Proposal to NSF – Granted on August 31st 2006

Steps Toward The Reinvention of Programming

A Compact And Practical Model of Personal Computing As A Self-Exploratorium

Alan Kay, Dan Ingalls, Yoshiki Ohshima, Ian Piumarta, Andreas Raab

We *make*, not just to have, but to *know*

Introduction: Over the years, our research has combined interests and inventions in computer science (object-oriented programming, reflective whole systems, networking), graphics (bit-map screens and "bitblt"), UI (overlapping windows and icons, modeless editing and interactions), education (particularly for children), design (aiming for beauty as well as problem solving), and computer engineering (learning how to efficiently make whole HW & SW systems). The latter has allowed us to tackle new approaches on a large scale, deploy and test real solutions without huge teams or being bound to particular computational models, tools or platforms. Some of the alternative approaches we've invented have become mainstream (and some not), but a very important process in our work over the years is to be able to control our own HW & SW destinies by making high-enough-level languages with powerful enough tools to allow our small but critical mass research group to successfully implement anything we want to try.

In short, we've been very interested in the *ideas and ideals* of personal computing since its inception, and have strived to come up with real advances for real users, by making real systems, deploying and testing them.

These alternative systems have been small in comparison with standard practice, e.g. today's Squeak Smalltalk[Sq1] covers much of personal computing, includes its own applications, operating environment, UI and development tools, runable specifications, etc., in about 2.8MB of code (about 200,000 lines). But our intuitive sense of "mathematical entropy" insists that an even more comprehensive design approach to whole-system personal computing could be smaller by a factor of 10 or more (a factor of 2 from removing non-used code, and another factor of 5 or more via a different, more advanced architecture and design).

Possibility And Proposal: This opens the exciting possibility of *creating a practical working system that is also its own model* – a whole system *from the end-users to the metal* that could be extremely compact (we think under 20,000 lines of code) yet practical enough to serve both as a highly useful end-user system and a "system to learn about systems". I.e. the system could be compact, comprehensive, clear, high-level, and understandable enough to be an "*Exploratorium of itself*". It would:

Contribute a better approach to personal computing overall, and many individual parts of it
Provide important advances in computer science, software engineering and understanding of design
Boost pedagogical and general understanding and learning about systems and how to explain them
Create a test-bed environment for experiments into "the programming of the future" for end-users and pros

Intellectual Merits: A large number of the most important ideas and intellectual contributions of this system are new (or are non-mainstream ideas from the past that will appear new) disposed in powerful and often novel ways. Those we find particularly interesting are: how the bootstrapping is done, a "universal object" approach to end-user facilities, alternatives to "OS", "apps", "web", etc., use of roles instead of inheritance, symmetric messaging and events, invertible processes, labeled histories, distributed objects, protection, separation of meanings from optimizations, coherence and persistence of meanings, instrumentation for many kinds of self-disclosure and explanation, new ways to program for end-users and adepts, etc.

Broader Impacts: Most early learning of programming is done in a non-scalable way, somewhat equivalent to banging together a doghouse with nails and planks. What is learned doesn't scale well by a factor of 5, let alone factors of 100+. Even more critical is that the deeper mathematics-like nature of the most powerful ideas in computing are poorly described and learned via papers: even if they are read, this is a difficult form for learning and understanding. We think that making a well designed system that is also instrumented to be learned, understood and changed could have a large positive impact on many areas of computing. It would constitute an example, and a kind of "the system is the curriculum" for learning many important powerful ideas, especially for teenagers, university students, and in the 3rd world, where there is a pool of literally billions of potential computer users and authors, who could make great use of a simpler stronger approach to personal computing and explanation.

Outline of the Proposal:

1. We start by giving an abstract of what we mean by "a whole personal computing system".
2. Then we show suggestive examples of some of the design approaches we expect to take.
3. Some of the technical details and powerful principles will be discussed
4. Then we outline how we plan to go about making the system over a period of several years
5. We end with discussions of collegial relationships, education, dissemination, etc.

1. What Do We Mean By "End-User to the Metal" Personal Computing?

Compact models of fundamental ideas have aided thinking in many sciences. We think they haven't been done often enough in computing, but when done, have had similar positive effects [Mo]. We want our model to be explanatory, but we also want it to run well enough to serve as a practical artifact with a wide range of application.

A good *model* of personal computing should certainly cover today's good ideas and scales, but because it is an extreme abstraction, it does not have to reverse engineer shortfalls in unity and design; instead, it can be unified as much as possible if it covers the worthwhile parts of the end-user experiences.

We can start with our computational environment as a world-wide "every person" communications network with suitable physical input and output effectors for all users. We will address typical hardware of today, including the tantalizing "$100 laptop" that uses underpowered and scaled down resources in order to be more affordable in more parts of the world. Our "from the metal to the end-user" design and implementation will use an architectural approach that is rather different from the standard stacked layers of monolithic code from "operating system" to "applications" to "user interface". We still have to deal with issues of resource allocation and creation, internet communication, protection, graphics, sound, etc., user interface, end-user experience and functionality, etc., but we will use quite a different approach to design and underlying architecture.

"Operating Systems": We want to be able to do what operating systems do (and more), but we think there are better paths than to make a traditional classical operating system as layers of code.

Internet inspiration: Neither the computers on the Internet, nor their operating systems, nor the objects in their software systems have to be identical in order for the system to work. Instead, the heterogeneous mixture that the system is made from must simply obey message passing conventions in order to interoperate. We can note in passing that one of the biggest problems in the development of object-oriented SW architectures, particularly in the last 25 years, has been an enormous over-focus on objects and an under-focus on *messaging* (most so-called object-oriented languages don't really use the looser coupling of messaging, but instead use the much tighter "gear meshing" of procedure calls – this hurts scalability and interoperability).

Once we have truly committed to "real objects", many things are possible. For example, we can safely deal with real objects that are written in other forms via a combination of message-passing and address space confinement (we will say more about this later on). In particular, we can modularize one of the most pernicious parts of making a new operating system – namely drivers – by noting that device drivers should never have to be part of any well designed system. This is because a much better way to organize drivers is to have the devices themselves be able to furnish them to a system that needs them in a form that can be run and used safely. Again, we see that the key here is to have both the device and its drivers be logical objects on the network (and hence already able to use the universal messaging conventions that are our sole standard). An imported driver can then be run in a separate address space for safety and communicated with via standard messages.

So we will need to have some of the objects of the system be able to deal with various kinds of low-level hardware resources and help make a nicer environment for other objects and their needs. We have done quite a bit of this kind of system building over the years and explain the gist how this will be done in the section on bootstrapping ahead.

"Applications": We want to be able to do what applications do (and more), but, as with "operating systems", we think there are better paths than the traditional annoying stovepipes that give rise to a few proprietary objects in a way that makes it difficult to combine. Something more like a desktop publishing system that could allow any and all objects to be freely combined visually and behaviorly would be much better. Just as a DTP system allows many different visual elements to be formatted in a wide variety of ways (and master templates made to capture the most useful forms) to cover the entire space of user documents, we would like to go farther in this direction to cover all of the end-user's needs with a single notion of objects, graphics, user interface, publishing and search. One metaphor that might help (and was an inspiration for many of these ideas) is "HyperCard on Steroids". To do this one would extend HyperCard to have any number of useful objects, allow all to be scripted, and allow the hyperCards to be both full-fledged media pages for docs, web, and presentations, etc., **and** to recursively be its own embedded media objects. We have made systems using this approach over the last 10 years and will show a few examples further on.

Graphics, Sound, etc.: The system will provide its own low-level media models to be carried out on the hardware of the host machines. The current Squeak Smalltalk supplies a variety of graphics facilities, including a modern version of the original historical "bitblt" with rotation, alpha-blending, multi-depth color, etc., and vectorized 2D and 3D objects, roughly equivalent to Flash and OpenGL. One of the more interesting included cross platform higher-level models is an MPEG player. In our proposed new system, all of these will be consolidated into a much

simpler scheme that provides a little more functionality with much less code (some of which is described further on).

Printing: We finesse the problem of printer drivers above, but we can't finesse the problem of being able to print from our system via drivers we have obtained from the printers. We think it is fair if this is solved by generating postscript document format from our media.

The Internet And "The Web": We will certainly provide connection to the Internet, but, because the web architecture is so *ad hoc*, we will instead provide an equivalent simpler functionality that can work within the existing web. For example, a really great web-like experience would be to be able to find, view, and WYSIWYG author arbitrary multi-media "pages" that are hyperlinked, searchable and allow "services". This is really easy, and the web could have been easily designed this way, given that HyperCard was already around and thriving to provide a suggestive model when the first web browsers were made. We will follow that ignored path rather than the existing one.

Protection and Safety[Prot]**:** An important point about "real objects" is that they are already protected (they will only receive messages that they want to receive, and they will only act on messages they want to act on). The early notion of capabilities as an unforgeable filter on privileges for particular objects fits very well into a real object scheme. The capability approach to modern protection via message sending which give rise to *promises for future transactions* is very compatible with our work and with David Reed's scheme [Re] for distributed transactional protected objects that has been extensively used in the Croquet [Cr] system over the last several years (more ahead).

End-User Experiences: For the end-user, protection and safety are intimately bound up with being able to recover from any and all errors. In this system we want to go very far in dealing with "invertible processes", in part because the self-disclosing nature of the design will encourage the end-users to both look at all levels of the system and to try things, many of which would have catastrophic consequences without good facilities for real-time checkpointing, versioning, undo and redo.

User Interface Environment: interaction, widgets, etc. In addition, we think that a large part of programming language design – not just for novices, but for all who program – is treating the language and how it is worked with as *user interface design.* We address this further in the next two sections.

Computations And Programming: we wish to have as simple as possible scripting "all the way down" and we want to be able to allow multiple styles of programming to work under one logically consistent framework. We also want to experiment with and invent new ways for both end-users and professionals to program. Examples of traditional novice and expert end-user programming are spreadsheeting, HyperCard-like scripting of media, searching, making templates for new media types, etc. We will combine these into a single metaphor.

Non-traditional end-user programming includes various kinds of problem solving via conditions and constraints, massively parallel "particles and fields" techniques, "-ductions" (de-, in-, ab-, etc.), etc. Some of these will be in the system as mainstream ideas, e.g. we have had enough experience with "particles and fields" and how this style can extend from prototype programming to massively parallel loosely coupled, to have many parts of the system programmed this way. Less tested new ways to program will be exhibited as alternative curiosities. One of the most interesting (we think) alternative programming methods is how the system itself is bootstrapped (more about this ahead), and allows down to the metal changes and additions to be made as new needs arise.

Some of our best results have come from adding novices, including children, into the mix of users that need to be served. This anticipates one of the 21st century destinies for personal computing: a real computer literacy that is analogous to the reading and writing fluencies of print literacy, where all users will be able to understand and make ideas from dynamic computer representations. This will require a new approach to programming.

Explanation and Self-Disclosure: All the personal computing systems we are aware of, including our own, are very poor at explaining themselves, even to professionals, and are essentially opaque to non-expert users. A number of AI and expert systems have employed filtered versions of deductive histories to produce a narrative explanation of their inferences [Exp]. We have experimented with these ideas to allow any structure to explain and show how it was made. These simple explanations can be annotated to produce richer and more forgiving renditions, but the real value in explanations are those that can be generated automatically in a setting that allows end-user exploration. Our particular approach to modeling time (see ahead) also allows extensive undo, redo, checkpointing, and the ability to run computations both forwards and backwards to understand what is going on. Our aim here is to have "the system be the curriculum". This will eventually require this system to go beyond being reflective to being *introspective* via a self-ontology. This can be done gradually without interfering with the rest of the implementation.

2. Design Approaches

Pluralitas non est ponenda sine necessitate - William of Occam
Things should be as simple as possible, but not simpler - Albert Einstein

An important process of Design is finding the fertile ground between Occam and Einstein. We can make the physical universe from a few elementary particles and fields, but the myriad combinations of these at every level seem to give rise to quite a bit of complexity and a large number of categories[De]. Similarly we can make "computing" from a few simple relations, but this doesn't prevent systems bloat. In Biology [Bio], one of our favorite sources of fruitful analogies, the scaling of entities is not smooth but jumps from rather small carbon based molecules to much larger entire cells that can play many kinds of roles derived from very similar architectures. Looking ahead to even more interesting possible analogies with Biology are the recent advances in understanding developmental processes of multicelled animals. Quite contrary to the assumptions of biologists in the 60s – when it was thought that structures, such as eyes and legs in insects and vertebrates were likely results of parallel evolution – it has now been discovered that the basic "tool-kit" genetic apparatus for building "bodies that work" is shared by all animals (e.g. the gene that promotes the creation of an eye in a mouse will promote the creation of a (fruit fly) eye in a fruit fly.

These large plateaus for stable structures suggest we take a similar and somewhat "theatrical view" of a system in which every entity at every level is portrayed by an intelligent actor wearing appropriate costumes and simply playing a role. Here we are "not multiplying entities unnecessarily", but are putting the burden on a single kind of object (and we hope that *it* can be explained simply enough to make the larger system much easier to understand than if it had been built from many thousands of seemingly different entities. The idea here would be to jump from primitives (derived from the HW) as directly as possible to "comprehensively capable objects" and to differentiate these in ways analogous to the 250+ derived cell types in our bodies, which are all variations of the original fertilized cell.

And, since much of "personal computing" is necessarily about the experiences of "everyone as an end-user" it is important at the user interface level to create the illusion (if not the reality) of an extremely simple but wide ranging environment for making and dealing with all manner of manifested ideas. In fact, we can go farther and say the end-users are the only important faction to serve, and the value of a personal computing system comes solely from how well its users are served.

This implies that our "actors" must be comprehensible by the end-users, and the user interface and user experience is a good starting place to invent the single kind of actor.

Many of the useful entities in the user experience can be viewed in more than one way (for example, a spreadsheet cell could show a number or a visual bar; an article in a browser could have a simpler "suitable for printing" view, etc.). Are these "views" separate kinds of entities, or could they be the very same kind of actors? That is, are costumes different from humans (they certainly are in virtually all theatrical experiences) or could a "costume" also be portrayed by an actor?

In this project, beauty is especially important since we intend to allow every part of it to be examined, explained, de- and reconstructed. Combining our need for beauty with our other goal of "extremely small size without giving up power" reminds us immediately of great ideas from the past, such as. McCarthy's "Lisp in Lisp", Sutherland's separation in Sketchpad of "constraints as measures" from its solvers, Engelbart's comprehensive approach to "augmenting human intellect", Irons' insights about compiling and bootstrapping extensible languages, Strachey and Landin's approach to user friendly forms for functional programming, Atkinson's end-user UI model in HyperCard, etc. These are a few examples drawn from a *rich context of prior work* by others which include about a dozen powerful principles that we'll use to design and build this system (see **Technical Notes**, page 9, for references).

"The Many" From "The One": At the first level of authoring we want to just pick useful objects from a "supplies bin" and organize them "WYSIWYG" as we see fit to make our best statement about our topic. Hypercard has been a main influence here [HC]. We will have printing quality text in a myriad of fonts, pictures, videos, hyperlinks of various kinds, constructions of simulation models, etc. We want to make these ideas public with a single button push, as "email", as a "blog", as a "web page", etc. We want many ancillary kinds of communication to be possible, including "pen-pals", mentoring via screen-sharing, chatting, return emails, annotations and comments, etc.

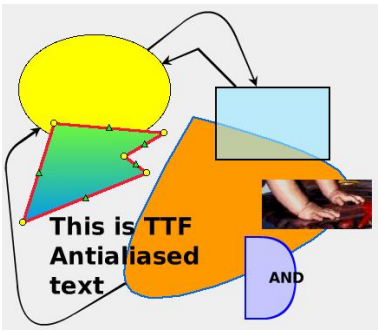
Both this first level of authoring and the next level (how all these useful objects are made) can benefit by an extreme unity in the underlying design. For example, we certainly want them to have the same behaviors as far as laying out our document is concerned. But, as we will see, they can also (a little more covertly) be the very same kind of object underneath, rather like actors on the stage wearing different costumes (including the scenery!) but all portrayed by human beings. Here are a number of examples done in our existing children's authoring system Squeak Etoys [Et].

All Objects Are "The Same" and Recursively Embeddable

At the end-user level, all objects seem to be made from a single kind of object – visibly, a kind of smart polygon that can have holes, be smoothed and filled, recursively embedded and scripted, etc. Like the theater in Shakespeare's time, in which even the trees and other scenery were people wearing costumes (pretty handy for changing scenes), it is possible to have all the objects be "the same" where the slight differences are in look and specific role, and more parametric than in specie.

"No Apps" – Instead, Any Of The Existing and Created Objects Can Be Arbitrarily Combined

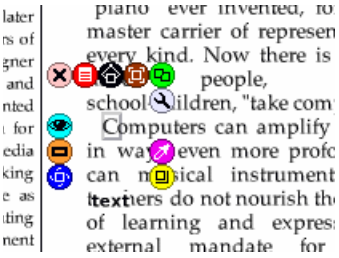
Separate applications are an old 60s idea which force a "stovepipe" mode of authoring that is limited to what the application can do, and this makes new ideas by end-users difficult to fit in. Looking at it from this point of view, there is only one "application" in our model – itself – and all old and new things are created, manipulated and presented there.



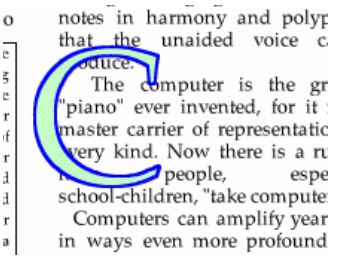
The many graphical costumes are all made from the same universal shape, and include ovals, rectangles, polygons and curves, pictures and drawings, TTF antialiased text that can flow from one container to another (for DTP), and smart connectors for makng diagrams. Any object can hold any other

"Desktop Publishing" is not an "application" but simply an organization of desired objects to look like a high-quality printed page. Any old or new objects can be simply dragged to become part of this "document".

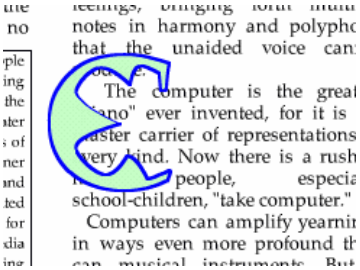
All the objects are "the same object". Here we click on a frame that contains text flow.



We get the same halo on the capital "C"

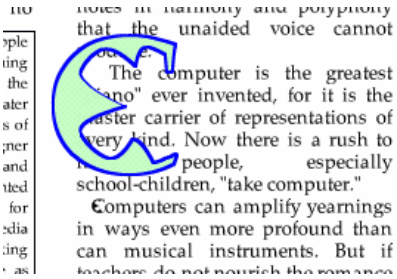


Diving into the character gives us the smoothed polygon that is the basis of all 2D graphics (and TTF Fonts).

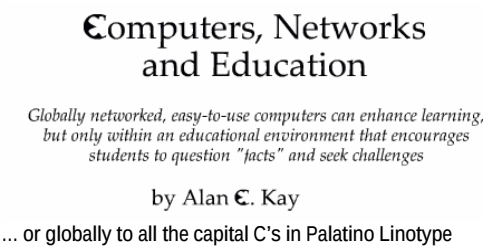


This is itself editable

... and it is a standard Etoy object, but playing a different role.



We can apply the change locally to our selected capital C...



... or globally to all the capital C's in Palatino Linotype

All Objects Are Scripted "All The Way Down"

Our model system will use a deeper refinement of the children's environment, but it will have enough of a similar flavor to motivate showing a few examples here. First let's paint a simple object, explore it, and then think about what things would be like if all of the objects in the system were the same.

The Handle Brings Up An Object's *Viewer*

Every object's viewer shows all of the object's properties and behaviors organized into "categories" (or "traits" or "roles").

We can paint a little red car

... show its handles

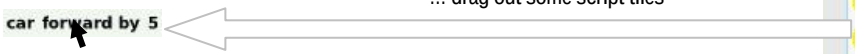
... get its viewer

Object viewer with some of the standard categories of properties and behaviors

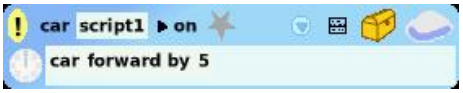
All Objects Are Scripted The Same Way

Scripts are made by dragging out tiles onto the desktop, and then dragging tiles into the script. Syntax is always correct.

... drag out some script tiles

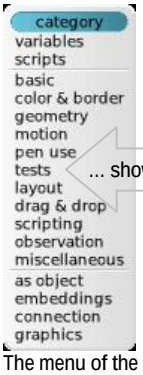


Tiles dragged from the viewer and dropped on the desktop ...



... make a script

The green "destination marker" shows up when tiles are dragged over a script.

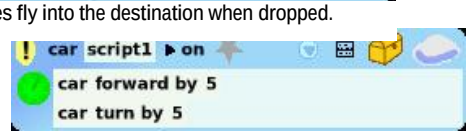


... show a role

The menu of the standard roles

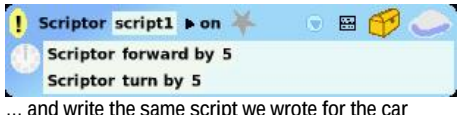
The car will move in a circle when the script ticks. If we have made "pen down" (in the "pen use" category) true, then the car will leave a trace as it moves.

The tiles fly into the destination when dropped.

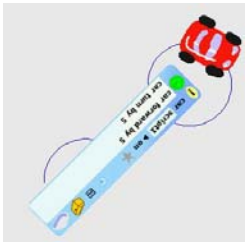


The script can be started "ticking" by clicking on the clock or by telling it to tick)

Because the script is a standard Squeak Etoys object, we can get its handles, and its viewer ...



... and write the same script we wrote for the car



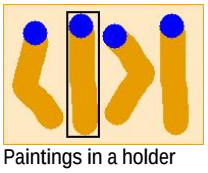
The result is that the scriptor will move in a circle and leave a pen trail of its own

We can do the same thing for any Squeak object, including a category in a viewer.

Or a movie even while it is playing.

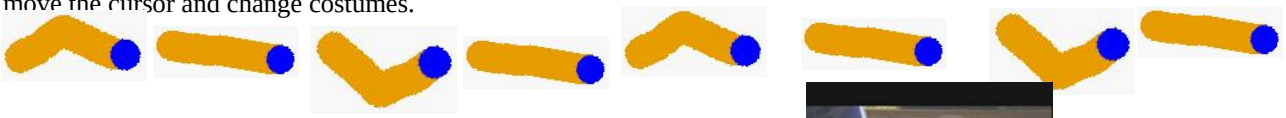
Animations Are Just Costume Changes …

To change our car into an animated worm, we paint some worm costumes, drop them into a holder and write a script to move the cursor and change costumes.



Paintings in a holder

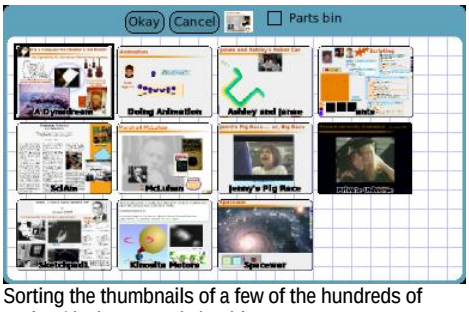
Script changes costume and moves the holder's cursor to the next painting.



… Movies and Videos Are Just Animations

Now we should realize that we have also made the guts of a movie player: we can just drop frames from the movie into a holder, write the two line script and it will play them.

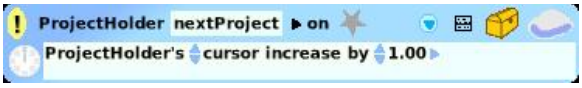
What's missing? Movies usually have hundreds to thousands of frames, each of which has lots of pixels. These frames are usually held as a file on the hard drive and are brought into the player application. Squeak Etoys provides automatic services for relating contents of files to Etoy objects, and also to compress and decompress pictures.



Sorting the thumbnails of a few of the hundreds of project/desktops made by this user

… Docs and Presentations Are Just "Animations"

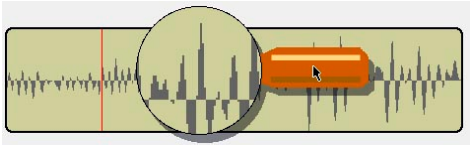
The Etoy project/desktop is where things are made, and the end-user can have any number of them. They also act as "pages" for documents, presentations and the web. The pages are sortable by hand and by script, and each sorted sequence can be named and used – this allows many different presentations and "books" to be made up from the same materials. *These "pages" are just standard costumes on regular Etoy objects.*Squeak "Book" (multipage documents that are like a HyperCard stack) and Squeak Presentations (like PowerPoint but more powerful) are the very same kind of structure. The "pages" can be any object (including a whole project), and turning is done by a script that looks like:



Project "page" shown full screen. All the authoring facilities are available, this is a live project, not just an image.

… Sound Synthesis Is Just An "Animation"

Now let's look at a different sequence in a holder. With the sound recorder we record a tone and see it is a sequence of bars, whose height indicates the sound pressure at that time.



One of the objects supplied with Etoys looks like a loudspeaker and if we move this object, the physical loudspeaker will move. We write a little script that is like the animation scripts, but instead of doing a "looks like" we will move the loudspeaker according to the height of the current bar, and we hear the tone! If we change the "increase by" to 2, we will hear the tone an octave above, and if we change it to 1.5 we will hear the tone a fifth above. We have just made a "Model T" real-time synthesizer.

Massively-Parallel Typical Element Programming: “Particles” and “Fields”

The metaphor of particles and fields can be applied to many ideas and situations – particularly given that the foundations of physics are couched in these terms, and thus “pretty much everything” can be represented this way.

A biological example starts with a salmon trying to swim upstream to its spawning grounds.

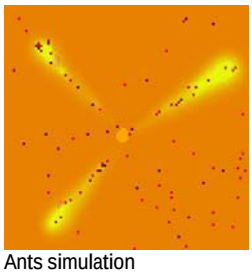
Virtually all biological organisms are able to follow gradients using almost no memory and an extremely simple (and very profound) strategy: if things are OK, keep going, otherwise tumble!

Here we have a “fish” looking for a more concentrated (darker) chemical. The gradient is not as smooth as it looks, so ...

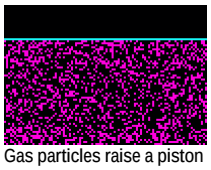
... we see there are a number of places where tumbling took place. But the upper right corner was still found pretty efficiently

It’s helpful for most learners to think through gradient following in the large with a single organism. But once done, there are many analogies with thousands and millions of particles that use gradient following for both efficiency and sending messages. For example, ants wander randomly looking for food. When they find it they carry it back to the nest and lay down a scent trail of chemicals that both diffuse and evaporate. If another ant that is foraging finds the scent trail it can follow it upwards towards the food source. The more interested ants the more scent and, if the scent evaporates slower than the supply, this means there is a lot of food and the bloom of scent will attract many ants.

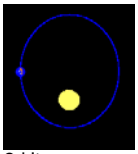
From a systems perspective, this is a graphic and interesting example of loosely coupled coordination of massively parallel agents. The “field” is a virtual object, and can be represented by mathematical relations, message passing and events, or, in this case, the “field” diffusing and evaporation of the scent is done by 10s of thousands of stationary particles that form a background grid (they are essentially communicating finite automata). The Etoy system can potentially handle about one million graphical particles at 10 frames per second on a laptop.



Ants simulation



Gas particles raise a piston



Orbits

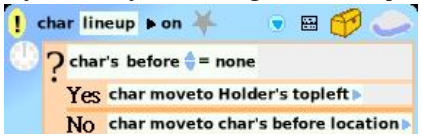
Newton’s Gravitation Law as a child’s script

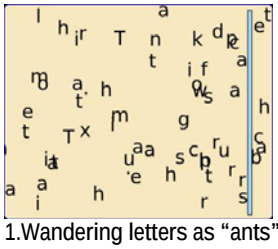
A particle per pixel

Massively parallel blurring

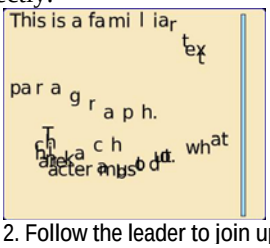
“Particle & Field” Text Paragraph in Etoys

Dynamically formatting the text is quite straightforward: the scripts tell the story directly.

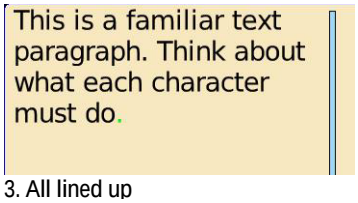




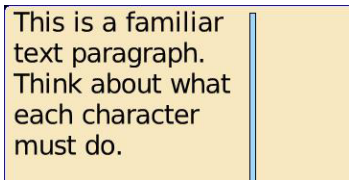
1.Wandering letters as “ants”



2. Follow the leader to join up



3. All lined up



4. Moving margin causes text flow

3. Technical Notes

We have now looked at the general goals **(1.)** and gist of the approach to design **(2.)**. Here we will discuss a few of the dozen or so "powerful principles" we will use to actually implement the plan. In the "everything is one" approach of this kind of design, we think of these principles as being aspects of a single principle, but which are broken out separately here to allow easier discussions of particular points and references to past influences.

For example, the idea of *polymorphic methods that establish a kind of algebra* [Alg] has been around since Sketchpad (though it is not used nearly strongly enough in most actual systems today). Our plan to use a rather *capable fleshed out object as a general agent* [GA] (a relatively new idea) with *sideways composition of roles* [Ro](dates to ca. late 70s at XPARC), for realizing most of the object behavior in the system, provides an extreme polymorphism, and will require a careful design that covers as much meaning in as understandable a way as possible. Another long standing idea that is known in some circles to be very powerful is *late-binding of everything* [LB], including the processes of development, underpinnings of a language and system, targets of communication, abilities of users to safely and instantly modify anything at any time, and so forth.

Programming in "Meanings": A newer idea that is moving towards the mainstream is that *specifications should be executable and debuggable* [Spec]. We want to go even further to "ship the specifications" – that is, the specifications should not just be a model of the meanings in a system, but should simply be the actual meanings of the systems. This leads to a corollary idea: that we should have an *absolute separation of meanings from optimizations* [SepM]. A simple example is that it is possible to define sorting in Prolog as a permutation of the inputs that obey a particular pairwise relationship. This is really simple and will work, but Prolog has no recourse but to generate all possible permutations of the input until one is found that satisfies the predicate. We certainly would like to have a way to use the myriad of heuristics devised for sorting to help things out. On the other hand, we might hate to give up our nice simple definition of what sorting means. This leads to the idea that any simple definition could be used to check heuristic optimizations.

In other words, though we want the meanings to be stated in as simple and powerful form as possible, and to be converted by the system into appropriate actions, the system will often not have enough information to automatically do the best practical approach to actions. The meanings should allow the system to be debugged and tested, and we want to safely deal with the ancillary pragmatic needs via separate mechanisms that will be automatically checked by the meanings. This idea has been used in logic (C-FOL and CYC) and relational programming (Thinglab and Kaleidoscope), but has not made it into mainstream system building.

Preservation of Meanings: Another set of related ideas that we want to bring into systems programming have to do with *meaning what we mean, and holding on to those meaning* [MM]. We want to enlarge the definition of "truth" to include modern scientific and engineering senses of the term. For example, Sketchpad, the ancestor of all graphics systems, allowed dynamic constraints to relate properties of its graphic constituents. Sutherland realized that (a) it would not be possible to reasonably try to solve all the constraints using symbolic math and logic, and (b) that approximate solutions within fixed error tolerances "engineering style" would be just fine in part because errors smaller than a pixel would not be noticed. So "truth" in Sketchpad was "engineering & scientific truth" (which means there are errors and tolerances attached). And "maintenance" meant continuously trying to get the errors below tolerance. Later, Knuth used a variation of these ideas in TeX to great effect.

A *set of support* is the collection of antecedents for some derived relation. *Truth maintenance* in a dynamic inference domain would require that relations be not just added but also subtracted when their set of support gets changed. We will apply this principle to the system's own meanings and programming.

A related principle is *Observe & Transform* [OT] (vs. Force and Smash), which takes ideas from functional programming, spreadsheeting, event-driving, forward inferencing, "needs-driven" invocation, loose coupling, etc.

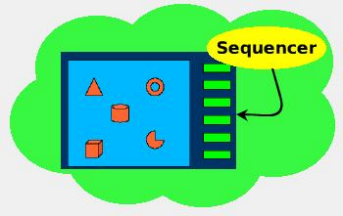
Massively Parallel Objects: Even less in the mainstream, the "*particle/field*" idea [PF] has been found more in specialty areas (such as finite-automata and FEM, swarm programming in biology, etc.), than as a general system building tool (although it is the center of systems such as Sketchpad and TEX, and is even found in an operating system such as MUSE). Traditional object-oriented design has tended to overbalance its attention on the objects and to give too rudimentary attention to message-passing. If the center of attention were to be shifted to messaging of all kinds, then the notion of "fields" immediately suggests itself as a more general way to think about inter-object relationships (the previous example of "ants distributing and sensing pheromones" is a good metaphor of this style).

We have been experimenting with writing many kinds of programs in these styles (see some toy examples in the previous section), and think the metaphors can be elevated to a general facility of "establishing dynamic relations within an automatic message/event passing environment" that includes: typical element programming, constrained relationships, inferencing of different kinds and directions, and *explicit models of time* (explained next).

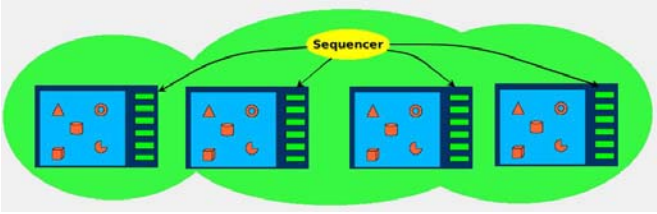
Time is nature's way of preventing everything from happening at once

Explicit Models Of Time: Traditional computing lets time be generated by the CPU and then tries to deal with concurrency, race conditions, etc., via mutual exclusion, semaphores, monitors, etc. This can lead to system lockup. Similarly, traditional logic is timeless, and this makes it difficult to reason about and produce actions. Ideas about "time" from Simula and McCarthy's situational calculus provided similar alternative solutions in their respective domains: namely, create *pseudo-time*, a model of time that uses a form of labeling of states to separate the acts of computing from pseudo-time flow. A variety of insights along these lines (Strachey, Wadge & Ashcroft, Hewitt, Reed, etc.) lead to massively parallel models that can distribute "atomic transaction" computing amongst local objects, and over networks. [EMT]

Several years of experimental systems building (Croquet [Cr 1]) have led us to efficient scalable practical "worlds of objects" (called "Islands" that make up "moveable replicable synchronizable (in pseudo-time and near real-time) aggregates of safe parallel computing". In practice, a 2D project-page-desktop or a 3D world in Croquet can usually be represented by an underlying island of a few hundreds to a few thousands of objects. Islands impose a pseudo-time discipline on their internal objects and intra-island communication. An island can have many replicas across networks of machines and they will perfectly perform atomic transactions in pseudo-time. Islands that get isolated will resynchronize to the most current world lines of themselves. Because the messaging discipline is absolute and "Internet-like" (an island is a kind of super-object that can play the role of a machine on a network), islands can be used to mix vastly different object systems into the same computations. And so forth.



Logical Island with objects, sequencer and events



"Physical Island" with transactional replicas of contained objects (often located on separate machines).

The messaging and labeling discipline imposed on islands and their internal objects also allows better undo and redo models, checkpointing, past and future histories of formerly unrepeatable events (EXDAMS, etc.), "possible worlds reasoning "and "truth maintenance" at both fine and coarse grains. These help computing, programming, debugging, and the generation of explanations, etc.

Another principle that originally harks back to the original Lisp, and is used in deep ways in Smalltalk, is that *systems should contain complete operational and late-bound enhanceable models of themselves* [MOP](Meta Object Protocol). This principle can be intertwined with the above mentioned: *specifications should be executable and debuggable*, and with our final principle: *completely late-bind the bootstrapping of the fundamental objects of the system* (see ahead).

The combination of these three principles makes up a very interesting research area of "deep and practical" computer science and software engineering. On the one hand – like McCarthy – we'd like to write down a half page of mathematics and have it represent the semantic intentions of our system. On the other hand, we require that the half-pager be transformable to a kernel that runs "really fast" on a wide variety of platforms.

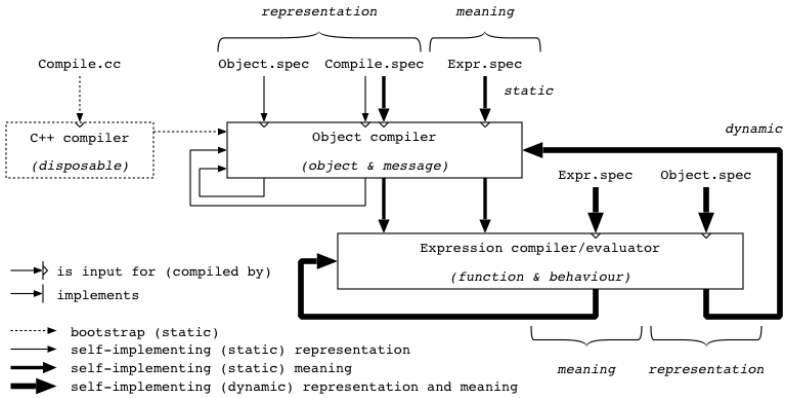
It is the grand object of all theory to make these irreducible elements as simple and as few in number as possible, without having to renounce the adequate representation of any empirical content whatever. – Albert Einstein

"From Nothing" Bootstrapping: We and others have done many systems over the years that use realizations of recursively embedded bootstrapping to quickly bring up and/or port a system with its own assistance [FNB]. Our most recent "from nothing" bootstrapper – called "Albert" – is able in a few hundreds of lines of code to make a kernel for Squeak Smalltalk that runs about 9 times faster than the existing interpreter (about 80% the speed of C++).

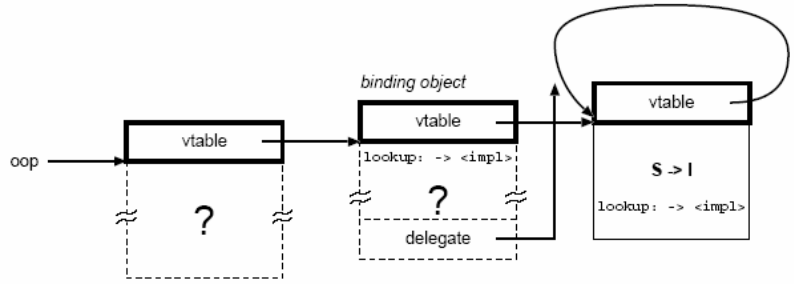
The kernel of our bootstrapper is a simple self-describing dynamic "object engine". Mechanisms in the kernel are limited to those that are necessary and sufficient for it to describe its own structure and behavior. Because this description is complete (every detail of the kernel is fully described) the whole system becomes pervasively late-bound, able to modify dynamically any part of itself from within.

Key to the tractability of this approach is the separation of the kernel into two complementary facets: representation of executable specifications (structures of message-passing objects) forming symbolic expressions and the meaning of those specifications (interpretation of their structure) that yields concrete behavior. Concrete behavior describes the messaging operation itself (the unit of communication between objects) and the sequencing of messaging operations within an object's response to a given message (method implementations, reified as first-class functions).

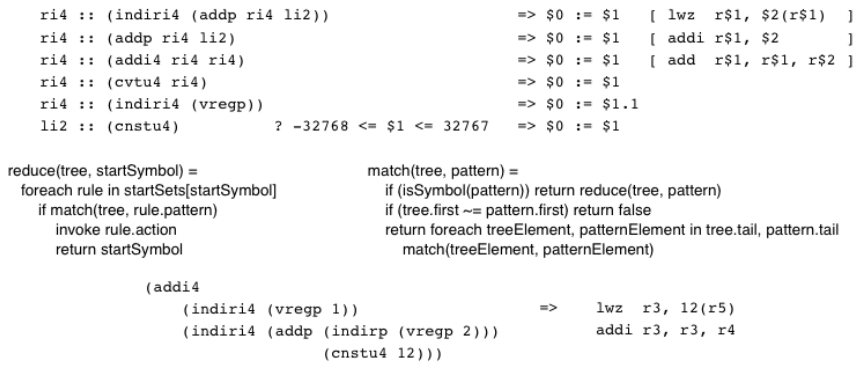
Representation and meaning are thus mutually supporting. Bootstrapping them into existence requires artificially (and temporarily) introducing a 'fixed point' in their interdependence. We accomplish this by writing a static approximation to the object and messaging implementation in which object structures can be created and interpreted as symbolic expressions to yield concrete behavior. Once complete, the resulting system can generate behavior that completely replaces its own original approximations with a fully-fledged dynamic implementation of messaging and methods. This event is a "singularity": once reached nothing can be known about the original approximations (they are, and should be, irrelevant); the resulting kernel is both self-sustaining and amenable to expansion from within towards any form of end-user computing environment.



Bootstrapping the circular implementation in Albert. A disposable compiler (written in C++) implements a simple message-passing object-oriented language in which a specification-based Object compiler (impementing the same language) is implemented. The system is now self-implementing but still static. A dynamic expression compiler/evaluator is then implemented using the static compiler and used to replace the static messaging mechanisms with dynamic equivalents. The system is now self-describing and dynamic – hence *pervasively late-bound*: its entire implementation is visible to, and dynamically modifiable by, the end user.



The kernel object model in Albert. A single principle obtains: every object is associated (directly or indirectly) with a binding object (mapping messages to methods) that describes its behavior. For objects physically extended in memory a pointer to the binding object is stored one word before the start of the object (to preserve identity relationships between encapsulated "foreign" objects). Nothing is assumed about the contents of an object (its state, if any, is manipulated only by the methods stored in its binding object). An "intrinsic" vtable stops the recursion by defining its own contents as a trivial method dictionary (a point of circularity in the object system). Sharing and reuse mechanisms (such as delegation) are not part of the object model *per se* but can be added simply by overriding the "lookup" operation.



Part of the expression compiler in Albert. Code generation (which might occupy tens of thousands of lines of code in typical compilers) can be made compact and elegant. Simple "bottom-up rewrite" rules (top of figure) describe the capabilities of a target machine in terms of physical resources (non-terminal symbols in the grammar) and abstract operations (the terminal symbols). Actions associated with each rule generate code and control how synthesized attributes are propagated upwards during rewriting. Top-down rewrite rules convert an object structure equivalent to the expression `s+v[3]' into an abstract "structured machine code" (bottom of figure). A simple algorithm (middle of figure) applies the bottom-up rules to the abstract machine code reducing the tree to a single non-terminal, emitting the code shown bottom right in the process.

Within the kernel the number of algorithmic abstractions is kept to a minimum. The mechanisms employed are designed to be generic and reusable. For example, the classic compiler problem of register allocation during code generation is a variant of the text formatting problem described earlier (where the letters become the nodes of a tree requiring registers, constrained by "register pressure" rather than by the width of a page). We believe that this, combined with several "inferencers" applying top-down and bottom-up rewrite rules to object structures, can elegantly and succinctly dispense with the vast majority of the complexity that would otherwise be required to implement the expression compiler component of the kernel.

4. The Implementation Of This Proposal

Our Research Environment: Viewpoints Research Institute is a 501(c)(3) public benefit organization that was set up to fund exploratory research processes similar to the classic funding in the 60s from ARPA and ONR, and in the 70s at Xerox PARC. We operate in a community that includes our own researchers (including grad students) and advisors, universities and university colleagues (e.g. UCLA, Kyoto, MIT, etc.), industrial laboratories, and a wide pool of open-source computerists around the world. The funding we secure is not tied to specific outside goals, but is used to fund the vital problems we think should be worked on. This means that, when needed, we can aggregate our resources to support larger efforts we think are important.

We mostly design and build whole systems in which a lot of attention is paid (a) to the end-users (especially children), and (b) to improving systems architectures. In the simplest sense, we think that the real personal computer revolution is yet to come, and we are trying to help invent it, in part by showing the mainstream community there are important alternatives to many of the choices that have been made over the last 20 years.

Motivations and Inspirations: One of our favorite inspirations is John McCarthy's "Recursive functions of symbolic expressions …" which outlined a theory of programming that could also be thought about mathematically. The argument led to a half page *eval* for the system expressed in the system. As John said in his history of Lisp talk, "… when Steve Russell saw that, he said, 'Why don't I program it and we'll have an interpreter for Lisp?'And he did, and we did". [Mc]

What was wonderful about this approach is that it was incredibly powerful and wide-ranging, yet was tiny, and only had one or two points of failure which would cause all of it to "fail fast" if the reasoning was faulty. Or, if the reasoning was OK, then the result would be a very quick whole system of great expressive power. (John's reasoning was OK.) In the early 70s two of us used this approach to get the first version of Smalltalk going in just a few weeks: one of us did what John did, but with objects, and the other did what Steve Russell did. The result was a new powerful wide-ranging programming language and system seemingly by magic [Ka].

Similarly, the present project has very few points of failure. It is based on about a dozen ideas we think are powerful, each of which has to do the job of hundreds of the more standard techniques used in most system design and implementation. This means that flaws in any of the ideas will also "fail fast", and if the ideas are "OK", then quite a bit of the system will "bring itself into being" very quickly.

The "Open Source Way" To Build Systems: Over the last decade, we have made 4 major systems "the open-source way", distributed them over the Internet, and helped set up open-source communities who enrich, document, and maintain them. The size of these communities varies from several thousands (Squeak and Etoys), to hundreds (Croquet) to around 50 (Tweak). Each of these artifacts is disseminated to all, and is maintained, documented and improved by each community. [VP]

As with others who have experimented with this relatively new process for getting large things done via virtual communities of interest, we are enthusiastic about the benefits, and quite confident that "if one great person in a million" would be interested in doing major work in a project, then the Internet can find several hundred of them!

Things seem to go really well if there is a dedicated focused group who define an architecture, create a fertile kernel with alluring powers, ways to port to many platforms, and pathways for improvement. The central group becomes the keepers of the architecture while encouraging much experimentation, debugging, and improvements that leverage the kernel, often in unanticipated ways.

This process is rather different from the expectations of classic funders (such as the NSF), where the general practice is to identify the team that is going to accomplish all the goals. The new dynamic, especially for systems of large scope, is to fund a kernel group (most of whose members can be identified), and expect (and trust) that the open-source dynamic will provide the (sometimes hundreds) of additional talented people to "put the leaves on the tree". As in our previous "design, invent and implement" projects, we are relying on the formation of a community to help do the myriad of tasks required to realize the aims of this project.

The biggest difference in this project from others that we've done is the size limit on code, and the use of specific techniques to achieve the planned features within the size limit. This means that the part of the aims of the kernel builders will be to make a framework that will help the eventual open-source volunteers to work more closely within the design guidelines than is sometimes the case. For example, our plan to completely separate meanings from optimizations means much of the design and prototyping can be done by the kernel group via creating runable meanings, and the open source community can be invited to supply optimizations using the provided structures.

Dealing With Large Scale Multiple Goals: If we were to extract the intertwined goals of this project into separate projects, then we might have:

1. Just come up with a new way to bootstrap comprehensive language environments
2. Just come up with a new architecture for a complete personal computing system
3. Just do a complete personal computing system in 20,000 lines of code
4. Just make an improvement in programming, especially for beginners and end-users
5. Just make a big improvement in having a system be able to explain itself

Each of these would have engineering and research components within the NSF grant scope and outside it, and each would have work to be done, problems to solve and associated risks. If we liked to make lists and boxes, we could come up with a 5x4 matrix of the above 5 sub-goals against the 4 E&Rs inside and outside the NSF funding. However, we look at this in a more organic way, and some of the most interesting parts of this project have to do with interactions between what could be considered as parts (for example between 3. and 4. above). But let's discuss them as separate for the moment.

For example, we think we know what the new architecture should be for (2.), so, if we are right, the problems and risks would mostly have to do with range and scaling. That is, we know from our previous experiments that we can cover much, if not most (if not all) of the user experience with our omni-agent+roles+costumes "HyperCard on steroids" model without needing to use inheritance, etc. What we don't know is how far down, out, up and sideways this can be pushed as a comprehensive model for the whole system (i.e. inheritance with all its problems nonetheless does serve at all levels, we'd like a simpler cleaner alternative and prove it out).

For (3.), to get the whole system in "20,000 lines of code" we have to improve on what we have already done by a factor of 5. The last factor of 2 (from ~40,000 LOC to ~20,000) is an unknown at this point. We are very sure that the meaning of this system can be expressed in less than 20K LOC, but we are not completely sure how much invention will be required to do this. Our intuition tells us that this is doable, just on the grounds that expressing and running "the math" would do it. That is, on just this goal, we should be able to "*APL* it" and win.

(4.) is a demand to have the programming work for beginners and end-users, and (5.) requires the system to be understandable. We think that we have a complete (if too small) answer for (4.), in that the conventions have been tested on many end-users in a number of systems and work pretty well (to pick three besides our work: Logo, spreadsheets, and HyperCard). We are quite sure that combining this past experience with the simple comprehensive architecture in (2.) will be successful.

However, the reason this answer is too small for our tastes is that there is much more that really needs to be done to help end-users (and improve programming for all. We expect to do some of this research under the aegis of the NSF funding.

In many ways, self-explanation (5.) has more unknowns than the others, even though quite a bit of work has been done over the years. The biggest question is where to draw the line for the scope of the NSF part of the project. We plan to work with "great explainers", such as our advisors Richard Dawkins and Danny Hillis, and with Richard Wurman, the author of more than 80 explanatory books, and with colleagues in the Exploratorium and other science museums). We know that we can allow the system to be viewed at every level, there can be "tours", the system will be able to tell and show how any structure was made, that the undo mechanisms planned will allow both relatively static structures to be deconstructed and reconstructed and dynamic processes to be captured, traversed, rendered into prose, and "what iffed". Our current notion is that we will not try in the NSF part of the project to do an ontology and make an AI that tries to understand how the system works (this is on our list, however, for "ancillary research", as is a great interest in also making ontologies for design and mentoring).

Process: Small groups with specially picked personnel often manifest a different dynamic than large ones in which many of the members have to be somewhat "plug compatible" and quite a bit more up front planning is necessary. 5-8 people who have lunch together every day can deal more flexibly with a general plan and what is being learned as a project progresses.

Number And Kinds Of People: The simplest rule of thumb for projects like this with our style of approach (and style of people) is 6-8 full-time professionals and 4-6 grad students. This costs out to more than we are asking NSF for. So, to make deliberations (and this proposal) simpler, we have matched up, as well as we can, the requested funding (~$1M/year) with the personnel this will support (about 4 full-time professional equivalents plus 3 grad students), and the goals that just these people with just the NSF support can do. This is to avoid worries about the project failing if other funding from other sources doesn't come through. In other words, we want to be able to successfully come up with really interesting and worthwhile results with smaller than natural resources. However, we expect to use some of our other resources to follow a number of paths that would not be directly funded by this grant.

Late-binding has to do with *what* has to be committed to, *when*, and *how much* can be *gracefully changed later* in the process. Partly because computing (both "science" and "engineering") is still more of an explorable field than a developed one, and partly because Moore's Law gives us different scales to deal with every few years, most projects – even those that have been around for a while, such as "operating systems" and "programming languages" – have much more of a learning curve built into them than anyone would like. This means that critically needed new ideas that were really required at the beginning of the project might only be discovered 3/4 of the way through, and will only be useful if the system is late-bound enough to allow reformulation without a complete rewrite.

Less Well Understood Parts of the Project: The two areas of this proposal that don't have complete engineering practice and answers at its start are (a) the range of ways to express meanings, and (b) the range and ways to derive explanations. Thus a key part of the initial design is to build the system so we can add better approaches to "programming meaning and explaining it" as we go, while still having a good enough initial approach to meaning and explanation to allow the system to be built without *requiring* new inventions. In other words, we expect to do some successful research here that will advance several states of the art, but need to have these be late-bound enough to not require new invention to do the system at all.

And we expect to learn more about other key principles of the system, even though they are more mature. For example, our bootstrapping system, Albert, is precisely aimed so that we can late-bind new approaches to low-level scaffolding, including of and about itself. This sounds scary, but we put it in the category of "well enough understood principles" that can be deployed in an engineering manner even though it will itself be extended. For us, one the exciting aspects of this project is to be able to experiment with having a less conservative and much more flexible and late-bound approach to the lowest levels than we've had in the past (the bottom layers are traditionally the most difficult to change in major ways and get away with it).

One of the processes we've found very helpful over the years is to "always have a running system": that is, to close the loop of initial bootstrapping as quickly as possible so the system can assist in its further development. This is very good for both *morale* (the beautiful little baby is born alive and grows, rather than being a tacked together Frankenstein's monster that we are trying to bring to life after the fact), and for *reality* (what is being improved gets to be instantly integrated and debugged in the existing systems context so there are far fewer conflicts and surprises).

One way we've done this in the past is to use a previous Smalltalk system to make a qualitatively new and different one, by taking advantage of the message passing and dynamic storage structures to allow two different systems to

coexist. Again we have the analogy of the developing embryo within the mother. We can get the new system to come alive very quickly by scaffolding where necessary in the old Smalltalk. We can gradually replace the scaffolding by programming in the new system. When enough has been developed, Smalltalk can use its "systems tracer" - which can find all the objects and linkages of the new system - and "give birth" by creating a brand new virtual memory with only the new objects and code.

Similarly we have done many experiments with our new bootstrapping system by using it to replace the underpinnings of our working Squeak Smalltalk (and the current statistics of 9x execution over the Squeak interpreter and .8 C++ speed were obtained in this manner).

Milestones: Five years is a good horizon for a project like this, and quite a bit should get done and demonstrable in the first 18 months to 2 years. Our rule of thumb is that a project which requires operating system properties is at least a 3 year job, and allocating time thoughtfully on either side of this is a very good idea. Part of the fun and aesthetic of this project is to get "pretty darn complete" functionality within the code size we've set ourselves. Quite a bit of the effort in this project will likely not be in the kernel, but will occur in the last 10%-15% of the facilities, and we have to set up processes that will avoid "asymptotic bogging".

We will follow the philosophy and practice of releasing parts of the system "early and often" to leverage input and testing by members of our open source community. These releases would likely be decoupled, i.e., kernal release separate from UI design release(s), smaller code bases, etc. Our process will include journaling of the system to enable rollback to earlier versions and to teach about construction.

We would expect to schedule a major site visit about 18 months to 2 years out that would show a nicely working system with many of the main elements we've discussed, but lacking some of the facilities and optimizations needed for a successful result. The processes leading up to that site visit would be a combination of (a) "meaning engineering" of the main framework of the system, its graphics, GUI, etc., and (b) many more experiments in structuring, scheduling, pattern driven event dispatching, and history manipulation, etc.

Being able to avoid optimizations over this period is a critical part of the process, and a very good way to do this is to use a lot of hardware assist early on everywhere possible. For example, we can finesse our initial graphics model by using the most powerful graphics accelerators brute force, and much later do the various levels of optimizations that will allow the graphics to run on regular and underpowered hardware (one of our eventual targets is the quite underpowered "$100 laptop" which is aimed at children in the 3rd world). Similarly, we can use CPU aggregates with high bandwidth memory connect to finesse our early computation models, so we can concentrate on getting the maximum number of possible experiments in "meaning mechanisms" early on.

5. Dissemination, Education and Outreach

Viewpoints Research Institute has an established community of practitioners, learners, and developers with whom we work and continually communicate and disseminate materials. This community spans professionals concerning themselves with education and the use of computers in the K12 arena as well as undergraduate and graduate schools, and "non-traditional" learning environments such as museums and after-school community technology centers, many of these affording access to technology to underserved populations.

We share our prototypes and findings through a variety of media and sources including websites, international conferences and workshops. In addition to presenting at academic conferences, we co-organize two international conferences and host one invitational workshop annually. Our community is global; we have colleagues testing and helping to develop curriculum based on our research throughout the U.S., Japan, Europe, South America and Canada.

In addition, Viewpoints is a participant in the "One Laptop per Child" project which has the potential to reach millions of children and their teachers worldwide. Alan Kay's affiliation with UCLA, MIT, and Kyoto University and as an oft time guest lecturer at other Universities will allow outreach to these campuses and their students through teaching courses and lecturing.

During the course of our research we will work with members of these communities to develop and test our designs, and curriculum for a variety of ages of learners and in subject matter areas including mathematics, science, computer science, and engineering.

There is a clear need for a usable system by high school students, especially in AP (Advanced Placement) classes in the U.S. that can ready students for the C.S. curriculum they will encounter when they enter college or university. Currently we recognize a gap in available tools in this particular arena and see our work as providing a solution to this need.

References

Page 1 [Sq] Smalltalk and Squeak Smalltalk

A. Goldberg and D. Robson, Smalltalk-80: the Language and its Implementation, Boston: Addison-Wesley Longman Publishing Co., Inc., 1983.

D. Ingalls, T. Kaehler, J. Maloney, S. Wallace and A. Kay, "Back to the future: the story of Squeak, a practical Smalltalk written in itself," in OOPSLA '97: Proc. of the 12th ACM SIGPLAN Conference on Object-oriented Programming, 1997, pp. 318-326.

See http://www.squeak.org for open source website.

Page 2 [Mo] Models of Programs: Many and Different

J. McCarthy, "A Basis for a Mathematical Theory of Computation", in Computer Programming and Formal Systems, pp. ,1963.

B. C. Smith, "Reflection and Semantics in a Procedural Language," Ph. D. thesis, MIT/LCS/TR-272, 1982.

J. V. Guttag, James J. Horning, et al., Larch: Languages and Tools for Formal Specification, New York: Springer, 1993.

Kestrel Development Corporation, Specware 4.1 Language Manual, 2001-2004. [2006 Jul 5], Available at: http://www.specware.org/documentation/4.1/language-manual/SpecwareLanguageManual.html

B. Lampson, "Collected lecture notes from MIT course 6.826," Principles of Computer Systems, http://research.microsoft.com/Lampson/48-POCScourse/Acrobat.pdf

A. Diller, "Z: An Introduction to Formal Methods, 2nd Ed.," John Wiley & Sons, London, 1994

D. Jackson, "Alloy: a lightweight object modeling notation," in ACM Transactions on Software Engineering and Methodology (TOSEM), vol. 11, no. 2, pp. 256 – 290, 2002..

Page 3 [Prot] Protection, Security, Safety

R. S. Barton, "A New Approach To The Functional Design of a Digital Computer", in Proc. of the Western Joint Computer Conference, 1961.

J. Dennis and E. C. Van Horn, "Programming semantics for multiprogrammed computations," in Comm. of the ACM, vol. 9, no. 3, pp. 143-155, 1966.

B. Lampson, "On reliable and extendible operating systems," in Proc. of the 2nd NATO Conf. on Techniques in Software Engineering, Rome, 1969. Reprinted in The Fourth Generation, Infotech State of the Art Report 1, 1971, pp. 421-444. See Cal TSS.

B. Lampson, "Computer security in the real world", Presented at the *Annual Computer Security Applications Conference*, 2000

M. Miller, C. Morningstar and B. Frantz, "Capability-based Financial Instruments," in Proc. of Financial Cryptography 2000, 2000. [2006 Jul 05], Available at: http://www.erights.org/elib/capability/ode/index.html

M. Miller, K. Yee and J. Shapiro, "Capability Myths Demolished," [Paper], [2006 Jul 05], Available at: http://www.erights.org/talks/myths/index.html

Page 3 [Re] Dave Reed's Distributed Object Scheme

D. P. Reed, "Naming and Synchronization in a Decentralized Computer System," Ph.D. thesis, MIT, 1978.

Page 3 [Cr] Croquet papers

D. A. Smith, A. Kay, A. Raab, and D. P. Reed, "Croquet – A Collaboration System Architecture," [Online paper], 2003, [2006 Jun 19], Available at HTTP: http://www.opencroquet.org/about_croquet/papers.html

D. A. Smith, A. Raab, D. P. Reed and A. Kay, "Croquet: A menagerie of New User Interfaces," in Proc. of the Second International Conf. on Creating, Connecting and Collaborating through Computing, 2004, pp. 4-11.

D. Reed, “Designing coquet’s TeaTim: a real-time, temporal environment for active object cooperation,” in Conf. on Object Oriented Programming Systems Languages and Applications, 2005, p. 7.

Page 3 [Exp] Some approaches to explanation generation

W. R. Swartout, "The Gist Behavior Explainer," in Proc. of the National Conference on Artificial Intelligence, 1983, pp. 402-407.

W. R. Swartout "XPLAIN: A system for creating and explaining expert consulting systems," Artificial Intelligence 21, (3), Sept., pp. 285-325, 1983.

W. Teitelman, “INTERLISP,” in ACM SIGART Bulletin, Issue 43, 8 – 9, 1973.

Natural Language Generation in Artificial Intelligence and Computational Linguistics (The International Series in Engineering and Computer Science) (Hardcover) by Cecile L. Paris (Editor), William R. Swartout (Editor), William C. Mann (Editor), Kluwer, 1991

D. B. Lenat, "Cyc: A Large-Scale Investment in Knowledge Infrastructure," Comm. of the ACM, vol. 38, Nov., pp. 33-38, 1995.

M. A. Cohen, F. E. Ritter and S. R. Haynes, “Herbal: A high-level language and development environment for developing cognitive models in Soar,” in Proc. of the 14th Conference on Behavior Representation in Modeling and Simulation, 2005, pp. 177-182.

Page 4 [De] Design Approaches

B. Lampson, “Hints for computer system design,” in ACM Operating Systems Rev., vol. 15, no. 5, pp. 33 – 48, Oct. 1983.

Page 4 [Bio] Biology Influences

J. Watson, Molecular Biology of the Gene, Menlo Park, CA: Benjamin-Cummings, 1965.

B. Alberts, et al., Molecular Biology of the Cell, 4th Ed., New York: Garland Science, 2002.

S. V. Carroll, Endless Forms Most Beautiful: The New Science of Evo Devo and the Making of the Animal Kingdom, New York: W.W. Norton & Company, 2005.

Page 4 [HC] Hypercard

D. Goodman, The Computer Hypercard Handbook, New York: Bantam Books, 1987.

Page 4 [Et] Squeak Etoys

B. J. Allen-Conn and K. Rose, Powerful Ideas in the Classroom: Using Squeak to Enhance Math and Science Learning, Glendale, CA: Viewpoints Research Institute, 2003.

A. Kay, “Squeak Etoys, Children & Learning,” [online article], 2005, [2006 Jun 20], Available at: http://www.squeakland.org/pdf/etoys_n_learning.pdf

A. Kay, “Squeak Etoys Authoring & Media, [online article], 2005, [2006 Jun. 20], Available at: http://www.squeakland.org/pdf/etoys_n_authoring.pdf

Page 9 [Alg] Polymorphic Algebras For Programming

I. E. Sutherland, “Sketchpad: A man-machine communication system.,” in Proc. AFIPS 1963 Spring Joint Computer Conf., vol. 23, , New York: Spartan Books, 1963.

O-J. Dahl and K. Nygaard, “SIMULA: an ALGOL-based simulation language,” Comm. of the ACM, vol. 9, Sept., pp. 671-678, 1966.

A. van Wijngaarden, Ed., “Draft report on ALGOL 68,” Mathematisch Centrum, MR 93, 1968.

K. Nygaard and O-J. Dahl, “The development of the SIMULA languages,” in HOPL-1: The first ACM SIGPLAN conference on History of programming languages, 1978, pp. 245-272.

Page 9 [GA] Fleshed out object as a general agent

D. Lenat, “Beings,” MS Thesis, Stanford, 1975.

Page 9 [Ro] Sideways Composition: Aspects, Traits, Roles

I. Goldstein and D. Bobrow, “Descriptions for a Programming Environment,” in Proc. of the 1st Annual Conference of the National Association for Artificial Intelligence (AAAI), 1980.

I. Goldstein and D. Bobrow, “Extending object oriented programming in Smalltalk,” in Proc. of the 1980 ACM Conference on LISP and Functional Programming, 1980, pp. 75-81.

B. Pernici, Objects with roles, ACM SIGOIS Bulletin , Proceedings of the ACM SIGOIS and IEEE CS TC-OA conference on Office information systems, Volume 11 Issue 2-3, March 1990.

Y. Coady and G. Kiczales, “Back to the future: a retroactive study of aspect evolution in operating system code,” in Proc. of the 2nd International conference on Aspect-oriented software development, 2003, pp. 50 – 59.

N. Schärli, “Traits — Composing Classes from Behavioral Building Blocks,” Ph.D. thesis, University of Berne, 2005. Available at HTTP: http://www.iam.unibe.ch/~scg/Archive/PhD/schaerli-phd.pdf

Page 9 [LB] Late Binding of Everything

J. McCarthy, “Recursive Functions of Symbolic Expressions and their Computation by Machine,” in Comm. of the ACM, vol. 3, no. 3, pp. 184 – 195, 1960.

J. McCarthy, LISP 1.5 Programmer’s Manual, Cambridge, MA: MIT Press, 1962.

W. Teitelman, “PILOT: A step towards Man-Computer Symbiosis,” Technical Report: TR-32, MIT, 1966.

E. T. Irons, “Experience with an extensible language,” in Comm. of the ACM, vol.13 no.1, pp.31-40, 1970.

D. Ingalls, “The Smalltalk-76 Programming System Design and Implementation,” in Proc. of the 5th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages, 1978, pp. 9-16.

B. C. Smith, “Reflection and Semantics in a Procedural Language,” Ph. D. thesis, MIT/LCS/TR-272, 1982.

Page 9 [Spec] Specifications that are runable and debuggable

W. R. Swartout and R. Balzer, “On the Inevitable Intertwining of Specification and Implementation,” Comm. of the ACM, vol. 25, Jul., pp. 438-440, 1982.

P. Zave, “Operational specification languages,” in Proc. of the 1983 annual conference on Computers : Extending the human resource, 1983, pp. 214 – 222.

Kestrel Development Corporation, Specware 4.1 Language Manual, 2001-2004. [2006 Jul 5], Available at: http://www.specware.org/documentation/4.1/language-manual/SpecwareLanguageManual.html

Page 9 [SepM] Separation of Meanings from Optimizations

I. E. Sutherland, “Sketchpad: A man-machine communication system.,” in Proc. AFIPS 1963 Spring Joint Computer Conf., vol. 23, , New York: Spartan Books, 1963.

A. Borning, “ThingLab – an Object-Oriented System for Building Simulations Using Constraints,” in Proc. of the Fifth International Joint Conference on Artificial Intelligence, 1977, pp. 497-498.

B. Freeman-Benson and A. Borning, “The Design and Implementation of Kaleidoscope’90: A Constraint Imperative Programming Language.” in Proc. of the IEEE Computer Society 1992 International Conference on Computer Languages, 1992, pp. 174-180.

D. B. Lenat, "Cyc: A Large-Scale Investment in Knowledge Infrastructure," Comm. of the ACM, vol. 38, Nov., pp. 33-38, 1995.

V. Sikka, “Integrating Specialized Procedures into Proof Systems,” Ph. D. thesis, Stanford, 1996.

Page 9 [MM] Meaning what we mean, and holding on to it.

I. E. Sutherland, "Sketchpad: A man-machine communication system.," in Proc. AFIPS 1963 Spring Joint Computer Conf., vol. 23, , New York: Spartan Books, 1963.

D. E. Knuth, The TeXbook (Computers and Typesetting, Volume B), Reading, Mass.: Addison-Wesley, 1986.

Inference Corporation, ART Reference Manual, Los Angeles: Inference Corporation, 1987.

K. Kahn, E. Tribble, M. Miller, and D. Bobrow. Vulcan: Logical concurrent objects. In B. Shriver and P. Wegner, editors, *Research Directions in Object-Oriented Programming*, pages 75--112. The MIT Press, 1987.

Bjorn Freeman-Benson and Alan Borning, "The Design and Implementation of Kaleidoscope'90, A Constraint Imperative Programming Language", *Proceedings of the IEEE Computer Society 1992 International Conference on Computer Languages*, April 1992, pages 174-180.

K. Marriott and P. J. Stuckey, Programming with Constraints: An Introduction, Cambridge, MA: MIT Press, 1998.

R. Fourer, D. M. Gay, B. Kernighan, AMPL, Pacific Grove, CA: Brooks/Cole, 2003.

K. Apt, Principles of Constraint Programming, Cambridge: Cambridge University Press, 2003.

Page 9 [OT] Observe and Transform

K. E. Iverson, A Programming Language, New York: John Wiley & Sons, 1962.

D. Bricklin, Early ideas for a spreadsheet application, (private communication), late 70s.

A. Kay, "Computer Software," Scientific American, Sept., pp. 41-47, 1984.

J. Hughes. Why Functional Programming Matters. *Computer Journal*, 32(2), 1989.

A. Gupta, C. Forgy, A. Newell, High-speed implementations of rule-based systems , ACM Transactions on Computer Systems (TOCS), Volume 7 Issue 2, May 1989

D. Gelernter, "Generative communications in Linda," ACM Transactions on Programming Languages and Systems, vol. 7, Jan. pp. 80-112, 1985.

D. Gelernter, "Coordination Languages and their Significance", with Nicholas Carriero, *Communications of the ACM*, 35 (2), February 1992, pp. 97-107

Page 9 [MP] Massive Parallelism: "Particles and Fields", etc.

I. E. Sutherland, "Sketchpad: A man-machine communication system.," in Proc. AFIPS 1963 Spring Joint Computer Conf., vol. 23, , New York: Spartan Books, 1963.

D. E. Knuth, The TeXbook (Computers and Typesetting, Volume B), Reading, Mass.: Addison-Wesley, 1986.

M. Resnick, Turtles, Termites, and Traffic Jams: Explorations in Massively Parallel Microworlds (Complex Adaptive Systems), Cambridge, MA: MIT Press, 1987.

Y. Yokote, F. Teraoka, M. Yamada, H. Tezuka and M. Tokoro, "The Design and Implementation of the Muse Object-Oriented Distributed Operating System,", in Proc. of First Conference on Technology of Object-Oriented Languages and Systems, 1989.

A. A. Chien, "Concurrent Aggregates(CA): An Object-Oriented Language for Fine-Grained Message-Passing Machines," Ph.D. thesis, MIT, 1990.

Y. Yokote, A. Mitsuzawa, N. Fujinami and M. Tokoro, "Reflective Object Management in the Muse Operating System", in Proc. IEEE Int'l Workshop on Object Orientation in Operating systems, 1991, pp. 16 - 23.

Y. Yokote, F. Teraoka, A. Mitsuzawa, N. Fujinami and M. Tokoro, "The Muse Object Architecture: A New Operating System Structuring Concept," in Operating Systems Review, vol. 25, no.2, 1991.

R. D. Cook, Finite Element Modeling for Stress, John Wiley & Sons, 1995

U. Wilensky, NetLogo, Center for Connected Learning and Computer-Based Modeling. Northwestern University, Evanston, IL, (1999). Available at: http://ccl.northwestern.edu/netlogo

V. Grimm, "Ten years of individual-based modeling in ecology: what have we learned and what could we learn in the future?," Ecological Modeling, vol. 115, pp. 129-148, 1999.

V. Grimm and S. F. Railsback, Individual-based modeling and ecology, New Jersey: Princeton University Press, 2005.

Y. Ohshima, "A GUI-based Interactive Massively Parallel Particle Programming System," IEEE Visual Language and Human Centric Computing 2005, 2005, pp. 91 – 98.

Page 10 [EMT] Explicit Models of Time

D.W. Barron, J.N. Buxton, D.F. Hartley, E. Nixon, and C. Strachey, "The main features of CPL," The Computer Journal, vol. 6, no. 2, pp.134-143, 1963.

O.- J. Dahl and K. Nygaard, "SIMULA: an ALGOL-based simulation language," Comm. of the ACM, vol. 9, Sept., pp. 671-678, 1966.

J. McCarthy and P. Hayes, "Some Philosophical Problems from the Standpoint of Artificial Intelligence," B. Meltzer and D. Michie, Eds., in Machine Intelligence 4, Edinburgh: Edinburgh University Press, 1969, pp. 463—502.

R. M. Balzer, "Exdams: Extendible debugging and monitoring system," in AFIPS Proc., Spring Joint Computer Conference, vol. 34, 1969, pp. 567 – 580.

E. Ashcroft and W. W. Wadge, "Lucid, a nonprocedural language with iteration", Comm. of the ACM, vol. 20, July, pp. 519-526, 1977.

C. Hewitt, "Control structures as patterns of passing messages," Artificial Intelligence, vol. 8, no. 3, pp. 323 – 364, 1977.

D. P. Reed, "Naming and Synchronization in a Decentralized Computer System," Ph.D. thesis, MIT, 1978.

L. Lamport, "Time, Clocks and the Ordering of Events in a Distributed System", in Comm. of the ACM, vol. 21, no. 7, pp. 558-565, 1978.

W. W. Wadge and E. Ashcroft, Lucid, the Dataflow Programming Language, New York: Academic Press, 1986.

D. Jefferson, B. Beckman, F. Wieland, L. Blume, M. Diloreto, "Time warp operating system," in Proc. of the 11th ACM Symposium on Operating system principles, 1987, pp. 77-93.

L. Lamport, "The Temporal Logic of Actions", ACM Transactions on Programming Languages and Systems 16, vol. 3, 872-923, 1994.

Page 10 [Cr 1] Croquet papers

D. A. Smith, A. Kay, A. Raab, and D. P. Reed, "Croquet – A Collaboration System Architecture," [Online paper], 2003, [2006 Jun 19], Available at HTTP: http://www.opencroquet.org/about_croquet/papers.html

D. A. Smith, A. Raab, D. P. Reed and A. Kay, "Croquet: A menagerie of New User Interfaces," in Proc. of the Second International Conf. on Creating, Connecting and Collaborating through Computing, 2004, pp. 4-11.

D. Reed, "Designing coquet's TeaTime: a real-time, temporal environment for active object cooperation," in Conf. on Object Oriented Programming Systems Languages and Applications, 2005, p. 7.

D. A. Smith, A. Raab, D. P. Reed and A. Kay, "A Hedgehog Architecture," [PowerPoint presentation], 2005, [2006 Jun 19], Available at HTTP: http://www.opencroquet.org/about_croquet/papers.html

D. P. Reed, "TeaTime: Designing the Architectural Framework for Croquet," [PowerPoint presentation], Oct. 2005, [2006 Jun 19], Available at HTTP: http://www.opencroquet.org/about_croquet/papers.html

Also see the open source website for Croquet: http://www.opencroquet.org

Page 10 [MOP] Meta-Object Protocols

J. McCarthy, “Recursive Functions of Symbolic Expressions and their Computation by Machine,” in Comm. of the ACM, vol. 3, no. 3, pp. 184 – 195, 1960.

D. Ingalls, “The Smalltalk-76 Programming System Design and Implementation,” in Proc. of the 5th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages, 1978, pp. 9-16.

B. C. Smith, “Reflection and Semantics in a Procedural Language,” Ph. D. thesis, MIT/LCS/TR-272, 1982.

G. Kiczales, D. Bobrow, J. Riviera, The Art of the Metaobject Protocol, Cambridge, MA: MIT Press, 1991.

Page 10 [FNB] “From Nothing” Bootstrapping

E. T. Irons, “Experience with an Extensible Language,” in Comm. of the ACM, vol. 13, no. 1, pp. 31-40, 1970.

L. Tesler, H. Enea and D.C. Smith, “The Lisp-70 pattern matching system,” J. Nilsson, Ed., in Proc. of the 3rd International Joint Conference on Artificial Intelligence, 1973, pp. 671-676.

D. Ingalls, “The Smalltalk-76 Programming System Design and Implementation,” in Proc. of the 5th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages, 1978, pp. 9-16.

G. Krasner (ed), Smalltalk-80: Bits of History, Words of Advice (Addison-Wesley series in computer science) (Paperback) 1983

P. Deutsch, “The past, present, and future of Smalltalk,” in Proc. of the 3rd European Conference on Object Oriented Programming, Cambridge University Press, 1989, pp. 73-87.

A. Kay, “The Early History of Smalltalk”, History of Programming Languages II, T. Bergin (ed.), Addison Wesley, 1996.

D. Ingalls, T. Kaehler, J. Maloney, S. Wallace and A. Kay, “Back to the future: the story of Squeak, a practical Smalltalk written in itself,” in OOPSLA ’97: Proc. of the 12th ACM SIGPLAN Conference on Object-oriented Programming, 1997, pp. 318-326.

J. Aycock, “A brief history of just-in-time”, in ACM Computing Surveys, vol. 35, no. 2, pp. 97 – 113, 2003.

I. Piumarta, “The Virtual Processor: Fast, Architecture-Neutral Dynamic Code Generation,” in Proc. of the 3rd Virtual Machine Research and Technology Symposium, 2004. Available at HTTP: http://www.piumarta.com/papers/vpu-vm04.pdf

I. Piumarta, “Accessible Language-Based Environments of Recursive Theories”[Internal Whitepaper], Glendale, Calif.: Viewpoints Research Institute, Inc., Sept., 2005. Available at HTTP: http://piumarta.com/papers/albert.pdf

Page 12 [Mc] How Lisp moved from “math” to a “programmable math”

J. McCarthy, “Recursive Functions of Symbolic Expressions and their Computation by Machine,” in Comm. of the ACM, vol. 3, no. 3, pp. 184 – 195, 1960.

J. McCarthy, LISP 1.5 Programmer’s Manual, Cambridge, MA: MIT Press, 1962.

J. McCarthy, “History of LISP,” in HOPL-1: The first ACM SIGPLAN conference on History of programming languages, 1978, pp. 217-223.

Page 12 [Ka] How Smalltalk Was Originally Implemented

A. Goldberg and A. Kay, “Smalltalk-72 Instruction Manual,” Technical Report SSL 76-6, Learning Research Group, Xerox Palo Alto Research Center, 1976.

A. Kay, “The early history of Smalltalk,” in HOPL-II: The Second ACM SIGPLAN Conference on History of Programming Languages, 1993, pp. 69-95.

Page 13 [VP] Open Source Systems by Viewpoints Research Institute

Websites: www.squeak.org, www.squeakland.org, www.opencroquet.org

Page 15 [Dis] Dissemination and Outreach

Initial Experiences of ALAN-K: An Advanced LeArning Network in Kyoto , Shin'ichi Konomi, Kyoto University; Kyoto Software Applications, Inc., Hiroki Karuno, Kyoto University, First Conference on Creating, Connecting and Collaborating through Computing, IEEE, p. 96. Available at: http://csdl2.computer.org/persagen/DLAbsToc.jsp?resourcePath=/dl/proceedings/&toc=comp/proceedings/c5/2003/1975/00/1975toc.xml&DOI=10.1109/C5.2003.1222340

Squeak in Spain as Part of the LinEx Project. Diego Gómez Deck, José L. Redrejo Rodriguez. Journal Title: Conference On Computing Frontiers, 2004. Available at: http://smallwiki.unibe.ch/school**squeak**ers/smalllandpaper/?action=MimeView

Una experiencia interdisciplinar en educación primaria mediante el uso de Squeak, (An experience with interdisciplinary primary education using Squeak), Adriana Gewerc Barujel, Fernando Fraga Varela, *Innovación educativa, ISSN 1130-8656,* Nº 15, 2005, pags. 223-233.