Rapid Software Evolution

Borislav Iordanov
Kobrix Software Inc.
biordanov@acm.org

1 Introduction

The complexity of modern software systems, described more than twenty years ago in the well-know paper by Fred Brooks (1) has become proverbial amongst practitioners. While the software engineering community has accepted and learned to cope, albeit in a limited way, with what Brooks termed *essential difficulties*, i.e. intractable obstacles due to the nature of software itself, there is a consensus that the ultra large-scale systems of the future call for a fundamental change in our understanding and practice of software construction (2).

In this paper, we present a new approach to software development, based on the idea of evolutionary engineering (3; 4) and describe a concrete platform facilitating its implementation. First we outline the essential high-level features of our proposal - a live, multi-paradigm, knowledge-driven, distributed network of operating environments where programmers and users interact within the same environment on a large scale. Next we describe the distributed memory architecture at the foundation of the platform - a generalized hypergraph data structure with a universal representation schema and discuss some of the low and high-level evolutionary dynamics based on it. Finally, we illustrate some social ramifications should such a platform be adopted.

2 Evolving Software

In recent years parallels between the living world and software programs have become recurrent. However, the practice of software development remains the fruit of historical accidents, largely untouched since the early foundational days of the field, when the theoretical focus was chiefly on algorithmic efficiency and the practice of

artifact construction inspired by industrial engineering processes that mandate a substantial amount of forethought due to the prohibitive costs of design flaws. Attacking the complexity of large projects through divide and conquer methods such as abstraction and modularity is the *modus operandi* of the engineering community. Even when inspired from living systems, researchers usually focus on defining the proper abstractions and modularization boundaries (5; 6; 7). This is only natural since engineering tasks are tackled top-down, starting from the problem and sub-dividing it into simpler ones. Unfortunately, the strategy does not scale as argued extensively in (3). Software systems suffer from the rigidity of their abstractions and from the unpredictable, non-linear interactions between their modularized components. What can be done?

When faced with the task of creating highly complex systems where top-down decomposition is not possible, one realizes a basic contradiction: emergent behavior is by definition unpredictable and therefore contrary to the engineering mandate of building systems with well-defined function. How does one achieve goal-directed emergence? The answer is through biological evolution. Classical engineering achieved its success by turning a descriptive theory about the world (physics) into a prescriptive tool. Similarly, we expect the theory of evolution to yield successful emergent systems when applied in a prescriptive engineering context.

Goal-directed evolution presupposes a fitness function, a way to measure the relative merits of organisms. It has been successfully applied at a low-resolution scale, where the fitness function is some easily computable numerical quantity, in the case of genetic algorithms. However, evolving large end-user programs from bit strings would be clearly impractical. At the opposite spectrum we have a market-based ecosystem of software programs, each grown by following a classical engineering lifecycle. There the fitness function is defined by collective human feedback and the process works, but with significant shortcomings: first, there must be an actual market for a particular application domain (frequently not the case for highly specialized, commissioned projects); second, the evolutionary time-scale is large (actually a function of the complexity of the software); third, the granularity is too coarse (we frequently witness the disappearance of high-quality features when the whole program loses the market battle).

A more promising approach to software evolution would frame it as a middle ground, between the above two extremes, and within its natural habitat: the cognitive landscape of human knowledge, interaction and information exchange. In other words, the right scale of evolutionary units is the one at which human cognition operates, the level of abstraction permitted by current technology and where entities can be recombined in a meaningful way. This could mean a single programming unit such as a data type or a procedure, or a piece of data upon which behavior is based in a clearly recognizable way. It is primarily humans who make up the environment where computer programs evolve and therefore all forms of human participation, from the hard-core programmer to the neophyte end-user, should be woven together into the same computing medium.

The sharing of the same interactive medium between consumers and producers of software is a key point if feedback must operate at fine granularity. Programs are usually conceived as standalone executable files compiled out of source code

representation in some high-level language. The abstract models present in the source code are lost during the compilation process and much engineering effort is spent in categorizing software behavioral decisions into *design time*, *compile time*, *deployment time*, *and run time*. However, alternative models where the compile time vs. run-time distinction is blurred have emerged over the years, the most notable perhaps being one of the first object-oriented languages Smalltalk (8) and its modern reincarnation Squeak (9). Another prominent example is the Self environment (10). Those systems are termed "live" systems because they are entirely made up of persistent live objects, completely exposed and modifiable by the user in the very form in which they were originally created. Our platform falls in the category of live systems, but with several essential differences.

First, we avoid tying the programming task to any meta-model, programming paradigm or language. Rationale: programmers with diverse backgrounds should be able to participate; software problems yield themselves better to one paradigm or another; diversity is good.

Second, in order for fine grained software artifacts to replicate, diversify, be selected for or against, the platform is distributed in nature, yielding a decentralized network of such live systems.

Third, knowledge representation capabilities are natively incorporated for the following main reason. In classical monolithic systems, the gluing of components is to a large extent based on the programmer's hidden intent behind each part (including, but not limited to the so called "implementation details" that software designers try to make irrelevant). In an evolutionary system where much variation and recombination of sub-components is common and where, in addition, such variation is to be primarily induced by contextual particularities of the environment, semantic metadata about both artifacts and environment is of essence for management features such as compatibility/applicability checks, exception handling and the like.

Finally, we note that human cognition is capable of operating at different resolution scales, depending on context. Therefore, the units of evolution should span different organization levels as well. A crucial component of the platform, allowing it to meet all of the above requirements, is a highly flexible, structured, distributed memory model which we now describe.

3 HyperGraph Structured Memory Domain

The memory model is based on the most fundamental principle of organization - aggregating two or more entities. In formal terms, it is 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. Edges and nodes are thus 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. An atom with arity 0 is called a *node* while an atom with arity > 0 is called a *link*. The atoms a link points to are called its *target set*. This structure was invented and proposed as a cognitive model for artificial general intelligence by Ben Goerztel (11). In addition, atoms are typed and carry a value as a payload. However, the connection of an atom with the rest of

hypergraph is independent of its type and value. The type system itself is embedded in the hypergraph structure and it is completely open and able to accommodate virtually all computer languages. Each type is also a hypergraph atom that is typed in turn. Types of types are called *type constructors*. The system is bootstrapped with a set predefined types and type constructors from which other types are built and evolved. The whole storage model is open and it is based on the following organizational schema:

Atom	→	Type Value TargetSet
TargetSet	→	Target1 Target2 . . . TargetN
Type	→	Atom
Value	→	Part1 Part2 ...
Value	→	RawData

Each element in the above grammar except RawData, which is simply a sequence of bytes, is a UUID (Universally Unique Identifier). This ensures a globally unique identity of all hypergraph elements and provides an universal addressing schema, thereby allowing the memory to be distributed into a decentralized network of local environments.

By convention, atoms are mutable while their values are not. This means that an atom can have its value replaced, or its target set modified while still preserving its identity and therefore the network structure of the hypergraph. There's no preset level of granularity for hypergraph atoms. Uniformity of the data representation schema has proven of essence in many systems, and we stick to that principle. Atoms range from simply-valued (e.g. strings, numbers) to complex records, self-contained user interface components, logical terms (e.g. represented as links between sub-terms), any kind of executable (in source or compiled form) or documentation resource[1]. Therefore, all those familiar software artifacts share the same representation medium and can therefore freely refer to each other opening the door for arbitrary layers and levels of organization. For instance, semantic information about executable entities is readily available to a run-time environment and interpreted to enforce constraints and manage dependencies.

In the following section, we illustrate how the hypergraph memory domain foundation is used to implement an evolutionary engineering software platform.

4 Dynamics of Evolutionary Software Construction

2.1 Overview

The platform is based on the idea of a large number of interconnected operating environments, which we call *niches*, linking together end users and developers within a single computational medium. A niche is bound to a single hypergraph instance. The environment is bootstrapped as a standard application based on the current user's configuration (essentially from where the user left of during the last run). Each

[1] It is not hard to imagine how all those examples can be represented with the general representation schema above. We omit such details for brevity.

instance is connected to other instances so that hypergraph atoms can be shared, replicated or distributed in a peer-to-peer fashion. Clusters of hypergraphs may exist within a single organization, or span geographic locations and serve as a collaborative medium for teams and programmers. We expect the topology of the niche network to resemble a scaled-down version of the topology of the internet.

The units of evolution are atoms. It is atoms that get replicated, vary and are selected for or against. Because an atom is a handle to an entity at any level of organization, the granularity of the elements of evolution is not mandated by the platform. Instead, it is contextual for a specific application, development task and/or organization. Selective pressures operate in two ways:

1. On atoms, where participants decide what atoms to keep in their niche. For instance, programmers decide what software components they need while end-users decide on "end-user" software artifacts, or the system itself decides to eliminate atoms that are no longer in use.
2. On an atoms' informational import (i.e. its type, value and target set). Recall that atoms always keep their identity while their informational import can vary between niches. This mechanism is important for establishing a boundary between a reference (the atom UUID) and its exact interpretation which is allowed to vary while still preserving referential integrity.

A user may be connected to such a cluster solely for consumption purposes without ever participating in variation of the population of atoms as created by programmers. On the other hand, more sophisticated users may induce variation by modifying data that drives behavior, for example by changing a significant "configuration setting" or encoding expert knowledge in some declarative logical form or creating a macro command.

2.2 Low-Level Dynamics

We now describe a few low-level scenarios of atom replication and variation. Assume a peer-to-peer communication layer between two given niches N and M and suppose that N promotes/publishes/offers an atom to M. This publishing can be triggered manually or automatically through some preset negotiation protocol, or simply be part of a larger update process. The atom may encompass functionality at any level - from the background color of a window to a word processing program. If the atom doesn't exist at M, it is simply transmitted and it's up to N to keep it or remove it at a later time. If the atom exists at M and it has the same informational import, nothing is done. If, on the other hand, the atom exists at M and has a different informational import, it is updated recursively by updating its type and target set. The old version is preserved under different UUID and tagged (via a hypergraph link) as being overwritten by this particular update. Thus an update can be completely reversed in the future - selected against.

What kind of variation of a given atom may we expected? The simplest case is when its value has changed. A more elaborate one is a type change since it may result in a different run-time instance of the atom. In this case approaches such as (12) can be adopted. But atoms in the target set can change as well. For example, one of the mechanism responsible for the efficiency of the evolutionary process in the

living world is sexual reproduction. This gives rise to generational variation through recombination (crossover) of existing characteristics. Hypergraph atoms exhibit such phenomena by mutating their target sets. For instance, suppose an atom A has type T, value V and a target set consisting of atoms X, Y and Z:

$$A \rightarrow T, V, X, Y, Z$$

For concreteness, one may imagine that A is a software module and X, Y and Z are sub-modules. Suppose further that at a certain point in time A is identical across a topology of three niches N_1, N_2 and N_3 where N_1 is connected to both N_2 and N_3, but N_2 and N_3 are not connected. Now, eventually Y turns into Y' at N_2 and Z turns into Z' at N_3. The niche N_1 can then acquire both changes sequentially thus recombining the two versions - a crossover effect. It may turn out that when both changes are combined, this has an adverse overall effect on the function of A in which case one, one or both updates can be rolled back. If, on the other hand, all goes well, N_1 may redistribute its aggregated version back to N_2 and N_3 which may in turn reverse the change because it fails within their context, or propagate it further.

The nature of the representation in the above example was essential for the possibility of crossover variation. For this reason, programmers are encouraged, but not forced, to create representations that rely on the hypergraph structure. Such representations may evolve, of course.

Note that those scenarios don't assume human communication. When a target niche is updating an existing functionality and it has a battery of automated tests to ensure quality and/or measure improvement, no human involvement is necessary.

2.3 High-Level Dynamics

At a larger scale, we note two important aspects of evolutionary dynamics. The first is stability vs. chaos. Given highly structured representations where dependencies are explicit in the graph structure, the system can be very helpful in tracking and resolving them. However, unpredictable inconsistencies can occur due to atom inter-dependencies that are not explicitly represented in the graph. And here lies one of the main difficulties: many failures in software are latent, subtle and the result of side-effects. This leads to (what is perceived as) chaotic behavior. The proposed platform does not attempt to solve the complexity problem by encoding every possible dependency between entities in a graph structure. Rather, it makes the process of parallel exploration of possibilities more efficient, in part by allowing for failure, experimentation and improvement at a fine-granular level. In this context, various niches assume roles similar to current development/deployment processes. For instance, in a mission/business critical environment, a "staging" niche would adopt a change before it is broadcast to "production" niches while a cluster of development niches will be highly fluctuating. A complementary aspect to niche role partitioning are atom update negotiation protocols which range from automatic synchronization to controlled, manually triggered requests. Stability then emerges at the local level out of social pressures (usage, tolerance) as well as computational resources constraints.

The second aspect is uniformization vs. diversification. Uniformization is the result of such pressures as predictability - one wants to know what to expect from a piece of software, common understanding - one wants to be able to share information such as training materials and costs reduction - cheaper to reuse something existing than building anew. Diversification is the result of adaptation to a context of use where context includes thing like business processes, individual preference, computational resources available. Uniformization and diversification occur at all levels of organization. For example, there's little reason for different versions of a sorting routine while there are good reasons for different implementations of an abstract data type (such as a key-value lookup structure) that perform better depending on the nature of the data. Similarly, there's little reason for the implementation of a pull-down menu component to vary from system to system, but there is good reason to vary the implementation of a command navigation system in the context of users with disabilities.

5 Social Implications

The long-term vision behind our effort is a large network of niches connecting participants with any conceivable software needs and technical knowledge. A sweeping fine-grained decentralization of the production of software artifacts would certainly alter its economic dynamics. The current open-source model of a large body of freely available software with a service economy around it may well serve as a precursor, albeit a mediocre one. Open-source software is still centralized and it follows standard engineering practices. In addition, code ownership remains the *de facto* reality. By contrast, our proposal entails the eventual abundance of interconnected fine-grained information artifacts with collective ownership.

The live aspect of the platform encourages more active participation from traditional end-users. Enforced or emergent variability within the software artifact space breaks the traditional information sharing about and trust in the predictability of immutable monolithic programs with identical behavior everywhere. One consequence could be the birth of localized services or artisans offering technical expertise in niche contextualization and absorbing the adverse effects of failed evolutionary experiments.

As a more down to earth example, consider the widespread software business model where a company licenses a product and offers customization services. The internal organization of such companies usually follows a pattern where the product is developed by a core team of highly skilled engineers with satellite teams providing solutions in the form of a set of services and customizations to different clients. This organization can be depicted in Figure 1. The product team interacts with several solutions teams and each solution team interacts with several customers in what amounts to a strict hierarchical organization. All customers share the same core product capabilities. Customizations are possible only when variability has been explicitly introduced upfront. When this is not the case, customer demands drive a core future to be detached as customizable. In fact, there's pressure for the core product to incorporate enough variability so as to meet everybody's needs, thereby

augmenting its complexity and creating the bottleneck so characteristic of other hierarchical structures. By contrast, in an evolutionary platform there is no core product at all (see Figure 2). The company instead maintains a cluster of niches, each loosely adapted to particular customers. Solution teams form a decentralized collaboration network where each manages a niche tailored towards specific business needs. The variety of the company's business environment is much better represented in the new arrangement as suggested by the Law of Requisite Variety (13).

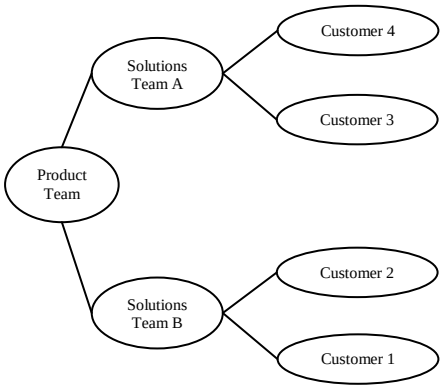

Figure 1

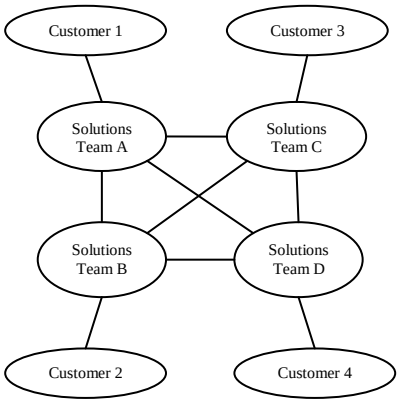

Figure 2

6 Final Remarks

In this work, our main assumption has been that the key to mastering the complexity of the software systems of the future is not the "right" set of abstractions, for it is their endless deluge that generates the complexity in the first place. Rather, it is in the process of goal-directed emergence as enacted through evolutionary engineering. Our ongoing work on the hypergraph memory domain and run-time environment for the platform (both free and open-source) can be found at http://www.kobrix.com.

7 Bibliography

1. **Brooks, Frederick.** *No Silver Bullet: Essence and Accidents of Software Engineering.* s.l. : Information Processing, 1986.
2. **Northrop, Linda et al.** *Ultra-Large-Scale Systems.* s.l. : Software Engineering Institute, Carnegie-Mellon, 2006.
3. **Bar-Yam, Yaneer.** *Large Scale Engineering and Evolutionary Change: Useful Concepts for Implementation of FORCEnet.* s.l. : Report to Chief of Naval Operations Strategic Studies Group, 2002. available at http://necsi.org/projects/yaneer/SSG NECSI 2 E3 2.pdf.

4. *When Systems Engineering Fails — Toward Complex Systems Engineering.* **Bar-Yam, Yaneer.** Piscataway, NJ : IEEE Press, 2003. International Conference on Systems, Man & Cybernetics. Vol. 2, pp. 2021-2028.
5. *Conscientious Software.* **Gabriel, Richard P and Goldman, Ron.** Portland, Oregon : s.n., 2006. Proceedings of the ACM Conference on Object-Oriented Programming, Systems, Languages and Applications.
6. *"Ercatons and Organic Programming: Say Good-Bye to Planned Economy.* **Imbusch, Oliver, Langhammer, Frank and von Walter, Guido.** San Diego, California : s.n., 2005. Proceedings of the ACM Conference on Object-Oriented Programming, Systems, Languages and Applications.
7. *A Commensalistic Software System.* **Fleissner, Sebastien and Baniassad, Elisa.** Portland, Oregon : s.n., 2006. Proceedings of the ACM Conference on Object-Oriented Programming, Systems, Languages and Applications.
8. *Smalltalk-80: The Language and Its Implementation.* **Goldberg, A. and Robinson, D.** s.l. : Addison-Wesley, 1993.
9. *Squeak (Smalltalk environment).* s.l. : http://www.squeak.org.
10. *Self: The Power of Simplicity.* **Ungar, David and Smith, Randall.** Orlando, Florida, USA : s.n., 1987. Proceedings of the ACM Conference on Object-Oriented Programming, Systems, Languages and Applications.
11. *Patterns, Hypergraphs & Embodied General Intelligence.* **Goertzel, Ben.** Vancouver, BC : s.n., 2006. IEEE World Congress on Computational Intelligence.
12. **Evans, Huw and Dickman, Peter.** *Zones, Contracts and Absorbing Change: An Approach to Software Evolution.* Denver : Proceedings of the ACM Conference on Object-Oriented Programming, Systems, Languages and Applications, 1999.
13. **Ashby, Ross W.** *An Introduction to Cybernetics.* London : Chapman & Hall, 1956.