Exploratory Research Proposal
Virtual-world-inspired Programming Language Design

Richard P. Gabriel
(Failed Proposal: big, colorful—but a sad color—corporation)
rpg@dreamsongs.com

Abstract

> *Unfortunately, most existing programming languages treat software as an isolated, closed-world formal system.*
>
> –ULS Report, page 89

When we—as people—inhabit the physical world, the laws of physics, chemistry, biology, sociology, and even of governments help us by providing constraints against which our actions can gain leverage to get things done. Software obeys few laws—computability is an important limitation, but it sets only a fairly abstract bar for feasibility. Types provide other laws, but so far they have not enabled the breakthroughs in programming language design we need to be able to construct reliable, resilient ultra large scale systems.

Physical laws: to gain similar advantages for software, I believe we need to construct virtual worlds that render laws real. I believe we need constraints that otherwise would be matters of choice, imagination, and standardization. With a virtual world, more stuff is "real," and I hope/expect that therefore the problems I see that are associated with abstraction and scale will be reduced. Further, with survival and health manifest, I expect it will be easier to write resilient and self-sustaining software.

Introduction

The following is from the SEI report on Ultra Large Scale systems [0]:

> *Current foundational models treat software as abstract, isolated, closed-world, mathematical programs. But real software does not fit this idealization at all—instead it is a concrete intentional artifact that is richly embedded into an environment of physical and intentional artifacts.*
>
> *The prevailing idealization treats semantics of programs as an inward gazing exploration of what the program might mean to the compiler. But many important semantic relationships in software profoundly cross the program/non-program boundary. Identifiers in the code refer to entities outside the code. Intentional artifacts other than code form part of the overall software ecosystem and have semantic references to and from the code (configuration files, build scripts, bug databases, email archives of design discussions, etc.).*
>
> *The idealization treats software as being abstract—but of course it is not. The computation is a physical process, running on real computers over real networks. The software is encoded physically, and has classic properties of physical systems, including the fact that its size affects aspects of viability and behavior such as scalability, distribution, latency etc.*
>
> —ULS Report Early Draft

The purpose of abstraction in programming languages is to isolate the programmer from the reality of the underlying programming language and hardware, where the only real things are bits stored in bytes and words in computer memory. These fabrications are raised up to be numbers of different types, strings, and maybe vectors and arrays—and some other things like this, perhaps some simply structured collections. The problem of programming is to take real-world concepts and constructs, and map those ideas to these computerish realities. The downside of abstraction is that when a program is thoroughly abstracted—made up of abstractions for all the concepts and things the programmer needs to handle—the web of abstractions depends on their interfaces to a degree that makes it hard to change one or a few. That is, unless the abstractions are perfect, it is likely some will need revision at the interface level, and without very good tool support this can be difficult, and further errors can be introduced.

Moreover, in a huge system, there can be so many abstractions—each like a little language, sometimes a microlanguage but sometimes substantial—designers and developers can become overwhelmed by all the little languages to learn and master. In small systems, the learning is easy or at least the benefits outweigh the costs. But in a

huge system, the burden of learning can be too much and flaws can accrue.

Here is what Paul Feyerabend says about Ernst Mach, scientist and philosopher:

> *We have seen that abstraction, according to Mach, "plays an important role in the discovery of knowledge." Abstraction seems to be a negative procedure: real physical properties . . . are* omitted. *Abstraction, as interpreted by Mach, is therefore "*a bold intellectual move.*" It can misfire, it "is justified by* success.*"*
>
> –Feyerabend, Farewell to Reason [6]

Other conceptualization of ultra-scale systems have tried to address these problems, most notably the "systems of systems" work. This endeavor, which has evolved over the past 10 or so years, is largely based on the assumption that the components of a system (of systems) are themselves systems, and therefore there is a mostly hierarchical decomposition of the "problem"—and this is not surprising because much of the progress we've made in constructing ever-larger software systems has depended on such decompositions. Again, though, this solution to complexity at scale—as in the case of abstraction—relies on concepts from within the framework of a purely abstract endeavor.

In most cases, the real world makes its appearance in software through the mediation of software developers, who are not usually experts at this—even though they are masters of abstraction, they are not masters of science nor of the domain for which they are writing software. Therefore, their abstractions tend to be a bit inept and eventually need to be revised. This makes for unstable code, and sometimes the need for adapters or impedance matchers to get the parts of the system (of different systems) working together. When the "same" real-world object or concept is abstracted by different software developers (in vastly different contexts), it is typical for there to be a mismatch, requiring programming and other adaptive measures to be taken.

One current approach to this problem—and an approach with a lot of energy and effort behind it—is to produce ontologies that describe the world which software developers can rely on, at least in a single project. An ontology is defined this way by Tom Gruber:

> *An ontology is an explicit specification of a conceptualization. The term is borrowed from philosophy, where an Ontology is a systematic account of Existence. For AI systems, what 'exists' is that which can be represented.... [A]n ontology is a specification used for making ontological commitments.... Practically, an ontological commitment is an agreement to use a vocabulary (i.e., ask queries and make assertions) in a way that is consistent (but not complete) with respect to the theory specified by an ontology. We build agents that commit to ontologies. We design ontologies so we can share knowledge with and among these agents.* [7]

Ontologies, again, operate within the realm of abstraction and conceptualization. Its elements are symbols and formalisms, and their interactions are linguistic. The real world speaks to us, as well, but with knocks and bumps, and with the need to eat and rest, and with the challenge to learn and adapt (as individuals and as a set of living organism) followed by death.

Beyond the (controversial idea of) problems with abstraction are the problems associated with keeping a software system running well. Lately developers have started to take seriously these problems by programming in additional mechanisms to care for the running software. I suppose you could look at garbage collection as a health-related mechanism. But the mechanisms being used today represent a narrow range of what I believe are the viable possibilities for true self-sustainability. For example, biological ideas have not been particularly exploited to build software systems.

Next, I believe there is a need for more programming paradigms and constructs. We seem to have run out of usefully different ideas based on the textual metaphor: the metaphor that the program is the text on the page, and that people and programs need to be able to reason about textual pages of source code.

Finally is this observation: although it doesn't seem like it, the constraints placed on what we can do and build by the laws of physics, biology, chemistry, and other disciplines concerned with the real (physical) world help us with the doing and the building. Gravity, for example, enables us to move, to build some things by merely piling, helps us organize our lives by keeping things in place, makes friction an interesting phenomenon, and probably was necessary for the creation of life itself.

<diversion>

In the early days of pre-statistical artificial intelligence research (I'm talking about the 1970s), there were a series of planning problems that were proposed to exercise research in (robot) planning, involving what was called the "blocks world." The blocks world was a sort of simulation of the real world that enabled researchers to perform experiments and do research without having to construct working (and possibly easily breakable) robots—which were beyond their capabilities at the time anyway.

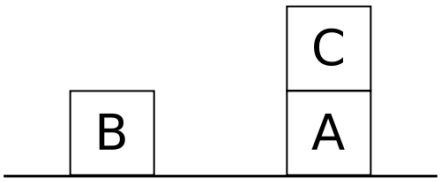

Figure 1

The blocks world was a purely symbolic representation, with statements (in Lisp) like these to indicate the configuration of blocks:

```
((a on b)(b on c))
```

Which is taken to mean that the block named `a` is on the block named `b`, and `b` is on `c`, which forms a tower. The problem is always to take a starting configuration of blocks, an ending configuration, and a one-armed robot, and to come up with a plan of block moves that would transform the starting to the ending configuration. Simple, eh?

In the mid-1970s Gerry Sussman discovered a blocks world problem that confounded many, and it has come to be known as the *Sussman Anomaly*. Here is the problem. The starting configuration is shown in Figure 1. The problem is get `A` on `B` and `B` on `C`—that is a stack with these blocks from top to bottom: `A`, `B`, `C`. The classic approach is to state that the ending configuration is this:

```
((A on B)(B on C))
```

The classic planners would try to establish one or the other of these goals first, and then the other. Trying to solve `(B on C)` first results in the situation shown Figure 2, and at that point `(A on B)` is not possible to achieve; and if the planner tries to solve `(A on B)` first, the resulting situation is shown in Figure 3. This is because the planner can try only the two orders, and there is no such thing as interleaving steps. The technical problem as Sussman has named it is "prerequisite clobbers sibling goal." His planning system—interestingly called "Hacker"—solves it by going into an error-ignoring mode, which enables it to do an irrelevant step first (moving `C` to the table). (And such a move might remind us of simulated annealing where a step that makes things worse overall might be permitted occasionally.)

The interesting thing about all this is something you (the reader) have perhaps noticed. Although it's not stated anywhere, the situation assumes that there is gravity, and that the blocks all need to be supported. There is syntax for that (the `(`*x* `on` *y*`)` statement), but it isn't used to state all the parts of the situation. Peter Norvig in *Paradigms in Artificial Intelligence Programming* codes up this problem, and though he states the initial configuration like this:

```
((c on table)(a on table)(b on table)
(space on c)(space on b)
(space on table))
```

He doesn't state the goal the same way—with explicit mentions of the table and open space—but as the 2-part expression I used: `((A on B) (B on C))`. If he were to state the goal like this:

```
((A on B)(B on C)(C on table))
```

a simple reordering of the goals would work with the code he was demonstrating (the general problem solver or GPS code). The failure is that even though some representations of the problem/configuration recognize that there is gravity, not all of them do. And these mistakes are being made by, as a character in *Raiders of the Lost Ark* might have called them, top men.

Instead, the AI planning researchers invented interleaving planners and other sophisticated techniques to handle this and other problems. One could argue that the goal stateent `((A on B)(B on C))` makes room for lots of other solutions (like putting `C` on `D` and making an even taller stack), but the solver doesn't know that for every block, *x*, (*x* `on` <*something*>) or it will fall however far it needs to

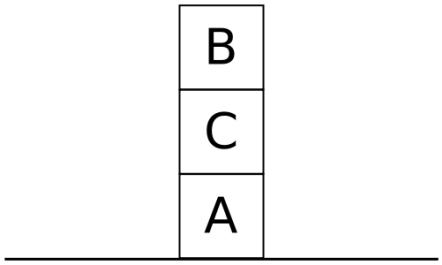

Figure 2

to make it true (unless it's in space).

My point is that what is easy for us to do is to work in a world with physical and other reality-based requirements and contraints, and we try to operate in a purely abstract world (with ontologies, perhaps), but we aren't disciplined

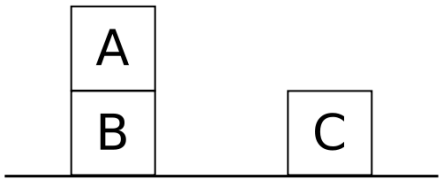

Figure 3

enough to do it well. Keeping it in our heads doesn't seem to work.

The Proposal

I want to create a virtual world/runtime—something like Second Life—where software runs. It would be like a virtual machine, but the stuff there—the software, what otherwise would be abstract constructs, and other useful artifacts such as business models—would be visible and as if physical. The virtual world would have laws of physics, biology, sociology, chemistry, and even computation that would together govern the things there, the laws enforced and enacted by simulation engines like the physics engines in some virtual worlds including games. Software would run there as it does now on a virtual machine, but the physical, biological, and other aspects of the software under the purview of the various simulation engines would obey the laws and suffer/enjoy the constraints imposed on them. But like other online virtual worlds, people can visit this world and observe the entities there. Manipulating software would not be limited to conceptual and textual tinkering.

The Purpose of the Virtual World

Unlike Second Life, the purpose of this virtual world is not to be a place where people visit to socialize or even to be the place where software developers do their work. It may turn out to be a good place to develop code, but the primary purpose is to provide mechanisms, metaphors, and insights that enable human designers to design better programming languages.

At present most of our programming languages are solidly based on mathematical foundations with some assistance from abstractions designed by software architects and developers to represent (small) parts of the real world for a particular purpose. Types help with this by providing some help with making sure that the very limited constraints types place on the software's text are kept to, but these constraints are mostly if not entirely aimed at making sure that the form of the software (as a text) is such that certain programming errors are avoided (early).

The problem with our current languages—speaking in generalities—is that the solid facts upon which they depend are limited and inflexible, requiring invented superstructures to bring the language closer to being directly usable. It's my hope and expectation that if the universe programs exist in had some more constraints, behaviors, and laws that the capabilities we could build into software could expand to make developing software easier and quicker.

The idea is to move from the world of text to something more like a real world. Our current languages and type theories are designed to operate on source text (or a tree-like representation of it), and it is important for people to be able to reason about local attributes—because people are not good at holding a lot of information in their heads. It's not a coincidence that it was in the context of languages that represented types at runtime—in the execution environment of programs—that advanced memory management was developed. I hope to replicate this kind of language success using runtime virtual worlds. I'll have a bunch of examples in the next section, but this is a good place for a simple one.

Suppose that in the runtime virtual world (RVW), software components that were constantly executing (and/or had a lot of work in their queues) were, like kryptonite, to glow cold green; the brighter the more overworked they were. Other software components, wandering around the RVW, could notice this and perhaps build a structure where there were a number of copies of the component, and with an input dispatcher and an output re-integrator around the copies, thereby improving throughput.

The wandering software could be completely generic—working with any system within a particular type of RVW. The only (minimal) requirements on "real application code" would be that the components that are manipulated present a second interface that enables them to be rewired and are able to make RVW expressions—the glowing cold green—and other gestures like that that reflect inner state. Software that is more in tune with the RVW will have capabilities like those suggested in the next section.

Possible Capabilities

The following are possible avenues of exploration for the idea of a runtime virtual world.

Physicality: Everything in the RVW is manifest: visible, tangible, making sounds, giving off odors and other emanations, and moveable including by self-propulsion. Actions and conditions that are visible in the real world are visible in the virtual one. For example, when a chunk of software is manipulating a real-in-the-virtual-world "data structure," it will have a hold on it or will otherwise be apparently in possession of it or tethered to it with a visible tether, and other chunks of software waiting to use it will be visible—perhaps standing in line or crowding around. Software that wants to use the "data structure" can observe the queue and decide to wait or go off and do something else or delegate to a colleague the task of waiting for the structure and doing whatever has to be done.

A software chunk will have a size and weight that depends on how large and complex it is. Being able to "nail 2 things together" (compose software objects) will depend on whether their underlying physical natures—and hence logical natures—would make that meaningful. In fact, the

exploration of how to render physical what is now done textually will be a major area of exploration.

Vision and other senses: In the runtime virtual world it's possible to see things. People can see them, and, if programmed to, so can the chunks of software. (For now I don't want to commit to the units of software that will appear as coherent entities. It would be tempting to call them components, but that might eventually have the wrong consequences or implications.)

Software that is transparent (see below)—like a house with glass walls—can have its inner workings observed, so other software can tell whether it is hard at work (perhaps in a tight loop) or just wasting time waiting for something to do. Perhaps by observing peculiar or atypical behavior, another software chunk can report on the ill health of the first.

Software chunks will have a physical appearance regardless of how they are programmed. They might be encased in a default physical shell with stereotypical physical properties, and perhaps the default appearance will be ugly. This will serve to encourage software development organization to take some time to pay attention to this aspect, which could have beneficial side-effects, such as making the software system more flexible and reliable.

Vision is not the only sense available. Hearing, smell, and touch could be as well. Touch might be only sensitive to hardness, impenetrability, and some textures. Hearing is a sense that carries a considerable distance but not as far as sight can. Smell is a closer in sense, depending on the concentration of odor.

The use of senses provides the designer with other avenues of interaction and communication than the delivery of "arguments" via function or procedure call, or message passing. By using smells, for example, a software chunk can indicate distress, a desire for assistance, or simply the state of its computation. Sound can be used the same way, with the main differences being the degree to which distinctions can be apprehended and, especially, the speed and persistence of the signal. Vision depends on nearly instantaneous transmission of the signal over long distances, requiring the thing to be seen be illuminated, and the duration of its visibility is the duration of that illumination. Sound travels quickly but nowhere near as fast as light, is caused by the thing heard vibrating, which can happen by action of the thing or by some other thing acting upon it. The duration of the signal is the duration of the vibration. Sounds can combine in ways that make the their "meaning" ambiguous or difficult to comprehend. This is also true of odors. Odors travel slowly, disperse—as does sound—and persist. It is a slow signalling mechanism, suitable for communicating slowly dawning or persistently important information.

Vision can impart a great amount of information. Sound less so. Odor can transmit a complicated structure (chemicals) that can perform actions on whatever it contacts, just as an acidic mist can corrode surfaces. In a cellular setting, complex proteins can monitor internal activity in other cells and take action, such as an immune reaction that will "order" a cell to self-destruct. In fact, it might make sense to make a virtual world at the cell level—where the primary simulation is closer to biological than physical—so that more complex information can be transmittable through software "cell walls."

How a chunk of software looks will reflect how it's behaving, what it's doing, and its general health. The color or appearance of a software chunk can indicate its origin or its general purpose. When two systems are combined, they will carry a visual, aromatic, sonic, tactile, or chemical fingerprint indicating its origins and perhaps capabilities. Its documentation can be glued to its surface.

Geometrical: Distance makes a difference. Software chunks near each other can do things together that more distant chunks cannot. Physical nesting means something. Non-hierarchical containment means something. Software motion can mean something; for example, several chunks operating concurrently can make progress toward their rendezvous point, and other software that is depending on what happens when the rendezvous occurs can see progress and plan accordingly. For example, "it looks like that will take a while, so why don't I go do this other task in the meantime."

Moving closer together or farther apart can enable not only more intimate or more formal communication, but also can enable the other sense "channels." Close up, the odor of a software chunk can be obvious and immediate. Information passing through touch can be enabled only through close or immediate contact.

People visiting the virtual world can move things, and perhaps by doing so they can improve performance or other resource utilization. Or software can be programmed to do that.

The size and shape of software could indicate its resource use, so a chunk with a resource leak will continually grow larger; software that spawns lots of (useless) processes will be seen like a cancer metastasizing. Other components can be programmed to observe such things and either try to repair, minimize the damage, or report it.

Note that this is an example of what I expect to be a theme for the research. It is possible today to program monitors that observe the behavior of code and take actions based on it. However, the incentive to do so is academic, and in some if not most cases, infrastructure must be invented. When systems are deployed in a RVW, a lot more infrastructure is there already, in a form that is observable by people who visit the world. Software that doesn't go to some effort to be an active part of that world will simply look and act silly. And a price can be paid for looking silly when the actions of other entities are unable to keep silly-looking things from dying or falling apart. This forms an incentive to try

harder, and with more of the abstractions of software rendered (virtually) real, there should be more opportunities to be creative—especially in dreaming up new language constructs and capabilities.

Biological: In some RVWs longevity might not be something to take for granted. A software chunk that does not obtain nourishment might die very soon. In such a RVW, all software chunks will eventually die and must reproduce to continue in any form. Nourishment can be provided in many forms, including the following:

- when work is injected into the virtual world, that work is nourishment, and it is provided to those software chunks that endeavor to do the work; when work is successfully completed, additional nourishment is provided to the software that contributed to completing it
- when some software is "ill," the software that cures it gains nourishment
- when software is born, it gains nourishment
- when software vanquishes misbehaving software, it gains nourishment
- when software improves the performance, resource usage, or throughput (or other such thing), it gains nourishment

Other biological mechanism can be programmed into the simulation. As mentioned, some can be at the cellular level where there is a rich and close set of interactions between what corresponds to cells and proteins. Immunity can be taken more literally in such a RVW, and evolution can be a more important part of the behavior of the world.

This aspect of the RVW—which is something that will be explored—is reminiscent of the "resource-limited computation" approach to artificial intelligence pursued in the 1970s and early 1980s, partially rediscovered by genetic programming and genetic algorithms during the last 5–10 years. Immortality implies stasis, which cannot be an particularly adaptable strategy. Forced turnover might change how systems are put together. It could be useful to explore this earlier literature to see what can be learned.

Transparency: Software chunks are automatically instrumented to show on or through their surfaces things that are going on inside. For example, software that throws a lot of exceptions (internally or not) would give off a particular sick color and perhaps an odor. Software that is in a tight loop would glow red then white hot (if you don't like the cold-green idea). Software that isn't doing anything would have a brown or black color and be laying on the ground. Software also would make sounds depending on what it's doing.

Some software is also literally transparent in that it's possible to see inside, and the stuff seen there is physically in the virtual world and manipulatable. Software that is not created to be a participating citizen in the runtime virtual world (can't see, doesn't move, isn't instrumented, etc) is called ***inscrutable*** and is shown as a black lump on the ground. But perhaps it won't last long in an RVW that simulates a living system: without nourishment it will die.

Effective: Software can have controls that stick out of its surface that can be used to control or customize it. Adjustments, feedback loops, control points, selectors, motion simulators—all these are ways for outside software or people to cause changes and action in a software chunk.

A person (or some software) could observe a remote relationship by seeing a tether from one chunk to another and perhaps intercept the interaction, redirect it, split it, or replace it.

User Interfaces: When a person wandering the virtual runtime comes across a software chunk that can be interacted with, its interface appears on its surface, and the person can expand that interface to the entire screen. A software chunk that is capable of accepting a variety of interfaces—such as a spreadsheet permitting the use of a general text editor to revise the text contained in cells—presents a default interface along with a slot for a user-supplied one. Imagine walking around with your own favorite user interface tools and being able to plug them into compliant software.

This is how development, debugging, and repair could be handled in the virtual runtime—with tools brought in by the developers which would be able to alter source code, install breakpoints, etc. Software would be variously transparent (or inscrutable) depending on what parts of its innards are available for inspection and modification.

This is one of the key points for investigation: Should most development be done actually in the virtual world, should some of it—the algorithmic part, for example—be done in text as it is now, or should the bulk or all of it remain a textual exercise? At present I don't see major or even substantial development being done by direct manipulation of representations in the RVW—such as by plugging wires from one place to another. It will be possible to do some tweaking and especially observation and debugging that way, but if any entities are intended to do work in and with the RVW by manipulating representations there, it is the software that inhabits it.

However, it might be advantageous for development to be situated within the RVW. I hope that more and better interaction modes will be developed as experience in RVWs increases. Perhaps a greater use of property sheets (or instrument panels) will be a natural outcome.

Domain-Specific: The virtual world would be divided into domain-specific ***rooms*** where particular ontologies would be in place as real objects. For example, in a business room there would be (virtual) physical objects representing (from our real world) money, orders, dates, and business relationships, such as customers, suppliers, and manufacturers. It would not be up to a software designer to invent a representation of money, for example—using money would be a

matter of manipulating objects found in the virtual world, with perhaps rules or processes for making new instances, mimicking the way that happens in the real world. Two software chunks would be provided affordances according to what they are/represent in the domain-specific world, so that a customer would be able to purchase items but not build things, unless that were also a role of the customer. Whole swaths of functionality and capabilities would be provided by the domain-specific room along with rules, laws, and other constraints (both hard and soft) controlling and regulating objects in that world.

This means that there is no need to provide translators and interpreters for data structures. Every bit of software created in a room will be able to understand and manipulate the underlying domain-specific objects there. This is intended to control *ad hoc* and uncontrolled abstraction, which can, in large-scale software systems, create a Babel of specific languages too numerous and likely too complex for designers and developers to understand and certainly to invent reliably.

The correctness of these designed-in objects in each room would be subject to relatively easy verification by experts in the domain because they can visit the room and do simple tests by direct manipulation. They can, for example, go to a room supporting cash, withdraw some from a virtual ATM, see how much they have, add it to their wallets and see whether it adds up properly, go to a vending machine and purchase items to see whether it acts like cash, etc. Money in this world would obey laws, like the laws of physics but appropriate to currency. For example, it would never be possible for any program to create money with negative worth or value (for the surreptitious purpose of being able to withdraw negative cash from an ATM, for example). A Java program that tried to do that by setting a field, for example, might "believe" it succeeded, but the thing it created would simply disappear, and the code would be left with nothing in its "hands."

In such a (virtual) world, it would be easy to invent things like monotonic variables (that either only grow or only shrink so that determining whether $x<y$ could take place before the final versions of x and y are known if one is fixed and the other monotonic in the right direction, for example.) Though such variables have already been invented, my claim is that operating in a (virtual) real world will make it easier to do metaphorical thinking and thus be able to come up with new programming constructs like this one.

Creating domain-specific worlds will seem to some to be like standardization, but I believe the realization of the objects—their appearances etc—in the virtual world will make them more easily and obviously complete and correct. One possible fallout from this is that perhaps interoperability and even the standardization process would become not only more stable, but more useful and even sensible. It's not hard to imagine a couple of corporations getting together and creating a common runtime virtual world with all the standardized components and behaviors built in.

In short, rooms are where the abstract becomes real.

Mapping to the real world: Some parts of the runtime virtual world can be mapped to the real world. Imagine software for a small office. The RVW for that system could have within it printers, computers, cameras, and other devices that correspond to ones in the actual office. By observing a software chunk printing to a printer—with a physical tether to it—a person could locate the printer in the real world. Moreover, and perhaps more importantly, a person could enter the RVW to direct output to a particular printer by locating it in the representation of the actual space.

The convergence of the real and virtual worlds would also help make programming / software constructs more tangible—more concrete rather than more abstract as we are pursuing now. With an active crossover, validation, usability, understandability, agility, and rapid value creation could be improved.

Scale and time: There is no reason to limit the level of scale to the few orders of magnitude that would be immediately apparent to a human visitor or a cell-sized entity. It might make sense for there to be a largely invisible microscopic world hidden from view from the large, main actors in the system, and an even larger ecosystem for even larger concerns. Observing stuff at other than the natural level of scale for the observer would require special instruments, and affecting things at those different scales would not be straightforward.

One of the most important and most difficult issues associated with scale is time. First, for most software, time is what passes while the cpu's clock ticks away. Not many pieces of software take time explicitly into account aside from real-time software and operating system task scheduling (cron jobs, for example). In a RVW the issue of time has at least to do with how fast software chunks are able to move and therefore react. Should there be explicit clocks or other time signals like day and night? What about when people show up and the time needs to slow down for them to see what's going on? Time will be tricky.

What We Might Learn

Dealing in a purely abstract world is not helping as much now as it did in the past, where abstraction hid irrelevant details. Now the abstractions *are* the irrelevant details.

By making abstractions real with physical, biological, and other laws in effect, my hope is that it will be possible to invent new programming constructs and even paradigms based on leveraging a real environment where software lives. For example, by embodying software in a biological setting we might be able to take repair, self-repair, and adaptation more seriously. And with a realistic concept of survival, competition, and nourishment, it might even be

possible to think about using evolution with the wild fitness function—the one in use in our real world.

Cognition will be a viable and real part of software, enabling intelligent fallback mechanisms formerly dreamt of only in artificial intelligence systems. By giving software a corporeality, developers will be able to better use the intuition they've gained by living in the real world, both to design and produce better and more reliable software, and to understand better what programming and software development are so that better languages and tools can be built.

By being able to visit the world of running software, people will be able to understand even better the software they've built and perhaps be better able to improve what they've produced. Damage to software and data will be visible as hidden mechanisms associated with the underlying simulation (the physics and biology engines, for example) do their work and show tears, discoloration, and distortion.

Naturally running the simulations in addition to the actual computations will be expensive. The bet is that the computational cost will be outweighed by the benefits gained by better programming languages, less transcoding, better self-sustenance, and a more intuitive operating environment. Paying for the additional computation for the simulation will be done with some of the cores on a set of (massively) multicore-based computers. The further bet is that it won't take many cores to do enough to effect the constraints imposed by the virtual world.

Another key problem to solve is the time mismatch between humans and computers. It's easy to imagine visiting a world where events happen on a human speed scale, and also for software interacting with software to be able to keep up with the speed, but people watching software interact will see at best blurs. Perhaps a statistical approach will work, or perhaps only some of the software's activities will be visible to people, with the high-speed movements and changes hidden. This could imply a different dimensionality for people and software, with 3 or 4 only in common. This exploration, if it comes up with something novel, could benefit ordinary debugging and inspection in Java and other languages.

Because the runtime virtual worlds will be much more elaborate than typical runtimes or virtual machines, with the possibilities for people to visit and truly examine the software inhabiting it, and because there will be a theme of exposing the health and status of software chunks, it might make sense to explore the idea of continual testing. That is, if there are unit tests (and even integration tests that can be executed without supervision), perhaps they can be run while the software system itself is executing, but during lulls. This might encourage more consistency checks to be designed and programmed by software creators. Perhaps if successfully completing tests with a clean bill of health is a way for software to not die in the RVW, these practices will be encouraged. This could change how development and particularly never-ending development is done.

The nature of development could change because the nature of the execution environment is so different. One side avenue of research—perhaps something another group could look at—is the sociology and psychology of adoption of new programming or software development techniques based on the available of new technology or infrastructure. Many, starting possibly with Donald Knuth, have recommended literate programming, but the amount of extra work required to do this well limited its adoption, despite the fairly clear and obvious benefits of the practice. No amount of technology brought to bear to make it easier seemed to help.

Research Steps in Detail

The following are the steps needed to explore the questions above. This is the section where the fail-fast stuff is described.

Is there any merit at all in this idea?: The first step is to find out whether it will be possible to learn *anything* from this approach or whether it has any advantages at all.

The first step will be to do some very simple experiments to determine whether any of the basic ideas can work. The first steps are to so some small, 1-person experiments.

Fail Fast Ideas

One would be to see whether it's possible to evolve entities that can move to software components in distress. These entities should be made up from parts not usually seen in computation. Another would be to see whether it's possible to evolve anything interesting, however minor, using the natural fitness function (survival). This would be an important accomplishment for the project, so seeing any progress early on would be a useful measure. A third test would be to see whether we can come up with a new computational construct based on the assumptions of a virtual runtime. This proposal has a couple, but for this test I would want something that could be tested in real (though toy) code.

Another major early test would be to construct a tiny, 2-d world with some laws in place and to see what it would be like to program in such a world—particularly whether it's possible to take advantage of the physicality of it. For example, can proximity be useful? How about seeing? How would seeing work? How about smell?

Assuming some degree of success for these early experiments, the next step is to extend the tiny world to one a little richer, and in it to see whether it's possible to effect changes based on learned or evolved components. The ultimate aim of this tiny prototype is to determine whether it's possible to construct a runtime world where the natural fitness function can operate. This would be a major discovery in that all evolved computational entities to date rely on a human-tuned fitness function. In general, each of the claimed possible advantages of the proposed approach will

be explored in the small in this tiny prototype. The point is to determine whether it's reasonable to attempt a larger, more elaborate, more fully featured prototype.

The tiny prototype will be in a dynamic language, and I'll be trying to leverage as much existing code as I can find that will work—but not worrying about great visualization or accurate/complete physics, accurate/complete biology. If the early experiments above are successful, I'll use the prototype to see whether I can come up with any new computational construct—perhaps applicable only in the runtime virtual world—or any insight at all into system building or programming that would be interesting to report to something like a programming language workshop. I suspect this could take 6–12 months, but I hope more toward the short end of that scale. I will at the same time try to engage some other collaborators to help.

This will constitute an early failure indicator—or more likely an it's-too-early-to-try-this indicator. Another measure that might be interesting is whether the results of this stage are interesting to any publication venue—again down to the level of a workshop.

I estimate this will be in **months 0–12** of the project, and just me.

What is the nature of runtime virtual worlds: After some work in the first stage, it will become possible to figure out what the real nature of the runtime virtual world could (or would) be. From what's written so far we know that we will be creating something like a virtual machine or runtime for executing software. Moreover, we will need to build in either a physics and biology engine or something like them. This implies that software "chunks" need not only to execute as designed but be manifest or encapsulated in such a way that the simulation engines can manipulate them. Will this be via an actual virtual machine? Will it be through a sort of proxy setup where an ordinary VM or runtime runs the application software and tethers hook it to a simulation engine which then feeds back into the ordinary execution? Answering these question is the second step in the program. The questions will, of course, be tentative because I expect there to be a number of twists and turns as we go along.

I estimate this will be in **months 3–9** of the project, and a combination of me and a postdoc.

Real or simulated?: Exploring the research questions perhaps doesn't require actual running applications but only simulations of them. Perhaps we can learn enough about the statistical behavior of particular applications to model them accurately enough to provide an adequate platform to study the benefits and attributes of runtime virtual worlds. We can likely get a sense of this during the initial prototyping stage. Possibly we can mine an actual system description at a depth suitable for simulation from Grady Booch's work on surveying software architectures.

I estimate this will be in **months 3–12** of the project, and the postdoc and me doing the work.

Language(s) and a fully functional prototype: There are a number of distinct programming languages in play in a real RVW executing a software system:

- the primary language of the application. This is a language like Java, probably.
- the language the RVW is written in, including the simulation engines. This could be C or C++, but Java could work and perhaps some other languages.
- the language that entities able to be manipulated by the simulation engines are written in. This language might need to be more flexible and dynamic than the others because these entities might need to evolve (literally), for example. This language also needs to be parallel or support threads, but, again, not in a rigid way. I imagine a dialect of Lisp would be a good choice for this. Also, this might enable us to leverage AI software already out there in repositories.

The question of language is likely to be a difficult and controversial one. To be honest about it, I'm fluent really only in Lisp-like languages, though I've programmed in others. I expect to use a mixed platform or set of platforms with interlanguage communications forming a key part of the system. I will need a programming partner who is energetic, enthusiastic, and able and willing to be extravagant in his or her thinking.

Doing a full-blown prototype that could run real code will probably require a team of people and about a year of duration. I estimate this will be in **months 12–24** with a team of 3 people plus me.

Visualization and simulation: When people visit a runtime virtual world, they need to see and perhaps hear, smell, and feel the stuff that's there. This requires some degree of visualization, and most people will expect Second-Life quality at least. Visualization is not high on my list of priorities, and I would be happy in the initial prototype stage with a simple 2d visualization, but perhaps we can do better when we see what's out there.

It's possible other existing platforms could be used—for example, Squeak or something like it. There are possibly existing physics engines, though perhaps not a biology engine. The ideas discussed earlier about contract and business

engines might have already been explored by researchers I don't know of offhand, and will be explored.

Exploring the possibilities for existing visualization and simulation engines and doing some preliminary prototyping can happen early on with a postdoc.

I estimate be in **months 3–6** for the postdoc.

Organization of the Project

The project will be mostly a 1-person or 2-person project at first. I hope we will be able to find a postdoc or similar partner. If we find enough results or promise, we can perhaps pursue a fully functional runtime virtual world that could run large or substantial Java-based systems. That would require a larger team with different characteristics. I have a lot of experience running software development projects.

But, because the ideas we will be exploring are possibly far out, I would like to enlist "volunteers" from the community at large. I imagine an open-source-like community with workshops and small symposia taking place regularly, possibly with an early kickoff workshop this Winter.

All the code will be open source.

Some universities have researchers or students who might be interested, including MIT with Martin Rinard, UC Irvine with Crista Lopes, and The Chinese University of Hong Kong with Elisa Baniassad.

In Conclusion

With software systems growing beyond human comprehension, depending on formal methods attached to abstraction is seeming less likely to be effective as the pace of growth of systems exceeds that of progress with those methods. Our old friend abstraction, once the cute puppy of software creation is now become a large, ugly dog displaying its earlier charm only on occasion. Or at least the ugliness is starting to become apparent. We need to find ways of coping with scale and complexity by using our super-performing computers rather than trying to squeeze every ounce of computational horsepower into the main computation. Not only is it too hard to understand all the abstractions that might appear in a large system, the extensive use of frameworks often means that data is transcoded many, many times—sometimes tens of thousands of times in a simple business transaction. By making abstractions real and outside designer and developer control, we can start to tame these problems.

An intriguing experiment perhaps worth trying.

References & Further Reading

[0] Northrop, L., et al, *Ultra-Large-Scale Systems: The Software Challenge of the Future*, The Software Engineering Institute, Carnegie-Mellon University, 2006.

[1] Gabriel, R., Goldman, R., "Conscientious Software," Proceedings of the 21st Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications, Portland, Oregon, 2006.

[2] Maturana, H., "Autopoiesis," in *Autopoiesis: A Theory of Living Organization*, Milan Zeleny (ed.), pp. 21–30, New York, North Holland, 1981.

[3] Maturana, H., Varela, F., "Autopoiesis: The Organization of the Living," in *Autopoiesis and Cognition: The Realization of the Living*, (1980), pp. 59–138, 1973.

[4] Rinard, M., "Automatic Detection and Repair of Errors in Data Structures," Companion to the 18th Annual ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages and Applications, Anaheim, CA, pp. 221–239, 2003.

[5] Rinard, M., Cadar, C., Nguyen, H., "Exploring the Acceptability Envelope," Companion to the 20th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications, San Diego, California, 2005.

[6] Feyerabend, Paul, "Farewell to Reason," London: Verso, 1987

[7] Gruber, T., http://www.aaai.org/AITopics/pmwiki/pmwiki.php/AITopics/Ontologies#readon