change: An Approach to Software Evolution”, Proceedings of the ACM Conference on Object-Oriented Programming, Systems, Languages and Applications, Denver, Colorado, USA, 1999

[15] Evens, Aden, ”Object-Oriented Ontology, or Programming’s Creative Fold”, Angelaki, Volume II, number I, April 2006

[16] The FEAST (Feedback, Evolution And Software Technology) projects, http://www.doc.ic.ac.uk/ mml/feast/

[17] Gabriel, Richard P., Goldman, Ron, ”Conscientious Software”, Proceedings of the ACM Conference on Object-Oriented Programming, Systems, Languages and Applications, Portlan, Oregon, 2006

[18] Chaitin, G.J., ”Algorithmic Information Theory”, Cambridge University Press, 1987.

[19] Chaitin, G.J., ”Information, Randomness & Incompleteness, 2nd edition”, World Scientific, 1990

[20] Girard, Jean-Yves, Lafont, Yves, Taylor, Paul, ”Proofs and Types”, Cambridge University Press, 1989

[21] Goertzel, Ben, ”Patterns, Hypergraphs & Embodied General Intelligence”, IEEE World Congress on Computational Intelligence, Vancouver, BC, Canada, 2006

[22] Goldberg, A., Robinson, D., ”Smalltalk-80: The Language and Its Implementation”, Addison-Wesley, 1993

[23] Holland, John H., ”The Hidden Order”, Addison Wesley Publishing Company, 1996

[24] Imbusch, Oliver, Langhammer, Frank, von Walter, Guido, ”Ercatons and Organic Programming: Say Good-Bye to Planned Economy”, Proceedings of the ACM Conference on Object-Oriented Programming, Systems, Languages and Applications, San Diego, California, USA, 2005

[25] http://jlogic.sourceforge.net/

[26] http://jscheme.sourceforge.net/jscheme/main.html

[27] http://www.kobrix.com/hgdb.jsp

[28] http://www.kobrix.com/scriba.jsp

[29] Kolmogorov, A.N., ”Selected Works, Volume III: Information

Theory and the Theory of Algorithms (Mathematics and Its Applications)", A.N. Shiryayev, ed., Kluwer, Dordrecht, 1987

[30] Lehman, M M, "Laws of Software Evolution Revisited", Software Process Technology - Proceedings of the 5 th European Workshop, pages 108–124, Nancy, France, October 1996. Springer-Verlag. Lecture Notes in Computer Science 1149.

[31] Lehman, M M, Ramil, J F, "Evolution in Software and Related Areas", ACM Proceedings of the 4th International Workshop on Principles of Software Evolution, Vienna, Austria, 2001

[32] Lewens, Tim, "Organisms and Artifacts", The MIT Press, 2004

[33] http://www.wolfram.com/products/mathematica/index.html

[34] Minsky, Marvin, "A Framework for Representing Knowledge", MIT, AI Memo 306, 1974, available at http://hdl.handle.net/1721.1/6089

[35] Noble, James, Biddle, Robert, "Notes on Postmodern Programming", OOPSLA 2002, Onward! track

[36] Nelson, Graham, "Natural Language, Semantic Analysis and Interactive Fiction", Inform 7 white paper, available at http://www.inform-fiction.org/I7Downloads/Documents/WhitePaper.pdf

[37] Perera, Roly, Foster, Jeff, György, Koch, "A Delta-Driven Execution Model for Semantic Computing", Proceedings of the ACM Conference on Object-Oriented Programming, Systems, Languages and Applications, San Diego, California, USA, 2005

[38] Perry, Dewayne E., "Software Evolution and 'Light' Semantics", Proceedings of the ACM Internation Conference on Software Engineering, Los Angeles, California, USA, 1999

[39] Perry, Dewayne E., "The Logic of Propagation in the Inscape Environment", Proceedings of SIGSOFT '89: Testing, Analysis and Verification Symposium, Key West, FL, December 1989

[40] Perry, Dewayne E., "Version Control in the Inscape Environment", Proceedings of the 9th International Conference on Software Engineering, (IEEE Computer Society Press), p142-p149, 1987

[41] Rinard, Martin, "Acceptibility-Oriented Computing", Proceedings of the ACM Conference on Object-Oriented Programming, Systems, Languages and Applications, Anaheim, California, USA, 2003

[42] Ungar, David, Smith, Randall, "Self: The Power of Simplicity", Proceedings of the ACM Conference on Object-Oriented Programming, Systems, Languages and Applications, Orlando, Florida, USA, 1987

[43] Sheard, Tim, "Languages of the Future", Proceedings of the ACM Conference on Object-Oriented Programming, Systems, Languages and Applications, Vancouver, British Columbia, Canada, 2004

[44] Smith, Brian Cantwell, "On the Origin of Objects", The MIT Press, 1996

[45] Squeak Smalltalk environment, available at http://www.squeak.org

[46] Solomonoff, R. J.,Information and Control,7,1,(1964)

[47] Stroustrup, Bjarne, "The C++ Programming Language", 3nd edition, Addison-Wesley, 1997

[48] Surowiecki, James, "The Wisdom of Crowds", Anchor Books, 2005

[49] Wagner, Andreas, "Robustness and Evolvability in Living Systems", Princeton University Press, 2005

[50] Zhao, Yinliang, "Granule-Oriented Programming", Proceedings of the ACM Conference on Object-Oriented Programming, Systems, Languages and Applications, Vancouver, British Columbia, Canada, 2004