Programming as an Experience: The Inspiration for Self

Randall B. Smith and David Ungar

Sun Microsystems Laboratories
2550 Casey, Ave. MS MTV29-116
Mountain View, CA 94043, USA

randall.smith@sun.com david.ungar@sun.com

Abstract. The Self system attempts to integrate intellectual and non-intellectual aspects of programming to create an overall experience. The language semantics, user interface, and implementation each help create this integrated experience. The language semantics embed the programmer in a uniform world of simple objects that can be modified without appealing to definitions of abstractions. In a similar way, the graphical interface puts the user into a uniform world of tangible objects that can be directly manipulated and changed without switching modes. The implementation strives to support the world-of-objects illusion by minimizing perceptible pauses and by providing true source-level semantics without sacrificing performance. As a side benefit, it encourages factoring. Although we see areas that fall short of the vision, on the whole, the language, interface, and implementation conspire so that the Self programmer lives and acts in a consistent and malleable world of objects.

1 Introduction

During the last decade, over a dozen papers published about Self have described the semantics, the implementation, and the user interface. But they have not completely articulated an important part of the work: our shared vision of what programming should be. This vision focuses on the overall experience of working with a programming system, and is perhaps as much a feeling thing as it is an intellectual thing. This paper gives us a chance to talk about this underlying inspiration, to review the project, and to make a few self-reflective comments about what we might have done differently. Although the authors have the luxury of commenting from an overview perspective, the reader should keep in mind that the this work is the result of efforts by the many individuals who have worked in the Self project over the years.

1.1 Motivation

Programmers are human beings, embedded in a world of sensory experience, acting and responding to more than just rational thought. Of course to be effective, programmers need logical language semantics, but they also need things like confidence, comfort, and satisfaction — aspects of experience which are beyond the domain of pure logic. These

concerns have traditionally been addressed separately by putting the logic in the language and providing for the rest of experience with the programming environment. The Self system attempts to integrate the intellectual and experiential sides of programming.

In our vision, the Self programmer lives and acts in a consistent and malleable world, from the concrete motor-sensory, to the abstract, intellectual levels. At the lowest, motor-sensory level of experience, objects provide the foundation for natural interaction. Consequently, every visual element in Self, from the largest window to the smallest triangle is a directly manipulable object. At the higher, semantic levels of the language, there are many possible computational models: in order to harmonize with the sensory level, Self models computation exclusively in terms of objects. Thus, every piece of Self data, from the largest file to the smallest number is a directly manipulable object. And, in order to ensure that these objects could be directly experienced and manipulated, we devised a model based on "prototypes." Just as a button can be added to any graphical object, so can a method be added to any individual object in the language, without needing to refer to a class. This prototype model and the use of objects for everything requires a radically new kind of implementation. In Self, implementation, interface, and language were designed to work together to create a unified programming experience.

In the following sections we in turn review the language semantics, the user interface, and the implementation. In each section we will try to point out where we think we succeeded and where we think we failed in being true to the vision.

2 Language Semantics

Self was initially designed by the authors at Xerox PARC [US87]. We employed a minimalist strategy, striving to distill an essence of object and message. Self has evolved over the years in design and implementation at Stanford University and most recently at Sun Microsystems Laboratories. A user interface and programming environment built in Self are part of the system: Self today is a fairly large system, and includes a complete programming environment and user interface framework.

A computation in Self consists solely of objects which in turn consist of slots. A slot has a name and a value. Slot names are always strings, but slot values can be any Self object. A slot can be marked with an asterisk to show that it designates a *parent*. Figure 1 illustrates a Self object representing a two-dimensional point with **x** and **y** slots, a parent slot called **myParent**, and two special assignment slots, **x:** and **y:,** that are used to assign to the **x** and **y** slots. The object's parent has a single slot called **print** (containing a method object).

When sending a message, if no slot name matches within the receiving object, its parent's slots are searched, and then slots in the parent's parent, and so on. Thus our point object can respond to the messages **x**, **y**, **x:, y:,** and **myParent**, plus the message **print**, because it *inherits* the **print** slot from its parent. In Self, any object can potentially be a

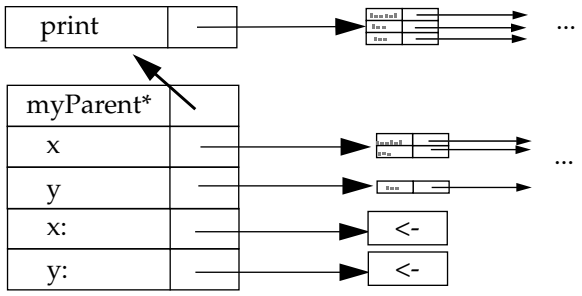

Figure 1. A Self point has **x** and **y** slots, with **x:** and **y:** slots containing the assignment primitive for changing **x** and **y**. The slot **myParent** carries a "parent" denotation (shown as an asterisk). Parent slots are an inheritance link, indicating how message lookup continues beyond the object's slots. For example, this point object will respond to a print message, because it inherits a print slot from the parent.

parent for any number of children, or it can be a child of any object. This uniform ability of any object to participate in any role of inheritance contributes to the consistency and malleability of Self and, we hope, contributes to the programmer's comfort, confidence, and satisfaction.

In addition to slots, a Self object can include code. Such objects are called *methods*; since they do what methods in other languages do. For example, the object in the **print** slot above includes code and thus serves as a method. However in Self, any object can be regarded as a method; a "data" object contains code that merely returns itself. This viewpoint serves to unify computation with data access: when an object is found in a slot as a result of a message send it is *run*; data returns itself, while a method invokes its code. Thus when the **print** message is sent to our point object, the code in the print slot's method will run immediately. This unification reinforces the interpretation that the experience of the client matters, not the inner details of the object used. For example, some soda machines dispense pre-mixed soda, while others dynamically mix syrup and carbonated water on demand. The user does not care to become involved in the distinction, what matters is only the end product, be it soda or the result of a computation.

Of course Self is not the only language that unifies access and computation. For example Beta [MMN93] does too. From a traditional computer science viewpoint, this unification serves to provide data abstraction; it is impossible to tell, even from within an abstraction, whether some value is stored or computed. However when designing Self we also sought to unify assignment and computation; this unification is slightly more unusual. Assignment in Self is performed with assignment slots, such as **x:** and **y:,** which contain a special method (symbolized by the arrow), that takes an argument (in addition to the receiver) and stuffs it in either the **x** or **y** slot. This desire for access/assignment symmetry can be interpreted as arising from the sensory-motor level of expe-

rience. From the time we are children, experience and manipulation are inextricably intertwined; we best experience an object when we can touch it, pick it up, turn it over, push its buttons, or even taste it. We believe that the notion of a container is a fundamental intuition that humans share and that by unifying assignment and computation in the same way as access and computation, Self allows abstraction over containerhood; since all containers are inspected or filled by sending messages; any object may pretend to be a container while employing a different implementation.

A contrast between Self and Beta may illustrate the role that our concern for a particular kind of programming experience played in the language design. In Beta, data takes two forms: values and references. Beta unifies accessing values with computation and also unifies assignment of values with computation, but uses a different syntax for accessing or changing references. For example, a data attribute containing a reference to a point would be accessed by saying pt[]-> but a method in the receiver returning a reference to a point would be run by pt->. Giving the programmer two forms of data can be seen as giving the programmer more tools with which to work. More tools can of course be a good thing, but since in our opinion, the value/reference distinction does not really exist at the sensory-motor level of experience, we chose a slightly less elaborate scheme with a single kind of data structuring mechanism. We also believe there are advantages to uniformity in and of itself.

The design of slots in Self has proven to be challenging because of the unified access to state and behavior. In Self the same message can be sent to either read a data slot or run a method in the slot. Where should the distinction be maintained? We have taken the position that method objects are fundamentally different, in that they run non-trivial code whereas data objects just return themselves. Still, this behavior of method objects means that they cannot be directly manipulated because they would run. So we have had to invent primitive objects called *mirrors* as a way of indirectly mentioning methods. Mirrors can be better justified as encapsulators of reflective operations and as relieving each object of the burden of inheriting such behavior. But mirrors can hamper uniformity; it is sometimes unclear whether a method should take a mirror as argument or the object itself. Another approach to unifying invocation and access would be to add a bit to a slot that would record whether or not the slot was a method or data slot. This approach has its own problems: what would it mean to put 17 into a method slot? In our opinion, this issue is not fully resolved.

The treatment of assignment slots in Self is also a bit troublesome. A Self assignment slot is a slot containing a special primitive (the same one for all assignment slots), which uses the *name of the slot* (very odd) to find a corresponding data slot *in the same object* (also odd). This treatment leads to all sorts of special rules, for instance it is illegal to have an object contain an assignment slot without a corresponding data slot, so the code that removes slots is riddled with extra checks. Also, this treatment fails to capture the intuitive notion of a container. Other prototype-based languages address this issue by having a slot-pair be an entity in the language and casting an assignable slot as such a

slot pair. Another alternative might be to make the assignment object have a slot identifying its target, so that in principle any slot could assign to any other.

Messages can have extra arguments in addition to the receiver. For example, **3 + 4** sends the message **+** to the object **3,** with **4** as argument. In contrast to many other object-oriented languages numbers are objects in Self, just as in Smalltalk. Because languages like Smalltalk or Self do arithmetic by sending messages, programmers are free to add new numeric data types to the language, and the new types can inherit and reuse all the existing numeric code. Adding complex numbers or matrices to the system is straightforward: after defining a **+** slot for matrices, the user could have the matrix freely inherit from some slot with code that sends **+**. Code that sends **+** would then work for matrices as well as for integers. Self's juxtaposition of a simple and uniform language (objects for numbers and messages for arithmetic in this case) with a sophisticated implementation permits the programmer to inhabit a more consistent and malleable computational universe.

There is more to be said about the language: new Self objects are made simply by copying—there are no special class objects for instantiation (see the later section entitled Prototypes and Classes). The current implementation allows multiple inheritance, which requires a strategy for dealing with multiple parents. It has block closure objects, and threads. (A few constructs that would be somewhat obscure in a language like Self, such as methods contained in the local slots of methods, are not yet supported by the implementation.)

2.1 Discussion

Our desire to provide a certain kind of programming experience has colored Self's stance on some traditional issues:

Type Declarations. In order to understand the design of the Self language it helps to examine the assumptions that underlie language design. In the beginning, there were FORTRAN, ALGOL and Lisp. In all three of these languages the programmer only has to say what is necessary to execute programs. Since Lisp was interpreted, no type information was supplied at all. Since ALGOL and FORTRAN were compiled, it was necessary for programmers to specify primitive type information, such as whether a variable contained an integer or a float, in order for the compiler to generate the correct instructions. As compiled languages evolved, it was discovered that by adding more static declarations, the compiler could sometimes create more efficient code. For example, in PL/I procedures had to be explicitly declared to be recursive, so that the compiler could use a faster procedure prologue for the non-recursive ones.

Programmers noticed that this static declarative information could be of great value in making a program more understandable. Until then, the main benefit of declarations had

been to the compiler, but with Simula[1] and PASCAL a movement was born; using declarations both to benefit human readers and compilers.

In our opinion, this trend has been a mixed blessing, especially where object-oriented languages are concerned. The problem is that the information a human needs to understand a program, or to reason about its correctness, is not necessarily the same information that a compiler needs to make a program run efficiently. But most languages with declarations confuse these two issues, either limiting the efficiency gained from declarations, or, more frequently hindering code reuse to such an extent that algorithms get duplicated and type systems subverted.

Self therefore distinguishes between concrete and abstract types. Concrete types (embodied by *maps*) are completely hidden from the Self programmer. Maps are only visible to the implementation, where they are used as an efficiency mechanism. Abstract types on the other hand are notions that the programmer might think about. Self has no particular type manifestation in the language: declarative information is left to the environment. For example, one language level notion of abstract type, the *clone family*, is used in the work of Agesen et. al. [APS93] in their Self type inference work. There is no clone family object in the Self language, but such objects can be created and used by the programming environment. In order to structure complexity and provide the freest environment possible, we have layered the design so that the Self language proper includes only information needed to execute the program, leaving declarative information to the environment. This design keeps the language small, simplifies the pedagogy, and allows users to potentially extend the intensional domain of discourse.

Minimalism. Why have we tried to keep the Self language minimal? It is always tempting to add a new feature that handles some example better. Although the feature had made it possible to directly handle some examples, the burden it imposed in all reasoning about programs was just too much. We abandoned it for Self 3.0. Although adding features seems good, every new concept burdens every programmer who comes into contact with the language.

We have learned the hard way that smaller is better and that examples can be deceptive. Early in the evolution of Self we made three mistakes: prioritized multiple inheritance, the sender-path tie-breaker rule, and method-holder-based privacy semantics.[2] Each was motivated by a compelling example [CUCH91]. We prioritized multiple parent slots in order to support a mix-in style of programming. The sender-path tie-breaker rule allows two disjoint objects to be used as parents, for example a rectangle parent and a tree node parent for a VLSI cell object. The method-holder-based privacy semantics allowed objects with the same parents to be part of the same encapsulation domain, thereby supporting binary operations in a way that Smalltalk cannot [CUCH91].

---

[1] Simula™ is a trademark of a.s. Simula

[2] In all fairness, the first author was across the Atlantic at the time and had nothing to do with it. On the other hand, if he had not wandered off maybe these mistakes could have been avoided.

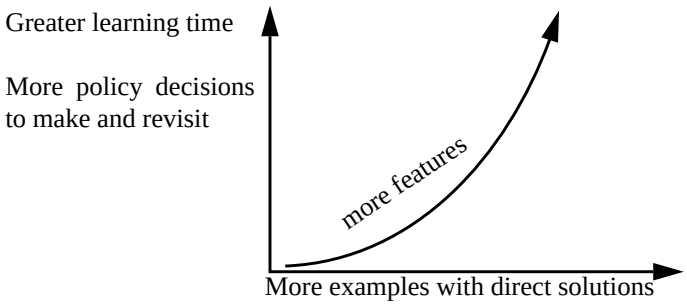

Figure 2. As more features are embedded in the language, the programmer gets to do more things immediately. But complexity grows with each feature: how the fundamental language elements interact with each other must be defined, so complexity growth can be combinatorial. Such complexity makes the basic language harder to learn, and can make it harder to use by forcing the programmer to make a choice among implementation options, a choice which may have to be revisited later.

But each feature also caused us no end of confusion. The prioritization of multiple parents implied that Self's "resend" (call-next-method) lookup had to be prepared to back-up down parent links in order to follow lower-priority paths. The resultant semantics took five pages to write down, but we persevered. After a year's experience with the features, we found that each of the members of the Self group had wasted no small amount of time chasing "compiler bugs" that were merely unforeseen consequences of these features. It became clear that the language had strayed from its original path.

We now believe that when features, rules, or elaborations are motivated by particular examples, it is a good bet that their addition will be a mistake. The second author once coined the term "architect's trap" for something similar in the field of computer architecture; this phenomenon might be called "the language designer's trap."

If examples cannot be trusted, what do we think should motivate the language designer? Consistency and malleability. When there is only one way of doing things, it is easier to modify and reuse code. When code is reused, programs are easier to change and most importantly, shrink. When a program shrinks its construction and maintenance requires fewer people which allows for more opportunities for reuse to be found. Consistency leads to reuse, reuse leads to conciseness, conciseness leads to understanding. That is why we feel that it is hard to justify any type system that impedes reusability; the resultant duplication leads to a bigger program that is then harder to understand and to get right. Such type systems can be self-defeating.[1]

[1] However, by emphasizing the ability to express intuitive relationships, such as covariant specialization, over the ability to do all checking statically, it is possible to do a better job. See [MMM90].

Prototypes and Classes. There are now several fairly mature object-oriented languages based on prototypes. (For overviews see [Blas94], [DMC92], and [SLS94].) These languages differ somewhat in their treatment of semantic issues like privacy, copying, and the role of inheritance. (One notable system, Kevo [Tai93a], [Tai92], [Tai93] does not have delegation or inheritance at all.) All these languages have a model in which an object is in a important sense self-contained. Prototypes are often presented as an alternative to class-based language designs, so the subject of prototypes vs. classes can serve as point of (usually good natured) debate.

However, depending on how one defines "class," one may or may not think that classes are already present in a prototype based system. Some (e.g. [Blas94]) see a Self prototype as playing the role of class, since it determines the structure of its copies. Others note that much of the current Self system is organized in a particular way, using what we call "traits" objects in many places to provide common state and behavior for sharing among children. Such sharing is reminiscent of that provided by a class. However, classes normally also provide the description of an instance's implementation, and a "new" method for instantiation, neither of which are found in a traits object.

In a class-based system, any change (such as a new instance variable) to a class will affect new instances of a subclass. In Self, a change to a prototype (such as a new slot) will not affect anything other than the prototype itself (and its subsequent direct copies).[1] So we have implemented a "copy-down" mechanism in the environment to share implementation information. It allows the programmer to add and remove slots to an entire hierarchy of prototypes in a single operation. Functionality that is provided at the language level in class-based systems has risen to the programming environment level in Self. In general, the simple object model in Self means that some functionality omitted from the language may go back into the environment. Because the environment is built out of Self objects, the copy-down policy can be changed by the programmer. But such flexibility comes with a price. Now, there are two interfaces for adding slots to objects, the simple language level and the copying-down Self-object level. This loss of uniformity could be a source of confusion when writing a program that needs to add slots to objects. Only time will tell if the flexibility is worth the complication.

A brief examination of the emulation of classes in Self will serve to illuminate both the nature of a prototype-based object model and the trade-off between implementing concepts in the language versus in the environment. In order to make a Self shared parent look more like a class, one could create a "new" method in the shared parent. This method could make a copy of some internal reference to a prototype, and so would appear to be an instantiation device. Figure 3 suggests how one might start to make a Smalltalk class out of Self objects. Mario Wolczko has built a more complete implementation of this, and has shown [Wol95] that it works quite well: he can read in Smalltalk source code and execute it as a Self program. There are certain restrictions on the Smalltalk

[1] Self prototypes are not really special objects, they are distinguished only by the fact that, by convention, they are copied. Any copy of the prototype would serve as a prototype equally well. Some other prototype-based systems take a different approach.

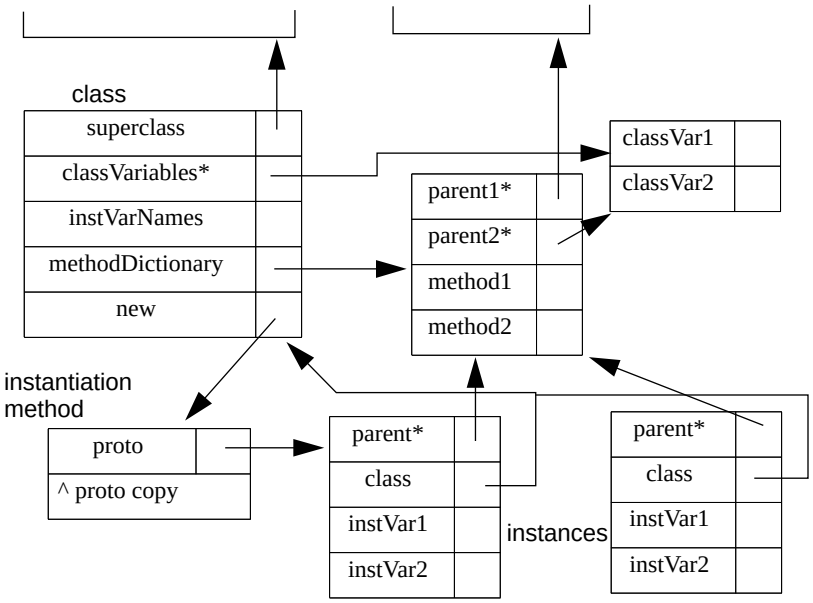

Figure 3. This figure suggests how Self objects might be composed to form Smalltalk-like class structures as demonstrated more completely by Wolczko [Wol95]]. He shows that, with some caveats, Smalltalk code can be read into a Self system, parsed into Self objects, then executed with significant performance benefits, thanks to the Self's dynamically optimizing virtual machine.

source, but thanks to Self's implementation technology, once the code adaptively optimizes, the Self version of Smalltalk code will generally run faster than the Smalltalk version.

Do prototype-based systems like Self have classes? The answer would seem to be that if your definition of class is not satisfied by the existing language elements, you can probably build something quite easily that would make you happy. (It would be difficult to do this kind of trick in a prototype-based language that did not have an inheritance mechanism.) Of course, almost any language can emulate any other, but in this case the classes are built directly out of the prototype-based objects so directly that the classes constructed out of Self objects run faster that those built-in to Smalltalk. General meta-object issues in prototype-based languages are tackled by the Moostrap system [Mul95].

The use of traits might be seen as a carryover from the Self group's Smalltalk experience. Interestingly, it is likely that our old habits might not have done Self justice (as observed in [DMC92].) There are many other ways to organize Self objects other than

by prototypes inheriting from a chain of traits parents, and many of these ways avoid a problem with the traits organization: a traits object appears to be an object but in fact cannot respond to most of its messages. For example the point traits object lacks x and y slots and so cannot respond to printString, since its printString slot contains a method intended to be shared by point objects. We probably would have done better to put more effort into exploring other organizations. When investigating a new language, your old habits can lead you astray.

3 The User Interface and Programming Environment

Self is an unusually pure object-oriented language, because it uses objects and messages to achieve the effects of flow control, variable scoping, and primitive data types. This maniacal devotion to the object-message paradigm is intended to match a devotion to a user interface based on concrete, direct manipulation of uniform graphical objects. By matching the language to the user interface, we hope to create an experience of programming that can be learned more easily, and can be performed with less cognitive overhead.

The notion of direct manipulation has been around for many years now, and it is interesting to note that some of the earlier prototype-based systems were visual programming environments [BD81], [Smi87]. The current Self user interface [SMU95], [MS95] enhances the sense of direct manipulation by employing two principles we call *structural reification*, and *live editing*. We will define these principles, and show with an example how the interface brings the feeling of direct object experience to the task of creating objects and assembling applications.

3.1 Structural reification

The fundamental kind of display object in Self is called a “morph,” a term borrowed from Greek meaning essentially “thing.” Self provides a hierarchy of morphs. The root of the hierarchy is embodied in the prototypical morph, which appears as a kind of golden colored rectangle. Other object systems might choose to make the root of the graphical hierarchy an abstract class with no instances. But prototype systems usually provide generic examples of abstractions. This is an important part of the structural reification principle: there are no invisible display objects. Any descendant of the root morph (or any other morph for that matter) is guaranteed to be a fully functional graphical entity. It will inherit methods for displaying and responding to input events that enable it to be directly manipulated.

In keeping with the principle of structural reification, any morph can have “submorphs” attached to it. A submorphs acts as though it is glued to the surface of its hosting morph. Composite graphical structure typical of direct manipulation interfaces arises through the morph-submorph hierarchy. Again, many systems provide compositing with special “group” objects which are normally invisible. But because we want things to feel very

solid and direct, we chose to follow a simple metaphor of sticking morphs together as though glued to each other.

A final part of structural reification arises from the approach to submorph layout. Graphical user interfaces often require that subparts be lined up in a column or row. Self's graphical elements are organized in space by "layout" objects that force their submorphs to line up as rows or columns. John Maloney [MS95] has shown how to create efficient "row and column morphs" as children of the generic morph. These objects are first class, tangible elements in the interface. They embody their layout policy as visible parts of the submorph hierarchy, so the user need only be able to access the submorphs in a structure in order to inspect or change the layout in some way. A possible price of this uniformity is paid by a user who does not wish to see the layout mechanism, but is confronted with it anyway.

An example in section 3.3 will illustrate how structural reification assures that any morph can be seen and can be grabbed, moved and inspected, and assures that graphical composition and layout constraints are physically present in the interface. Structural reification is an important part of making programming feel more tangible and direct.

3.2 Live Editing

Live editing simply means that at any time, an object can be directly changed by the user. Any interactive system that allows arbitrary runtime changes to its objects has a degree of support for live editing. But we believe Self provides an unusually direct interface to such live changes. The key to live editing is provided by Self's "meta menu," a menu that can pop up when the user holds the third mouse button while pointing to a morph. The meta menu contains items such as "resize," dismiss," and "change color" which allow the user to edit the object directly. There are also menu elements that enable the user to "embed" the morph into the submorph structure of a morph behind it, and menu elements that give access to the submorph hierarchy at any point on the screen.

The "outliner" menu item creates a language-level view of the morph under the mouse[1]. This view shows all of the slots in an object, and provides a full set of editing functionality. With an outliner you can add or remove slots, rename them, or edit their contents. Code for a method in a slot can be textually edited: outliners are important tools for programmers. Access to the outliner through the meta menu makes it possible to investigate the language-level object behind any graphical object on the screen.

The outliner supports the live editing principle by letting the user manipulate and edit slots, even while an object is "in use." The example below illustrates how a slot can be "carried" from one object to another, interactively modifying their language level structure.

---

[1] Lars Bak designed the outliner framework for Self.

Popping up the meta menu is the prototypical morph's response to the third mouse button click. All morphs inherit this behavior, even elements of the interface like outliners and pop-up menus themselves. But wait! Pop-up menus are impossible to click on: you find them under your mouse only when a mouse button is already down. To release the button in preparation for the third button click causes the pop-up to frustratingly disappear. Consequently, we provide a "pin down" button, which, when selected, causes the menu to become a normal, more permanent display object. The mechanism is not new, but providing it in Self enables the menu to be interactively pulled apart or otherwise modified by the user or programmer.

Live editing is partly a result of having an interactive system, but is enhanced by features in the user interface. This principle reinforces the feel that the programmer is working directly with concrete objects. The following example will clarify how this principle and the structural reification principle help give the programmer a feeling of a working in a uniform world of accessible, tangible objects.

3.3 Example of Direct Application Construction

Suppose the programmer (or motivated user) wishes to expand an ideal gas simulation, extending the functionality and adding user interface controls. The simulation starts simply as a box containing "atoms" which are bouncing around inside. Using the third mouse button, the user can invoke the meta menu, and select "outliner" to get the Self-level representation of the object (Figure 4). The outliner enables arbitrary language-level changes to the ideal gas simulation.

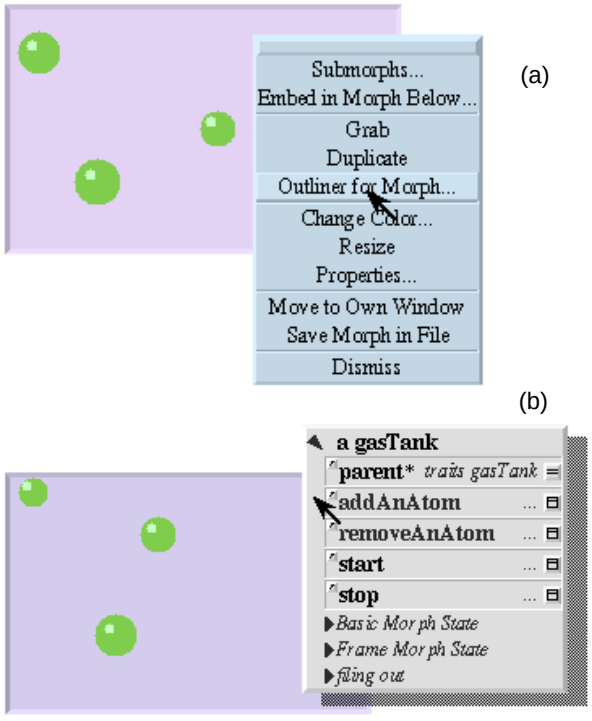

Figure 4. In a Self window, the user pops up the meta menu on the ideal gas simulation (a). Selecting "outliner" gives the Self-level representation to the user, which can be carried and placed as needed (b). (The italic items at the bottom of the outliner represent slot categories that may be expanded to view the slots. Unlike slots, categories have no language level semantics and are essentially a user interface convenience.)

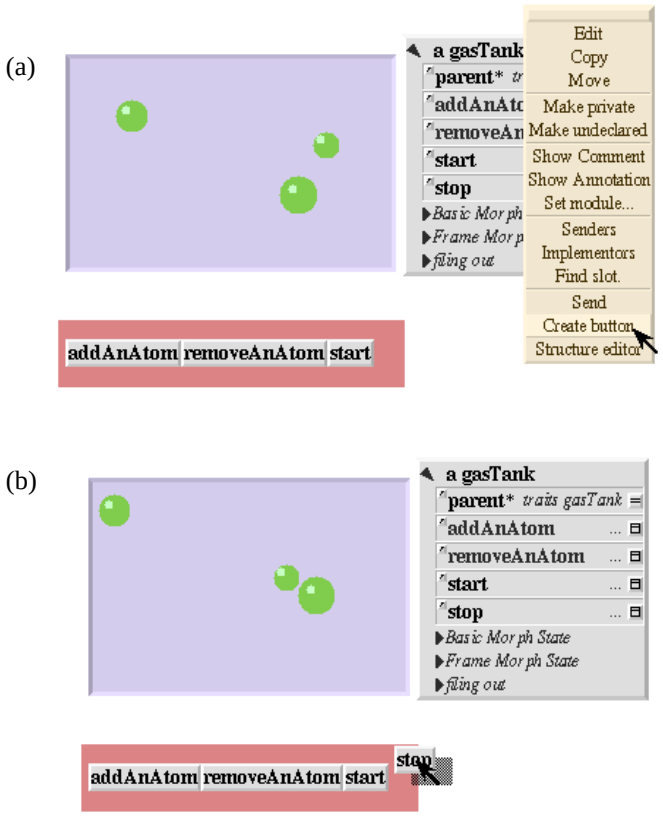

Figure 5. The middle mouse button pop up menu on the "stop" slot (a) enables the user to create a button for immediate use in the user interface (b). This button will be embedded in a row morph, so that it lines up horizontally.

With the outliner, the user can start to create some controls right away. In the outliner, there are slots labeled "start" and "stop." These slots can be converted into user interface buttons by selecting from the middle-mouse-button pop-up menu on the slot (Figure 5). Pressing these buttons starts and stops the bouncing atoms in the simulation. This is an example of live editing at work: in just a few gestures, the programmer has gone through the outliner to create interface elements while the simulation continues to run.

The programmer may wish to create several such buttons, and arrange them in a row. The programmer selects "row morph" from a palette: when the buttons are embedded into the row, they immediately snap into position. Once the row of buttons is created, the programmer wishes to align the row below the gas tank. Again, the programmer can create a column frame: when the gas tank and the button row are embedded into the column frame, they line up one below the next (Figure 6)

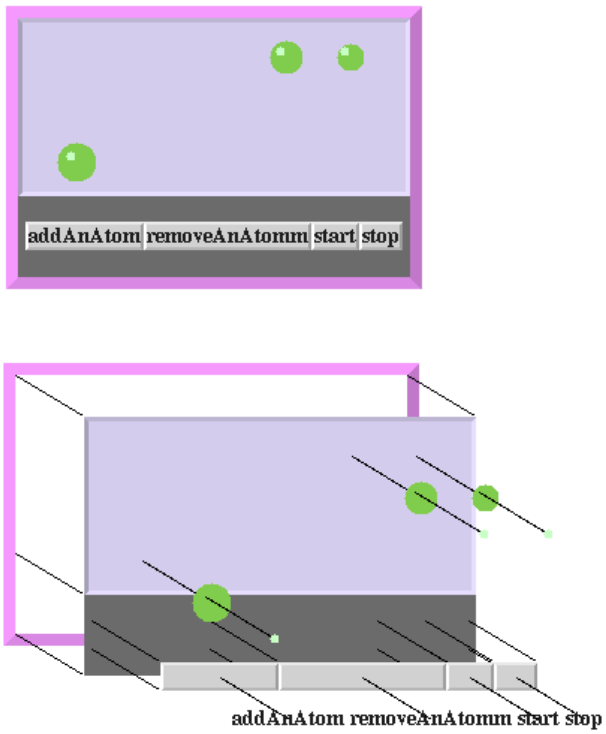

Figure 6. Composite graphical effects are achieved by embedding: any kind of morph can be embedded in any other kind of morph. The ideal gas simulation at left is a compound morph whose embedding relationships are shown at right.

Figure 6 also illustrates how composite graphical effects are achieved through the morph-submorph hierarchy. The interface employs morphs down to quite a low level. The labels on buttons, for example, are themselves first class morphs.

The uniformity of having “morphs all the way down” further reinforces the feel of working with concrete objects. For example, the user may wish to replace the textual label with an icon. The user can begin this task by pointing to the label and invoking the meta menu. There is a menu item labeled “submorphs” which allows the user to select which morph in the collection under the mouse he wishes to denote (see Figure 7). The user can remove the label directly from the button’s surface. In a similar way, the user can select one of the atoms in the gas tank and duplicate it. The new atom will serve as the icon replacing the textual label. Structural reification is at play here, making display objects accessible for direct and immediate modification.

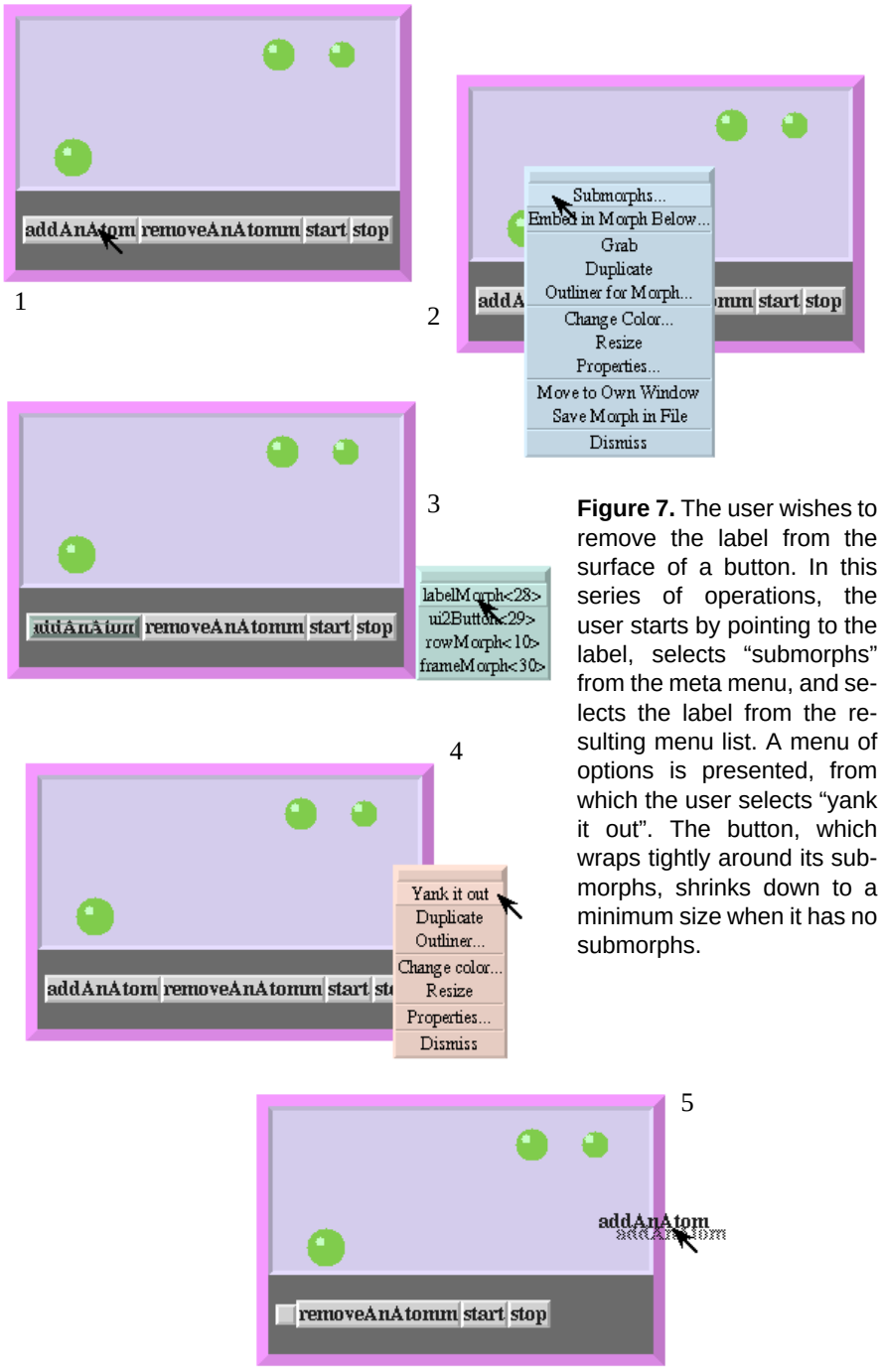

Figure 7. The user wishes to remove the label from the surface of a button. In this series of operations, the user starts by pointing to the label, selects “submorphs” from the meta menu, and selects the label from the resulting menu list. A menu of options is presented, from which the user selects “yank it out”. The button, which wraps tightly around its submorphs, shrinks down to a minimum size when it has no submorphs.

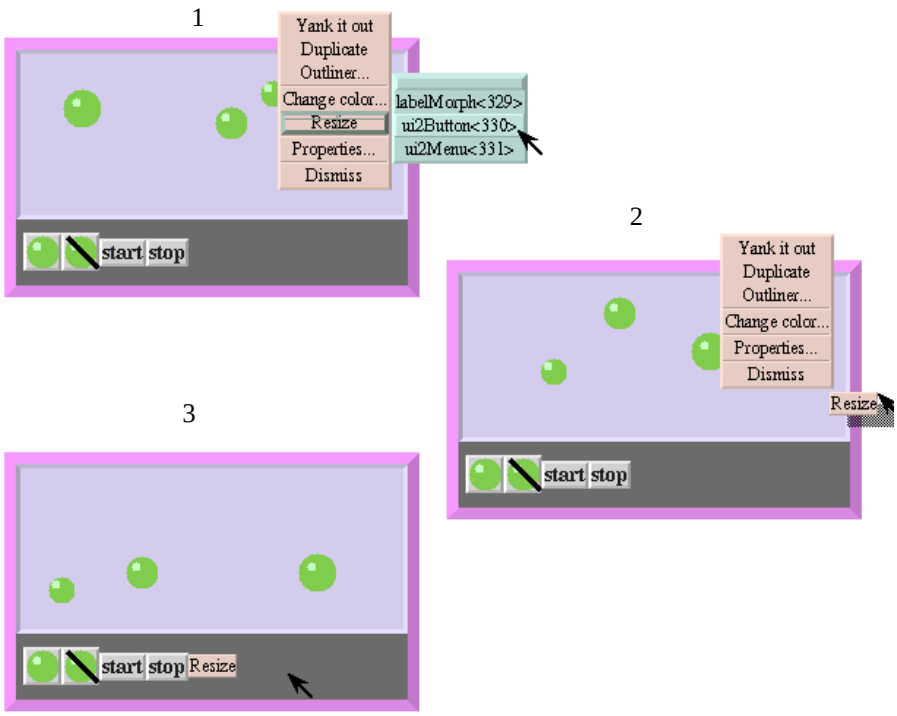

Figure 8. The environment itself is available for reuse. Here the user has created the menu of operations for the gas tank, which is now a submorph of the surrounding frame. He has "pinned down" this menu, by pressing the button at the top of the menu. He can then take the menu apart into constituent buttons: here the is gets the resize button which is then incorporated into the simulation.

As mentioned above, all the elements of the interface such as pop-up menus and dialogue boxes are available for reuse. As an example, the programmer may want the gas tank in the simulation to be "resizable" by the simulation user. The programmer can create a resize button for the gas tank simply by "pinning down" the meta menu and removing the resize button from the menu, as illustrated in Figure 8. This button can then be embedded into the row of controls along with the other buttons.

Figure 9 illustrates how the programmer can modify the behavior of an individual atom to reveal its energy based upon color. A morph has a slot called "rawColor" that normally contains a "paint" object: in this example the programmer replaces that object with a method, so that the paint will be computed based upon energy level. When the change is accepted, the modified slot immediately takes effect. In Figure 10, the programmer is shown copying the slot into the atom's parent object, so that it can be widely shared with other atoms. Of all atoms have a rawColor slot that overrides this slot in the parent. The programmer might at this point remove the rawColor slot from the prototypical atom, so that all new atoms will have this energy-based color.

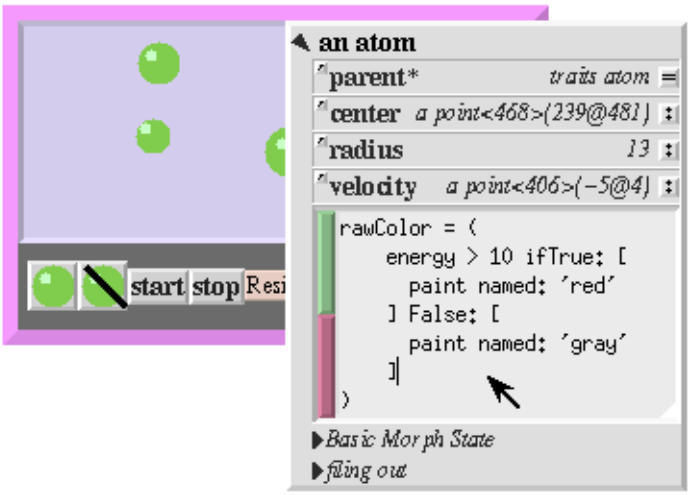

Figure 9. The user has selected one atom on which to experiment. The user changes the "rawColor" slot from a computed to a stored value by editing directly in the atom's outliner.

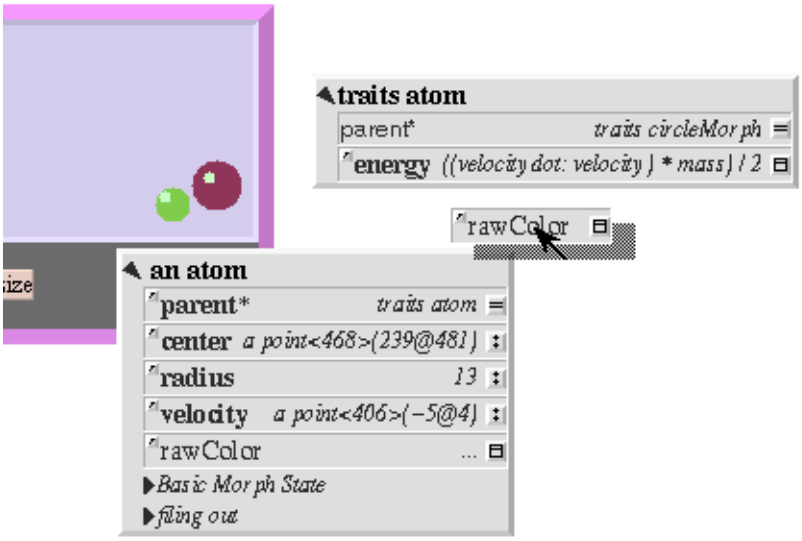

Figure 10. The user copies the modified rawColor slot as a first step in getting the computed method of coloring more widely shared. Because slots can be moved about readily, restructuring changes are relatively light weight, enhancing the sense of flexibility.

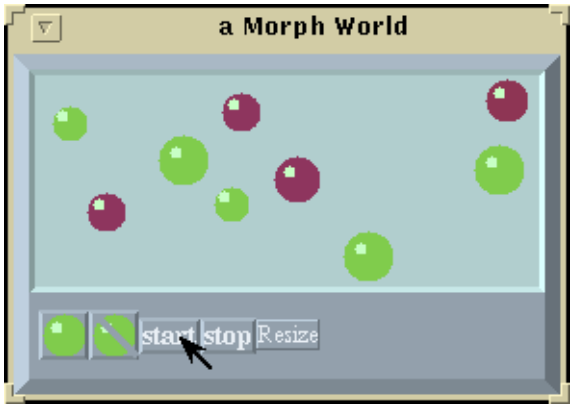

Figure 11. The completed application. The user has invoked a color changing tool that unifies colors across most submorphs, and has invoked the meta menu item “move to own window.”

Figure 11 shows the completed application. All the atoms now reveal their energy as they move, and controls for running the simulation appear in a row across the bottom of the object. The programmer has invoked a meta-menu item “move to own window” that wraps the application in its own window-system frame.

It is important to note that during this whole process, the simulation could be left running — there was no fundamental need to enter an “edit” mode, or even stop the atoms from bouncing around. The live editing principle makes the system feel responsive, and is reminiscent of the physical world’s concrete presence. Structural reification means that the parts of the interface are visible and accessible for direct modification.

3.4 Issues

While the principles of live editing and structural reification help create the sense of working within a world of tangible malleable objects, we could imagine going further. Here we discuss some of the things that interfere with full realization of our goals.

Multiple views: The very existence of the outliner as a separate view weakens the sense of directness that we are after. After all, when I want to add a slot to one of the simulated atoms, I must work on a different display object, the atom’s outliner. We have never had the courage or time to go after some of the wild ideas that would enable unification of any morph with its outliner. Ironically, Self’s first interface, Seity, probably did better on this issue [Cha95], [CUS95].

Self’s programming environment enforces the constraint that there only be one outliner on the screen at a time for a given object. If the user asks to see some object when its

outliner is already on the screen, the outliner will do an animated slide over to the mouse cursor. This constraint encourages identification between the object and its outliner in the programmer's mind. But as we mention above, particularly for morph objects, the identification does not always hold.

Because there is a difference between outliners and the objects they represent, there is a fundamental dichotomy in the system that interferes with the direct object experience goal. Is the programmer to believe that the outliner for some list object really is the list? If so, does the list have a graphical appearance as an intrinsic part of itself? Does the list have an intrinsic display location and a color? We have chosen the easy answer, "no." The object that represents the list to the programmer is not the same as the actual Self list object. Once we take this easy road, it is fundamentally impossible to always maintain the impression that the objects on the screen actually are the Self objects. Unfortunately, it is often clear that outliners are just display stand-ins for the real, invisible Self object, buried somewhere in the computer.

Text and object: There is a fundamental clash between the use of text and the use of direct manipulation. A word inherently denotes something, an object does not necessarily denote anything. That is, when you see the word "cow," an image comes to mind. It is in fact difficult to avoid the image, that is the way words are supposed to work. They stand for things. However, when you manipulate a pencil, what comes to mind? It depends much more on who you are, what the context is, and so on. In other words, a pencil does not by itself denote anything. Consequently, textual notation and object manipulation are fundamentally from two different worlds. The pencil and the word denoting pencil are different entities.

Text is used quite a bit in Self, and its denotational character weakens the sense of direct encounter with objects. For example, many tools in the user interface employ a "printString" to denote an object. The programmer working with one of these tools might encounter the text "list (3, 7, 9)." The programmer might know that this denotes an object which could be viewed "directly" with an outliner. But why bother? The textual string often says all he needs to know. The programmer moves on, satisfied perhaps, yet not particularly feeling like he has encountered the list itself. The mind set in a denotational world is different than that in a direct object world, and use of text creates a different kind of experience.

Issues of use and mention in direct manipulation interfaces are discussed further in [SUC92].

3.5 Summary

The Self user interface helps the programmer feel that he or she is directly experiencing tangible objects. Two design principles help achieve this feeling. First, *structural reification*, assures that the graphical relationships at play in a particular arrangement of submorphs is manifest directly in display objects on the screen. Second, *live editing*

means that there is no need for a course grained "edit mode;" rather, objects are always available for immediate and direct editing. The use of textual names for objects, and the distinction between an object and its representation are two problems that weaken the experience of directly working with objects. But on balance, we feel that the Self user interface successfully presents the illusion of being a world of readily modified, physically present objects.

4 Implementing Self

The implementation of a language is usually approached from a mathematical, or mechanistic viewpoint: what is desired is the creation of a program that interprets programs in another language (be the created program a compiler or interpreter). On the other hand, we have taken the view that the goal of the implementation is to fool the user into believing in the reality of the language. Even though Self objects have no physical existence, and there is no machine capable of executing Self methods, the implementation must strive to convince the user otherwise. That is why, despite all the tricks, the programmer can always debug at the source level, seeing all variables and single stepping, and can always change any method, even inlined ones, with no interference from the implementation.

4.1 Transparent Efficiency

The implementation techniques for Self have been presented previously so we will only summarize the briefly here. (See [Hol94], [HU94a], [HCU92], [HCU91], [Cha92], [CU91], [CU90], [CUL89], and [USCH92] for more details.)

Self presented large efficiency challenge because its pure semantics implied that every access, assignment, arithmetic operation and control structure had to be performed by sending a message. Worse yet, the Self user interface's uncompromising stance on structural reification placed further demands on efficiency: everything on the screen down to the smallest triangle is implemented by its own separate object, each with its own redraw and active layout behavior. At the same time, in order to produce the experience of concrete objects, the system had to be as responsive as an interpreter. Self's implementors were confronted with a fundamental problem: to be responsive, the compiler could not afford to spend time on the elaborate optimizations needed for the language, no matter how effective they might be. Caught between Scylla and Charybdis, Self needed something completely different. Instead of relying on a single compiler for both speed and cleverness, Self adopted a hybrid system of two compilers: one fast, the other clever. Instead of always optimizing everything, type feedback permits the system to adaptively optimize code without introducing long pauses.

Figure 12 shows an overview of the compilation process of the system. The first time that a method is invoked, the virtual machine uses dynamic compilation to create an "instrumented" version of the method that counts its invocations and the types of the receivers at each call site. When the invocation counter crosses some threshold, the opti-

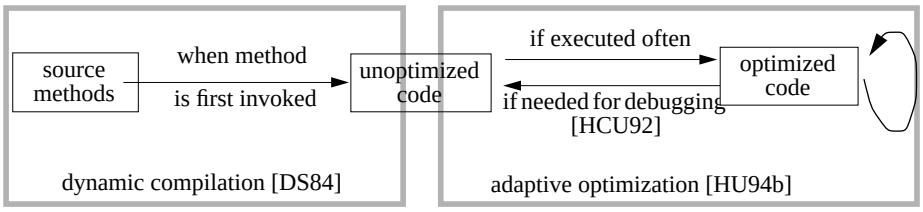

Figure 12. Compilation in the Self system (from [HU94b]).

mizing compiler is automatically invoked and is guided by the counters at the call sites. In this way, Self feeds type information back to the compiler to adaptively optimize code.

Some language implementations force the programmer to choose between interpretation and compilation, or between different modes of compilation. Placing this burden of choice upon the programmer's shoulders can only weaken his confidence in the reality of Self, and force him to consider the difference between the program as written and what really runs. Although Self employs two compilers and a myriad of optimizations, the programmer never chooses nor even knows which have been employed on his code.

Results on two medium-size cross-language benchmarks (Richards and DeltaBlue) suggest that Self runs 2 to 3 times faster than ParcPlace Smalltalk-80[1], 2.6 times faster than Sun CommonLisp 4.0™ using full optimization, and only 2.3 times slower than optimized C++ [HU94a]. Of course, these Smalltalk and Lisp implementations may not include aggressively-optimized compilers, but the C++ language has semantics that make many more concessions to efficiency over purity, simplicity, and safety.

Most implementations strive for efficiency and employ optimizations that show through to the programmer. Tail recursion elimination, for example, optimizes methods that iterate by calling themselves at their ends, but makes it impossible to show a meaningful stack trace. This destruction of information would show through to the user, who might need to see the missing stack frames in order to debug her program. So, Self does not optimize tail-recursion. Instead, endless loops are built in as a primitive operation. There are other optimizations left undone in Self, see [Hol94] for a list.

4.2 Responsiveness

Many systems impose long or unpredictable pauses upon their users. But, a pause in the middle of a user task such as the addition of a slot to an object could ruin the experience of a consistent world. Such pauses, by their very existence, alert users that some mischief is afoot. If the program (be it Self or any other language) were the reality, there would be no pauses upon changing it. Since we believe that such pauses can destroy the fragile illusion of reality, we have striven to reduce them in the Self implementation. In

[1] Smalltalk-80™ is a trademark of ParcPlace Systems.

fact, pauses for compilation were a serious problem in early versions of Self, and inspired an effort to speed up the compiler. Now, any method can be changed in a second or two. Our ultimate goal is the elimination of perceptible pauses.

In [HU94b], Hölzle and Ungar measured the compilation pauses occurring during a 50-minute session of the SELF 3.0 user interface[1] [Cha95] [CUS95]. Their analysis grouped individual pauses into clusters that would be perceived as pauses by the users. The results indicated that there were few intrusive clustered pauses; on a current-generation workstation, only 13 pauses would exceed 0.4 seconds, and on a next-generation machine, none would exceed 0.3 seconds (see Figure 13).

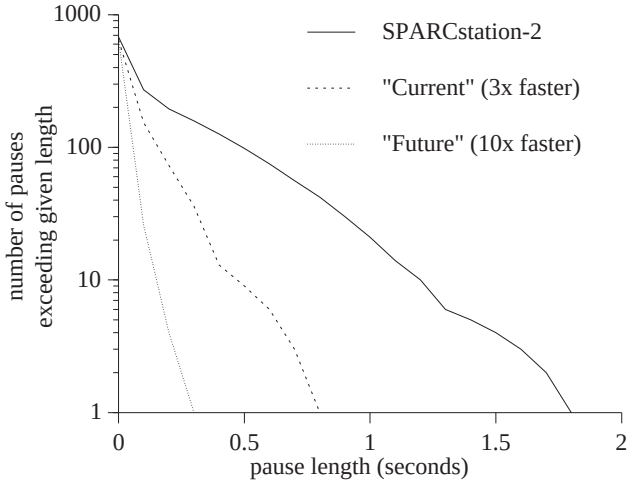

Figure 13. Compilation pauses (from [HU94b]).

Clustering pauses made an order-of-magnitude difference, and reporting individual pauses would have resulted in a distorted, overly optimistic characterization. The idea of pause clustering is one example of how our vision of providing a particular kind of experience to the programmer affects the standards that must be used to evaluate the system.

4.3 Malleability

By now, it should be clear to the reader that our philosophy of implementation as deception places additional burdens on the implementation, such as avoiding inconsistent behavior or unexplainable pauses. What may come as a surprise, though is that a requirement for malleability arises as a logical consequence of this unusual philosophy. For example, in both Smalltalk and Self, the if-then-else control structure is realized by sending a message “ifTrue:IfFalse:” to a boolean object (two arguments are included, a block to execute for true, and another for false). Each boolean simply implements this

[1] The Self 4.0 user interface described in section 3 places more demands on the implementation and its pause behavior is not as good as these measurements suggest.

message with a message that executes the appropriate block; true's ifTrue:IfFalse: method runs the true block and false's runs the false block. If Self or Smalltalk objects are real, if they directly and faithfully execute, if the user is in control, then the user should be able to change these methods and observe the results. But in Smalltalk, extra performance is obtained by short-circuiting this method in the implementation [GR83] and though he may change the method, the programmer cannot alter its behavior. This failure of malleability cannot help but raise disturbing questions and insecurities for the programmer. That is why Self does not short-circuit this or other such methods, even though providing such malleability without an efficiency loss extracts its cost from the implementation.[1]

4.4 Encouraging Factoring and Extensible Control

Our non-traditional approach has led to techniques that provide a very traditional benefit: factoring is free. Suppose two methods in the same object contain several lines in common. By moving the common lines into a new method and sharing the method in both callers, the programmer can centralize them and make the program easier to change. Most implementations of object-oriented programming languages would slow down the program with an additional procedure call, providing a strong disincentive to the programmer for factoring small operations. In C++, for example the programmer must either ask explicitly for inlining (and give up virtual semantics) or pay the price of a many-statement overhead. Since Self relies on automatic inlining, no such price is paid. Consequently Self programmers feel much freer to factor programs, and the system is written in an unusually well-factored style. For example, a do-while loop is implemented with many levels of message passing before bottoming out in primitives. We believe that the performance characteristics of Self's implementation techniques encourage programmers to write programs that are easier to maintain.

In addition to free factoring, the Self implementation makes user-defined control structures as efficient as the built-in ones. Unlike for Smalltalk, there is no disincentive for the programmer to use a block; the implementation in most cases can inline it away. Since we believe that control abstraction is necessary for real data abstraction, making control abstraction free can help encourage programmers to write better programs. We strongly believe that in all languages with user-extensible control, such as Smalltalk and Beta, much benefit could be realized from adopting implementation techniques that put the user-defined control structures on the same footing as the system-defined ones.

4.5 Open Issues

Although the current Self system is in daily use by a number of people, several concerns remain about its implementation.

Overcustomization. The Self compiler creates another copy of a method for each kind of object that inherits it. Sometimes, the method is so trivial that the copies waste code space and compiler time for no good reason.

---

[1] In fact, at first the second author thought the cost would be too great. But Craig Chambers' compiler convinced him otherwise.

Memory Footprint. Because enough information is preserved to maintain source-level semantics, Self takes more space than other systems. Self 4.0 barely fits in a SPARCstation[1] with 32 Mb of real memory. Although we believe that programming time is more precious than memory cost, this resource requirement puts Self out of reach of many current users.

Real-Time Operation. Although much progress has been made in the elimination of perceptible pauses, the system still feels like it "warms up" when running the Self 4.0 environment. Hard real-time operation is not possible with today's system.

User Control. Within the Self group, a debate rages over how much control a user should have over the optimization process. On one hand, users want to be able to tell the system how much, where, and when to optimize. On the other hand, the effort to add this ability might be better spent doing a better job automatically, and giving users this control could distract them from their own tasks and destroy the fragile illusion of Self's reality. So far, we have kept the control over optimization entirely within the virtual machine, as an experiment in the philosophy of implementation as deception.

4.6 Summary

Confidence, comfort, satisfaction—what do these desires imply for implementation techniques? They rarely show up as topics in compiler papers, yet we believe that these goals have exerted a profound influence over Self's implementors.

5 Conclusions

The Self language semantics, implementation, and user interface have been guided by the goal of generating a particular kind of experience for the user of the system. Programmers directly work in a uniform world of malleable objects in a very immediate and direct fashion. Self moves towards giving objects a kind of tangible reality.

The language helps give rise to this experience by its use of prototypes, which provide a copy-and-directly-modify mechanism for changes. Self's treatment of slots with its symmetry for assignment and access reflect the deep connection between perception and manipulation at the sensory-motor level, while also enabling objects to reimplement state as behavior and reflecting an intuition about how objects are behaviorally perceived in the real world.

Self's design departs significantly from other object-oriented languages by separating information needed to run the program from information about the programmer's intentions. It distinguishes abstract types, used for the programmers understanding and reasoning about correctness, from concrete types, used to run and optimize the program. The former is left to the environment, the latter is left to the implementation. In our

[1] SPARCstation™ is a trademark of SPARC International, licensed exclusively to Sun Microsystems Inc.

opinion, this approach avoids a number of undesirable consequences that often follow from attempts to integrate these two forms of information.

Finally, in designing Self, we have learned one lesson by making mistakes: examples can persuade the designer to include additional features which later turn out to produce incomprehensible behavior in unforeseen circumstances. This might be called "the language designer's trap." Minimalism, simplicity and consistency are better guides. They benefit every programmer, not just the ones who need advanced features. We suspect that many of today's object-oriented languages could profit by dropping features.

The Self user interface and programming environment provides a direct object experience for creating objects and assembling applications by adhering to two principals: *Structure reification* makes the graphical containment structure and layout rules themselves appear as graphical objects and assures that any graphical object can be manipulated, displayed, connected to other objects, or customized. *Live editing* ensures that any object may be changed at any time without halting activities. A simple gesture takes the user from any graphical object to its programming-language-level counterpart, just as real-world objects can be taken apart. Consequently, programmers need not pore through long object libraries to find out where to start, but can simply find some graphical widget in the environment, like a button in a menu, that is similar to what they want, and proceed to dissect, inspect, modify and reassemble it. At no time must they retreat from a concrete object to some definition of an abstraction.

Two problems in the user interface interfere with achieving our goal. The existence of multiple views for a graphical object and its Self-level outliner dilutes the experience and these views should be merged. The duality between text and object goes deeper and does not readily present a solution.

The consistency and purity of the Self language together with the ubiquitous use of objects and live layout in the interface place enormous demands on the Self implementation, but more interestingly, the desire to create a particular kind of programming experience imposes its own unique requirements on the implementation. Thus, the implementation foregoes optimizations that cannot be hidden such as tail-recursion elimination. It also supports full source-level debugging, single stepping, and allows the programmer to change any method, even basic ones such as addition and if-then-else, at any time. Since a long pause for compilation would alert the programmer to the existence of a lower level of reality, the implementation works hard to avoid such pauses. We view the implementation not as an interpreter of programs, but rather as a creator of the illusion that the Self objects are real.

Along the way Self's implementation techniques of adaptive recompilation and type-feedback achieve some traditionally-important but rarely achieved goals as well: the elimination of run-time penalty for factoring and for user-defined control structures. A programmer may chop up a method as finely as desired without slowing it down, and may introduce new abstractions that combine control and data without paying a run-

time price. These characteristics encourage the create of programmers that are smaller and more malleable.

When all is said and done though, this paper can only suggest, tease, or maybe hint at what it is like to create with Self. In order to most fully appreciate the experience of interacting with a lively, responsive world of objects, effortlessly diving in to change them and create more, freely mixing data and programs, and only getting coffee when you are tired instead of when you change your program, you will have to obtain the Self 4.0 public release and try it out for yourself. May your journey be fruitful.

6 Acknowledgments

The past and present members of the Self group, highly talented individuals each, have made this body of work possible. The authors consider themselves fortunate to have known and worked with them. Sun Microsystems Laboratories has hosted the project for the past four years, for which we are deeply grateful. Special thanks to Mario Wolczko and Ole Lehrmann Madsen for comments on the draft.

7 References

[APS93] Ole Agesen, Jens Palsberg, and Michael I. Schwartzbach. *Type Inference of Self: analysis of Objects with Dynamic and Multiple Inheritance,* in Proc. ECOOP '93, pp. 247-267. Kaiserslautem, Germany, July 1993.

[Blas94] G. Blaschek. **Object-Oriented Programming with Prototypes,** Springer-Verlag, New York, Berlin 1994.

[BD81] A. Borning and R. Duisberg, *Constraint-Based Tools for Building User Interfaces,* ACM Transactions on Graphics 5(4) pp. 345-374 (October 1981).

[CUL89] Craig Chambers, David Ungar, and Elgin Lee. *An Efficient Implementation of Self, a Dynamically-Typed Object-Oriented Language Based on Prototypes*. In OOPSLA '89 Conference Proceedings, pp. 49-70, New Orleans, LA, 1989. Published as SIGPLAN Notices 24(10), October, 1989.

[CU90] Craig Chambers and David Ungar. *Iterative Type Analysis and Extended Message Splitting: Optimizing Dynamically-Typed Object-Oriented Programs.* In Proceedings of the SIGPLAN '90 Conference on Programming Language Design and Implementation, White Plains, NY, June, 1990. Published as SIGPLAN Notices 25(6), June, 1990.

[CU91] Craig Chambers and David Ungar. *Making Pure Object-Oriented Languages Practical.* In OOPSLA '91 Conference Proceedings, pp. 1-15, Phoenix, AZ, October, 1991.

[CUCH91] Craig Chambers, David Ungar, Bay-Wei Chang, and Urs Hölzle, *Parents are Shared Parts of Objects: Inheritance and Encapsulation in Self*. Journal of Lisp and Symbolic Computation, **4**(3), Kluwer Academic Publishers, June, 1991.

[Cha92] Craig Chambers. **The Design and Implementation of the Self Compiler, an Optimizing Compiler for Object-Oriented Programming Languages.** Ph.D. Thesis, Computer Science Department, Stanford University, April, 1992.

[CUS95] Bay-Wei Chang, David Ungar, and Randall B. Smith, *Getting Close to Objects*, in Burnett, M., Goldberg, A., and Lewis, T., editors, **Visual Object-Oriented Programming, Concepts and Environments,** pp. 185-198, Manning Publications, Greenwich, CT, 1995.

[Cha95] Bay-Wei Chang, Seity: **Object-Focused Interaction in the Self User Interface,** Ph.D. dissertation, in preparation, Stanford University, 1995.

[DS84] L. Peter Deutsch and Allan M. Schiffman. *Efficient Implementation of the Smalltalk-80 System.* In Proceedings of the 11th Annual ACM Symposium on the Principles of Programming Languages, pp. 297-302, Salt Lake City, UT, 1984.

[DMC92] C. Dony, J. Malenfant, and P. Cointe, *Prototype-Based Languages: From a New Taxonomy to Constructive Proposals and their Validation*, in Proc. OOPSLA '92, pp. 201-217.

[GR83] Adele Goldberg and David Robson, *S***malltalk-80: The Language and Its Implementation.** Addison-Wesley, Reading, MA, 1983.

[HCU91] Urs Hölzle, Craig Chambers, and David Ungar. *Optimizing Dynamically-Typed Object-Oriented Programs using Polymorphic Inline Caches.* In ECOOP' 91 Conference Proceedings, pp. 21-38, Geneva, Switzerland, July, 1991.

[HCU92] Urs Hölzle, Craig Chambers, and David Ungar. *Debugging Optimized Code with Dynamic Deoptimization*, in Proc. ACM SIGPLAN '92 Conferences on Programming Language Design and Implementation, pp. 32-43, San Francisco, CA (June 1992).

[Hol94] Urs Hölzle. **Adaptive Optimization for Self: Reconciling High Performance with Exploratory Programming.** Ph.D. Thesis, Stanford University, Computer Science Department, 1994.

[HU94a] Urs Hölzle and David Ungar. *Optimizing Dynamically-Dispatched Calls with Run-Time Type Feedback.* In Proceedings of the SIGPLAN 94 Conference on Programming Language Design and Implementation, Orlando, FL, June, 1994.

[HU94b] Urs Hölzle and David Ungar. *A Third Generation Self Implementation: Reconciling Responsiveness with Performance.* In OOPSLA'94 Conference Proceedings, pp. 229-243, Portland, OR, October, 1994. Published as SIGPLAN Notices 29(10), October, 1994.

[MMM90] Ole Lehrmann Madsen, Boris Magnusson, and Birger Møller-Pedersen, *Strong Typing of Object-Oriented Languages Revisited.* In ECOOP/OOPSLA'90 Conference Proceedings, pp. 140-149, Ottawa, Canada, October, 1990.

[MMN93] Ole Lehrmann Madsen, Birger Møller-Pedersen, Kristen Nygaard, **Object-Oriented Programming in the Beta Programming Language,** Addison-Wesley Publishing Co., Wokingham, England, 1993.

[Mul95] Phillipe Mulet, **Réflexion & Langages á Prototypes,** Ph.D. Thesis in preparation, Ecole des Mines de Nantes, France, 1995.

[MS95] John Maloney, and Randall B. Smith, *Directness and Liveness in the Morphic User Interface Construction Environment*. In preparation.

[Smi87] Randall B. Smith. *Experiences with the Alternate Reality Kit, an Example of the Tension Between Literalism and Magic*, in Proc. CHI + GI Conference, pp 61-67, Toronto, (April 1987).

[SUC92] Randall B. Smith, David Ungar, and Bay-Wei Chang. *The Use Mention Perspective on Programming for the Interface*, In Brad A. Myers, **Languages for Developing User Interfaces,** Jones and Bartlett, Boston, MA, 1992. pp 79-89.

[SLS94] R. B. Smith, M. Lentczner, W. Smith, A. Taivalsaari, and D. Ungar, *Prototype-Based Languages: Object Lessons from Class-Free Programming (Panel)*, in Proc. OOPSLA '94, pp. 102-112 (October 1994). Also see the panel summary of the same title, in Addendum to the Proceedings of OOPSLA ‘94, pp. 48-53.

[SMU95] Randall B. Smith, John Maloney, and David Ungar, *The Self-4.0 User Interface: Manifesting the System-wide Vision of Concreteness, Uniformity, and Flexibility.* To appear in Proc. OOPSLA ‘95.

[Tai92] Antero Taivalsaari, **Kevo - a prototype-based object-oriented language based on concatenation and module operations.** University of Victoria Technical Report DCS-197-1R, Victoria, B.C., Canada, June 1992

[Tai93] Antero Taivalsaari, **A critical view of inheritance and reusability in object-oriented programming.** Ph.D. dissertation, Jyvaskyla Studies in Computer Science, Economics and Statistics 23, University of Jyvaskyla, Finland, December 1993, 276 pages (ISBN 951-34-0161-8).

[Tai93a] Antero Taivalsaari, *Concatenation-based object-oriented programming in Kevo.* Actes de la 2eme Conference sur la Representations Par Objets RPO'93 (La Grande Motte, France, June 17-18, 1993), Published by EC2, France, June 1993, pp.117-130

[US87] David Ungar and Randall B. Smith, *Self: The Power of Simplicity*, Proceedings of the 1987 ACM Conference on Object Oriented Programming Systems, Languages, and Applications (OOPSLA), Orlando, FL, October, 1987, pp. 227–242. A revised version appeared in the Journal of Lisp and Symbolic Computation, 4(3), Kluwer Academic Publishers, June, 1991.

[USCH92] David Ungar, Randall B. Smith, Craig Chambers, and Urs Hölzle. *Object, Message, and Performance: How They Coexist in Self.* Computer, 25(10), pp. 53-64. (October 1992).

[Wol95] Mario Wolczko, *Implementing a Class-based Language using Prototypes*, In preparation.