Annotating Objects for Transport to Other Worlds

David Ungar
Sun Microsystems Laboratories
2550 Garcia Ave., Mountain View, CA 94043
(415) 336-2618, (415) 969-7269 (fax)
david.ungar@sun.com
http://self.smli.com/~ungar

Abstract

In Self 4.0, people write programs by directly constructing webs of objects in a larger world of objects. But in order to save or share these programs, the objects must be moved to other worlds. However, a concrete, directly constructed program is incomplete, in particular missing five items of information: which module to use, whether to transport an actual value or a counterfactual initial value, whether to create a new object in the new world or to refer to an existing one, whether an object is immutable with respect to transportation, and whether an object should be created by a low-level, concrete expression or an abstract, type-specific expression. In Self 4.0, the programmer records this extra information in annotations and attributes. Any system that saves directly constructed programs will have to supply this missing information somehow.

1. Introduction

Computer Scientists usually think of a computer program as a static description, abstraction, or model of a dynamic computation. This formulation has led to significant accomplishments: methods for the analysis of complex problems, a clean separation of programming from execution, frameworks for reasoning about a computation's correctness and speed, ways to capture intangible intentions and plans (as declarations), and pedagogical advantages for people with mathematical backgrounds. Given these benefits, it is not surprising that most object-oriented programming systems require some descriptive information right at the start. For example, before a Smalltalk, C++, Eiffel, or Beta programmer can begin to experiment with a point object, he must first step back and define a description, the class of all possible points. Or, before a C++ programmer can use a hand-drawn icon in her program, she must figure out some way to get the picture data into her program (see Figure 1).

But, it may not always be appropriate to enforce description before experimentation. While some problems may demand a lot of prior analysis, others may demand more exploration of the solution space. Some programmers may work best by designing first and coding later, but others, especially more casual programmers solving smaller problems, may work better by iterating construction and design. We are interested in building a programming environment that ultimately can support both styles in order to foster creativity, exploration, and accessibility.

The Self object-oriented programming system strives for a more concrete and direct feeling [Smith 1], [Smith 2]. We hope that this emphasis will make it easier to explore and create desired behaviors and that the more descriptive information can be added later, if so desired. That is why the language is based upon prototypes instead of classes, why the implementation hides the information needed to describe objects' formats, and why the user interface supports hyperdirect manipulation. That is also why a Self

OOPSLA '95 Austin, TX, USA
© 1995 ACM 0-89791-703-0/95/0010...$3.50

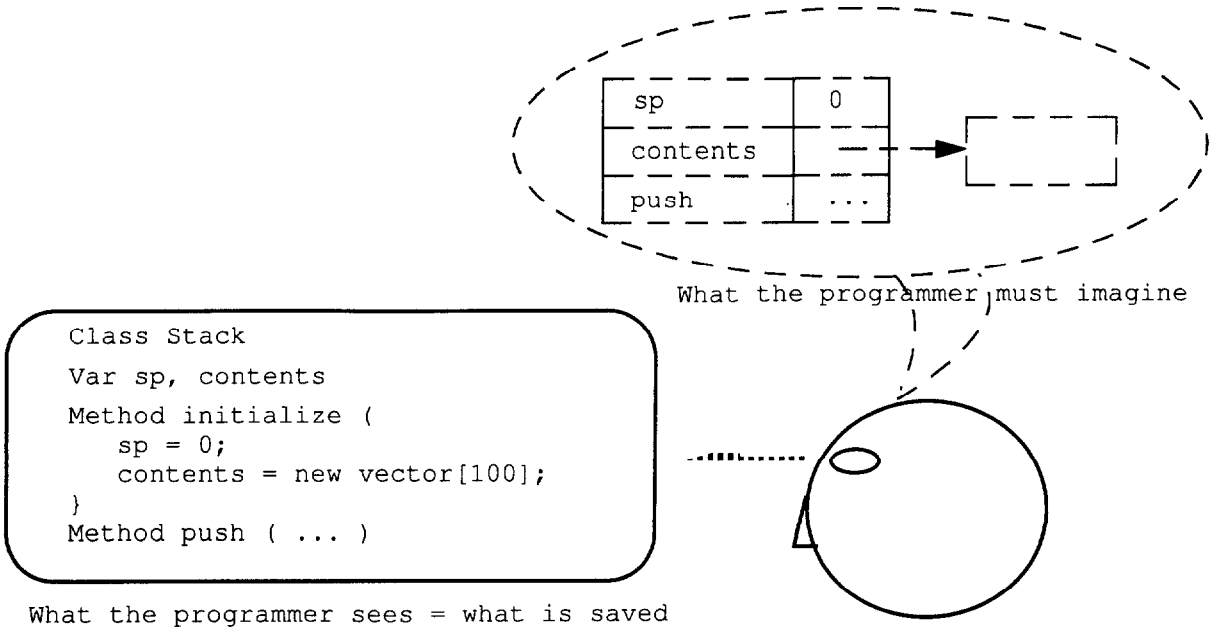

Figure 1. A program as a description: Traditionally, a program is thought of as describing objects to be created at run-time. Because it is a description, it can be directly used in many worlds of objects. For example, the stack class shown here could be used in many separate worlds of objects. But, being a description, the program cannot be directly used to visualize the objects in a running program; the programmer must make a mental leap.

program is considered to be a collection of live, running objects, rather than static text. Since the programmer directly manipulates objects by adding methods or variables and by setting variables to their initial values, we say that the Self system and its applications are built out of *directly constructed objects or programs*. Such a concrete notion of a program can bring the programmer closer to his task, as if he were touching them instead of looking at them (Figure 2).

However, any system that posits a less-descriptive model of programming must address some thorny issues, and one of the thorniest is saving programs. A program (or cluster of objects) constructed in one world of objects (a.k.a. a snapshot, or virtual image) needs to be saved in some archival form such as a source file so that it can be moved into another world of objects. While the code can just be copied, the web of data references defies simple transcription. For example, the user might build a stack object out of a stack pointer and a vector, then try to move the stack into another world of objects. A literal move would try to move the vector object and would literally copy the stack pointer value, when what the user really intends is to create a new vector and set the stack pointer to zero. *A system that attempts to move directly constructed objects like this stack must confront the gap between the extensional information present in memory and the intentional information it needs.*

Recognizing and characterizing this gap was the hardest part of building the Self 4.0 transporter, a subsystem intended to move directly constructed Self programs from one world to another. In the course of developing the transporter we were forced to confront many situations in which different kinds of missing information had to be supplied. In other words, by taking on the task of recreating a more descriptive, intentional program from an existential, extensional web of objects, we had to rediscover what would have been conveyed in the description that was not contained in the objects. At this writing (1995), the transporter has gone through one major redesign, has been in daily use for 2 years in its present form, and is part of a publicly available system.[†] This paper

[†] Browse http://self.smli.com or ftp from self.smli.com.

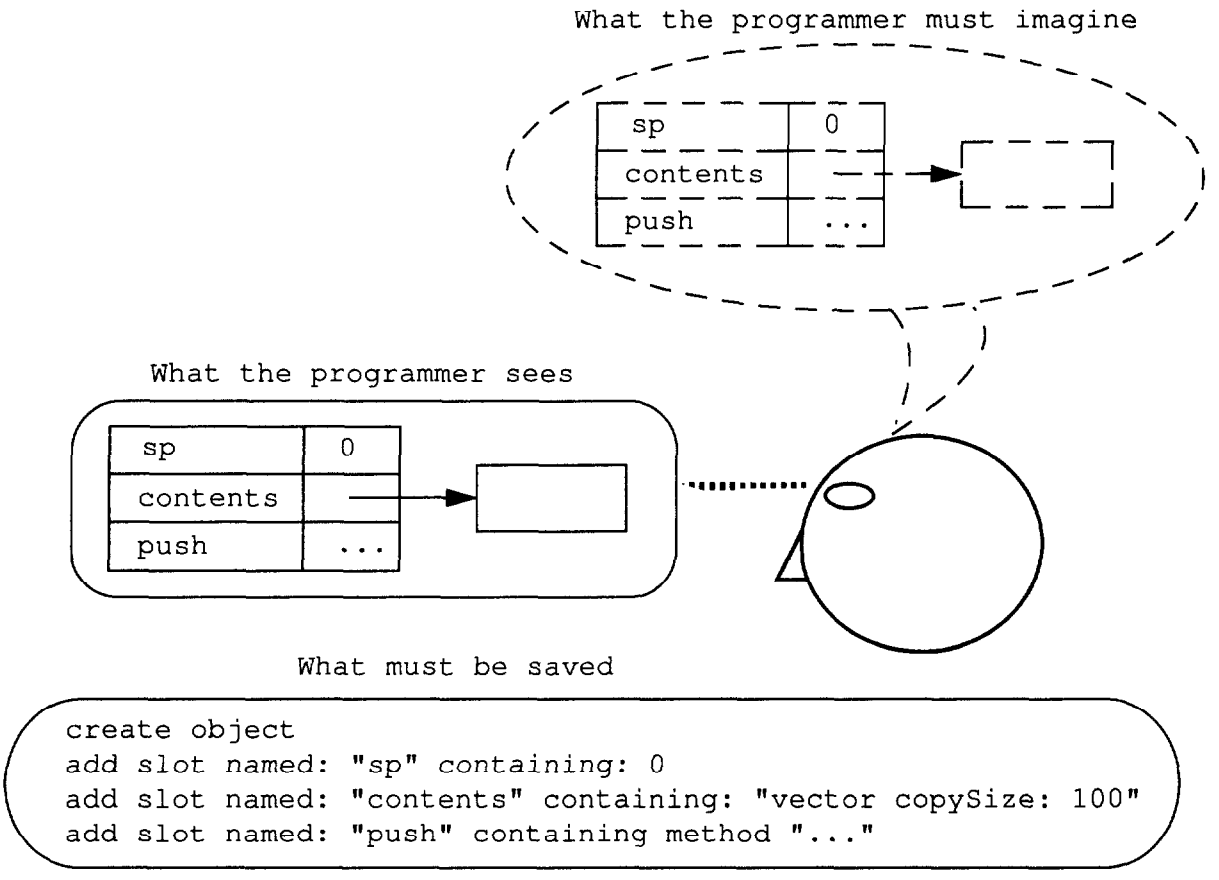

Figure 2. A program as the objects themselves: When a program is directly constructed with concrete objects, the programmer does not have to make such a large mental leap from the program to the actual objects. However, this view of a program omits information that would be needed in order to save it in a form that could be imported into another world of objects. In this example, although a stack object may have any value in its stack pointer, the source should always create the `sp` slot with zero in it. This information about initial values is missing from the objects themselves.

summarizes what we have learned about the gap between conventional, descriptive programs and directly constructed ones. We hope that other researchers who are faced with similar problems can profit from a discussion of the difference between the information needed to perform a given object-oriented computation and the information needed to transport the objects that perform that computation into another world.

The rest of this paper introduces our modules in section 2, discusses the pieces of descriptive information needed to transport programs in section 3, presents some measurements in section 4, recounts prior work in section 5, and summarizes itself in section 6.

2. Background

Before discussing the extra information needed for any system that transports directly constructed programs, it will be helpful to explain some design choices peculiar to the Self 4.0 transporter. Others will confront these issues, but may well choose different solutions.

2.1 What is a module? Changes or Pieces

As mentioned above, this work addresses a need to move programs from one web of objects to another. Accordingly, it would seem to follow that a program should be modelled as a change to a web of objects. Alternatively, a program could be modelled as pieces of stuff (objects, slots) in the world of objects. For

example, in the "changes" model, a programmer could point to a method and ask "What module caused this object to be created?" whereas in the "pieces" model a programmer would ask "To which module does this object belong?" The "changes" model holds out the promise of a powerful, rigorous framework: If a program could be represented as a set of changes, and if these changes were appropriately commutative, it would be easy to transport it.[†] The "pieces" model seems more concrete, since an object or slot is easier to picture than a change.

Given our obsession with concreteness, it is not surprising that we chose the "pieces" model. Two smaller concerns also contributed to this choice: minimizing the research risk in an essential portion of infrastructure, and the desire to be able to bootstrap the system by reading source files into an empty world. We therefore opted to save the pieces as Self source expressions suitable for bootstrapping and to continue to manage changes with a conventional source code control system, the Revision Control System (RCS) [Tichy].

Although other systems may opt for the more powerful "changes" model, they will still have to cope with a scarcity of intentional information in directly constructed programs. For example, under the "changes" model, the stack pointer is considered to be an addition to the stack instead of a part of it, but this paradigmatic shift does not resolve the dilemma of whether to save its current value or its initial value. Although we took the simpler path, these lessons should also help more adventuresome travellers.

2.2 What belongs to a module? Objects or slots?

What is the unit of object-oriented programming? In most programming textbooks writing a program is described as creating one or more new *classes*. For example, a program to construct palindromes might add a new `Palindrome` class. In Self the same operation would be the creation of one or more new objects, that could either function as prototypes, or repositories of shared behavior called traits [Ungar]. So the equivalent operation would be the creation of a new object to serve as the prototypical palindrome, and another object to be the parent of all palindromes holding the shared traits.

In a Smalltalk image there are many instances that are not part of a program, but are created by the program. These objects need not be transported, unlike the classes that comprise the program. Similarly, a Self world also contains objects created by the program that should not be saved by the transporter. However since Self unifies classes and instances, its incidental objects are harder to distinguish from the essential ones that must be written as part of the program. The Self 4.0 transporter observes this distinction by only attempting to save objects that are accessible from the lobby, which is the root of the global name space. These saved, globally accessible objects are called *well-known objects*. For example the prototype palindrome and its parent would be well-known, but most copies of the prototype would not.

On the other hand, in addition to the creation of new well-known objects, the introduction of new functionality to a system often requires the *extension of old well-known objects with new attributes*. For example, a Smalltalk program to find palindromes might add a method to class `String` called `isPal-indrome`. It could not subclass `String`, because it must operate with strings created by other programs. In Self, a slot can be used to hold a method, a local or global constant, or a class, instance, or local variable. Therefore the Self analogue to extending existing classes is the addition of slots to existing objects (see Figure 3). So for the transporter, a program is not a set of objects, but rather is composed of individual slots. Such a fine granularity adds a degree of flexibility that seems useful for any singly dispatched object-oriented language.[‡]

[†] For two interesting frameworks, see [Bracha] and [Ossher].

[‡] Object-oriented languages with multiple dispatch may be able to achieve the same flexibility by merely supplying additional arguments. For another view on why modules should be different than classes, see [Szyperski].

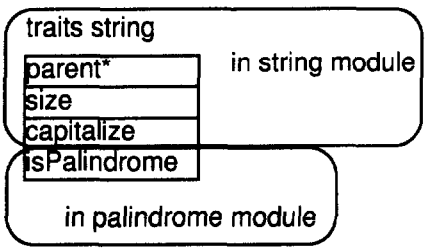

Figure 3. Why modules are composed of slots instead of objects: `Traits string` is the parent of all strings and holds their shared behavior. Most of its slots belong to the string module, but another module, `palindrome`, has been added to the system. This module has extended the behavior of all strings by adding a new slot to `traits string`, called `isPalindrome`. This incremental extension would be much harder if all the slots in an object were constrained to reside in the same module.

2.3 Source Files and Order Independence

Since the raw virtual machine reads a file at a time, one module corresponds exactly to one source file. To allow for large composite modules or subsystems, each module includes a list of submodules, so that reading in the source file for a module also causes the submodule files to be read.

Sometimes the slots in an object are spread out among several files, so that the object is built incrementally as each file is read. It is even possible to read in a file that adds a slot to an object before reading in the file that creates the object! In order to reduce the chances for disaster that sensitivity to file ordering would engender, the transporter endeavors to remain insensitive to it. For example, when reading a file that adds a slot to an object, the transporter will create a placeholder object if the object itself has not been created yet. Subsequently, when the object itself is created, any pointers to the place holder are redirected to the real object, and any slots in the place holder are added to the real object. Addressing this side issue removes one barrier to the transporter's usability; other similar systems may also have to resolve ordering issues.

3. What is Missing?

To briefly recap, the Self transporter allows the programmer to take a program that adds some functionality to a snapshot and move that program to another snapshot. It accomplishes this task by writing the slots that comprise the program to a source file that can be read in and evaluated in the new snapshot to recreate the program. However, in order to write out the slots, the transporter must recover information about the programmer's intentions that is missing from the web of objects. Any system that migrates webs of objects from world to world must face the task of recovering such information. This section enumerates the kinds of information that we believe must be so recovered. It is summarized in Table 1.

Kind of information	Needed generally vs. needed for Self	How Supplied	Where
Which module does a slot belong to?	General	per-slot annotation	section 3.1
Use slot's actual contents vs. a fixed initial value?	General	per-slot annotation	section 3.2
Should slot just reference a preexisting (global) object?	General	per-object annotation	section 3.3
Should identity of an object be respected?	General	sending message to the object	section 3.4
Is it possible to create an object with an abstract expression and if so what?	General	sending message to the object	section 3.5
Is this object ineligible for abstract creation (i.e. a prototype)?	Self	sending message to the object	section 3.5
Should this object inherit slots from another and if so, which?	Self	per-object annotation	section 3.6

Table 1. Extra information needed to move objects

3.1 Mapping Slots to Modules

The first decision facing the transporter concerns which slots to put in which modules. An earlier version of the transporter required the programmer to specify module boundaries. For example, the programmer would indicate that the global slot "point" belonged to the point module, and the transporter would infer that every slot in the transitive closure (up to slots explicitly designated in other modules) were also in the point module (Figure 4, top).

This centralization was convenient for the transporter, but difficult for the programmer, who could not readily find the information pertinent to a given object or slot. To make matters worse, the system could not even quickly tell the programmer if a module was correctly specified because an expensive global closure computation was needed to ensure that none of the module's transitively reachable slots were also ambiguously transitively reachable in some other module.

To solve these problems, the present (1995) transporter adopts a more explicit approach; each slot is annotated with the name of its module. The annotations are separate enough so that they can be normally hidden to avoid distracting the programmer when not needed, and integrated enough to feel like a concrete part of a slot or object when exposed (Figure 5). The programmer can inspect or change which module a slot belongs to by reading or writing its annotation, and the transporter can prompt for this information if it is missing. Since each object can possess slots in different modules, the programming environment shows a summary of the modules of an object, sorted by frequency (Figure 5, top).

To support this decentralization, the Self Virtual Machine was extended to allow any slot or object to be annotated with another object,[†]. In addition to holding transportation information, annotations also provide a convenient place for comments on slots or objects.

[†] The spatial overhead to support annotations is n+1 words per well-known object, where n is the number of slots in the object.

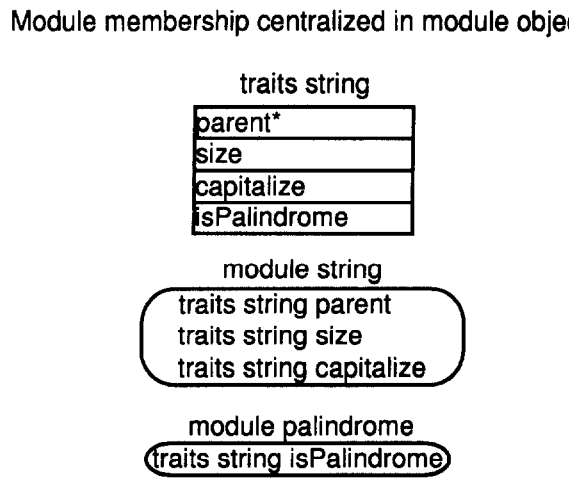

Figure 4. Centralized vs. Distributed Module Information: In the first version of the transporter, information about module membership was concentrated in objects representing the specific modules, as shown on top. This arrangement was so confusing to programmers that the transporter was redesigned to store this information in the actual slots, as annotations, as is shown on the bottom.

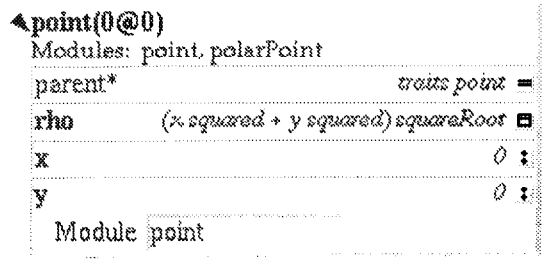

Figure 5. The module annotation in a slot: The user has exposed the annotation of the `y` slot in the prototypical point. The portion of the annotation shown here indicates that the slot belongs to module `point`. The same object contains a `rho` slot in module `polarPoint`. The module summary shown at the top of the object, shows the modules of all the object's slots, sorted by frequency.

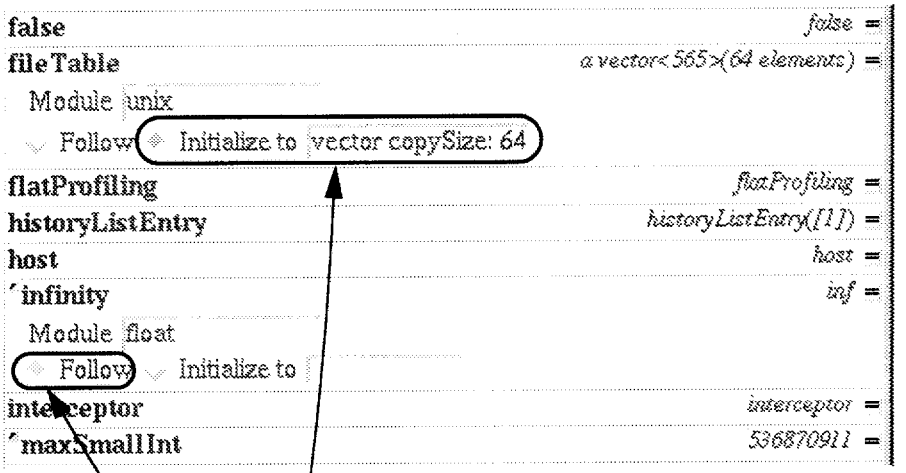

Figure 6. Actual vs. Initial Contents in the Self 4.0 programming environment: This illustration shows several slots that are globally accessible. The `infinity` slot has the `Follow` button selected in its annotation, indicating that the actual value of this slot is to be used by the transporter. On the other hand, the `fileTable:` slot's annotation has `InitializeTo` selected and the expression: `vector copySize: 64` entered in its initialization field. This annotation will ensure that the transporter writes an expression for a new, empty vector as the contents of the `fileTable` slot. If this were not done, data for currently open files would be saved.

Unfortunately, simplifying the programmer's model complicated the transporter's task. In order to write out a single source file without inspecting every slot, the transporter has to maintain a cache that maps modules (source file names) to sets of slots. And, in order to let the programmer know which modules have been changed and need to be saved, the transporter must incrementally (if conservatively) track any transitive consequences of actions in the programming environment. For example, if the user removes a slot in one module that renders part of another unreachable and therefore subject to garbage collection, the other module needs to be saved, too, in order to remove the transitively deleted information.

Both needs are met by the same mechanism: a global search that fills a cache and an incremental traversal from the point of change that (conservatively) updates the cache after every programming change. (In Self a programming change is an addition of any slot, a deletion of any slot, or a change to the contents of a constant slot, e.g. a slot holding a method.) In addition to maintaining the mapping from module to set of slots so a module can be written out, the cache also maintains information for each module about which slots have been altered since saving the module, so that a list of changed modules can be displayed. A simple change such as editing a method requires no search at all; the one slot is added to its module's dirty slot set. A more complicated change (such as altering the topology of a name space) takes longer, but still only a few seconds.

3.2 What's in a slot? Actual or Initial Contents

Once the transporter has determined that it must write out a particular slot it must construct an expression for the value of the slot. If the slot contains a method this expression is merely the source code for the method, but if the slot contains data it is not so clear what to do. A purely extensional transporter would always write out an expression for the actual contents of a slot, but sometimes the programmer intends an initial value to be written instead. Consider a slot holding a list of cached items. Although the current contents of the slot is a non-empty list, this slot needs to be initialized to an empty list when it is read in. In order to obey the programmer's intention that a slot be initialized to a counterfactual value, the slot may be annotated with an expression for its initial value. If a slot is so annotated, the transporter ignores its contents and writes out the expression instead (Figure 6).

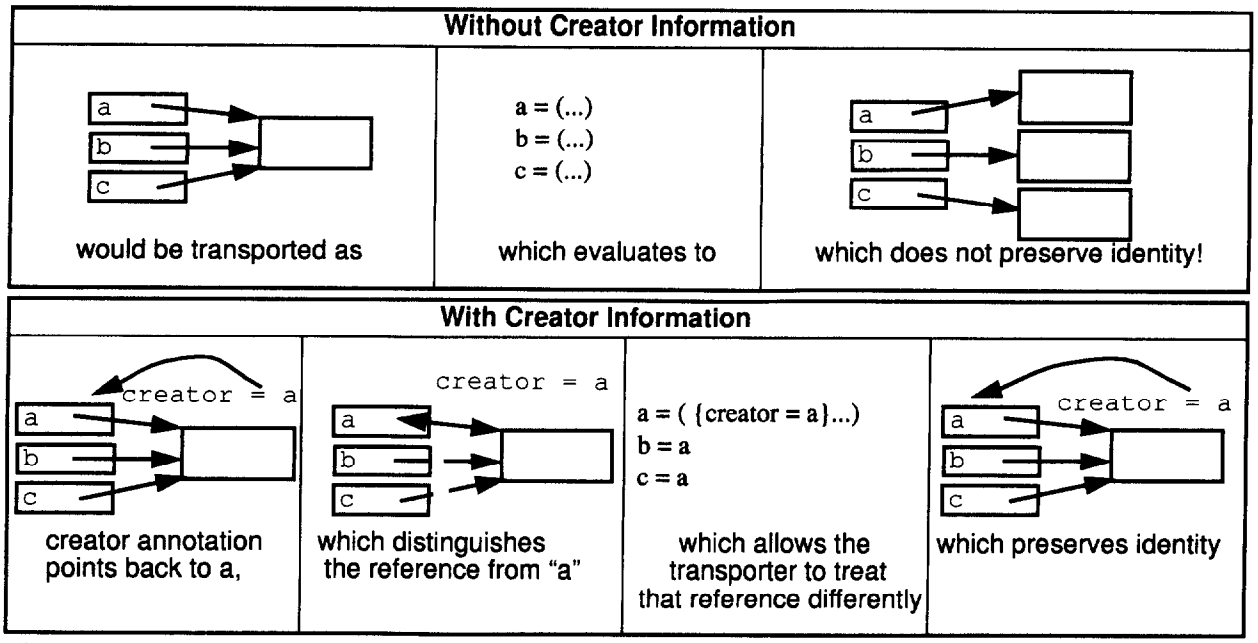

Figure 7. Creator annotations preserve identity: No matter how many other slots refer to it, an object must be created only once. The Self 4.0 transporter obeys this constraint by annotating each object with a backpointer to a single slot responsible for the object's creation. In this example, three slots in different modules all point to the same object. Without creator annotations, three different objects would result from reading the three modules. With creator annotations, one of the slots is designated as the creator, so that the three slots' relationship can be maintained.

3.3 Maintaining Identity: Reference vs. Creation

Although a slot's annotation may direct the transporter to write out its actual contents, just what that means remains open to question. Does the programmer intend for the slot to create a new object or merely to refer to some existing one? If two slots point to the same object, only one of them had better create the object; the other should be initialized to refer to the contents of the first (Figure 7).

In other words, every time the transporter follows a reference to an object, it must decide if that reference or some other one creates the object. If an object is reachable by more than one reference, there is no way to tell which one was intended to create it without additional information.

Accordingly, each object is annotated with its creator slot, a backpointer to the slot that is intended to create it. (The backpointer actually points to an object containing the slot's holder and name.) In Self the inheritance graph funnels through an object called the `lobby` that plays a role somewhat like `Class Object` in Smalltalk. Global variables are expressed in Self by putting slots in the lobby or one of its parents. Although the creator annotation only points one level back, by transitively following these annotations the transporter can find the complete path from the lobby to an object, if it exists. The transporter then uses this path to install the slot when it is read in.

The creator annotation also lets the transporter distinguish between well-known objects created by reading in source files, and clones created by running programs. In Self, a copy of the original prototypical `set` will look the same as the original, and will even have the same annotations. However, since its putative creator slot (`set`) will not point back to the clone, the transporter can tell the difference between objects it must file out and those that merely have been incidentally created. Thus, the creator information helps identify a well-known object without a costly search for all references to the object.

3.4 When Does Identity Matter?

Most objects include externally observable mutable state, and so their identities matter. For example, if two variables refer to the same stack, they are causally connected; pushing a value with one will affect the value popped from the other. Since programs rely on causal connections, the transporter must preserve them and the creator annotations described above accomplish this. However, some objects behave immutably; although such an object may cache information, there is no way to cause an externally visible side-effect upon it. For example a point object in Self behaves immutably, so that adding two points yields a new point containing the sum, rather than changing one of the old ones. For such objects, their values are more important that their identities, which are not observable anyway. (For example, points override the identity message, ==, to send equality, =.) With such objects, it is better to avoid maintaining their identity, and to just transport out a new copy for every reference. Otherwise, for example, every slot containing a particular color object would be initialized as referring to whatever slot was annotated as the creator of that color. The transporter must preserve an immutable object's value rather than its identity.

Unlike the other information discussed up to this point, externally observable mutability is associated with an object's abstract type, and follows inheritance patterns. For example all of Self's number objects (small integers, big integers, floats) are externally immutable and all of them also inherit from a common ancestor. Because of mutability's correspondence with inheritance patterns, the transporter does not use annotations to encode this property. Instead, it is encoded in an attribute, `isImmutableForFilingOut`. Objects inherit a default value of `false`, but immutable objects override this slot with `true`. Implementation aside, the important point is that whether or not an object is observably immutable cannot be effectively determined extensionally and must be supplied with extra information.

3.5 Abstract vs. Concrete Creation

Although it is always possible to create an object by concretely enumerating its slots, such a low-level expression is not acceptable when there is a more, succinct abstract expression that will do. For example, a point could be filed out as `(| x← 3. y ← 4. parent* = traits point |)` but `3@4` is far better. In addition to its conciseness and legibility, the more abstract expression is much more robust in the face of change to the implementation of points. For example, `3@4` would still work if points were reimplemented as polar, but the slot expression would not.

Unfortunately, this choice between concrete and abstract representation cannot be made by simply inspecting the objects. Each object must be asked if it is willing to supply an abstract representation and if so, what that expression is. As in the case for mutability, the abstract representation depends on the object's position in the inheritance hierarchy (its user-defined abstract type). Accordingly the transporter sends each object a message, `storeStringIfFail:`, to find such an expression if it exists.

One last piece of information is needed because of Self's prototypical model. The prototype must always be created concretely. If the prototypical point were defined as `0@0` in the source file, when reading the file the "@" method would attempt to copy the prototypical point, which would not yet exist! Thus for Self, there is also a message to identify the prototype that is needed for `storeStringIfFail:` and which should not be created abstractly (called `storeStringNeeds`). If the object cannot be created abstractly, the transporter creates a new object slot-by-slot. The availability and construction of data-type-specific abstract initialization expressions is the last piece of generally missing information about an object that must be supplied by the programmer.

3.6 "Classes" in Self: Inheriting Structure

In a minimalist language, some information that has been omitted from the language design may have to be reintroduced as intentional information for saving a program. Although the Self language includes inheritance for sharing state and behavior, it does not include any mechanism to inherit containers of state. For example, the prototypical `morph` object (a graphical element in our user interface framework) contains many slots that every morph should have, and some mechanism is needed to ensure that their presence is propagated down to more specialized morphs like the circleMorph.

In a class-based language, this need is met by a rule ensuring that subclasses include any instance variables defined in their superclasses. But in Self, a parent link inherits only state and behavior, not information about which slots are present. This omission keeps the Virtual Machine simple and increases flexibility, since a child can override an instance variable with a method. Originally, we expected to share format information with data-parents [Ungar], but never implemented dynamic inheritance efficiently enough.

Instead, slots from one prototype are automatically copied down to others by annotating the other objects with the source of the copy (copy-down parent) and a list of slots to omit from copying (Figure 8).

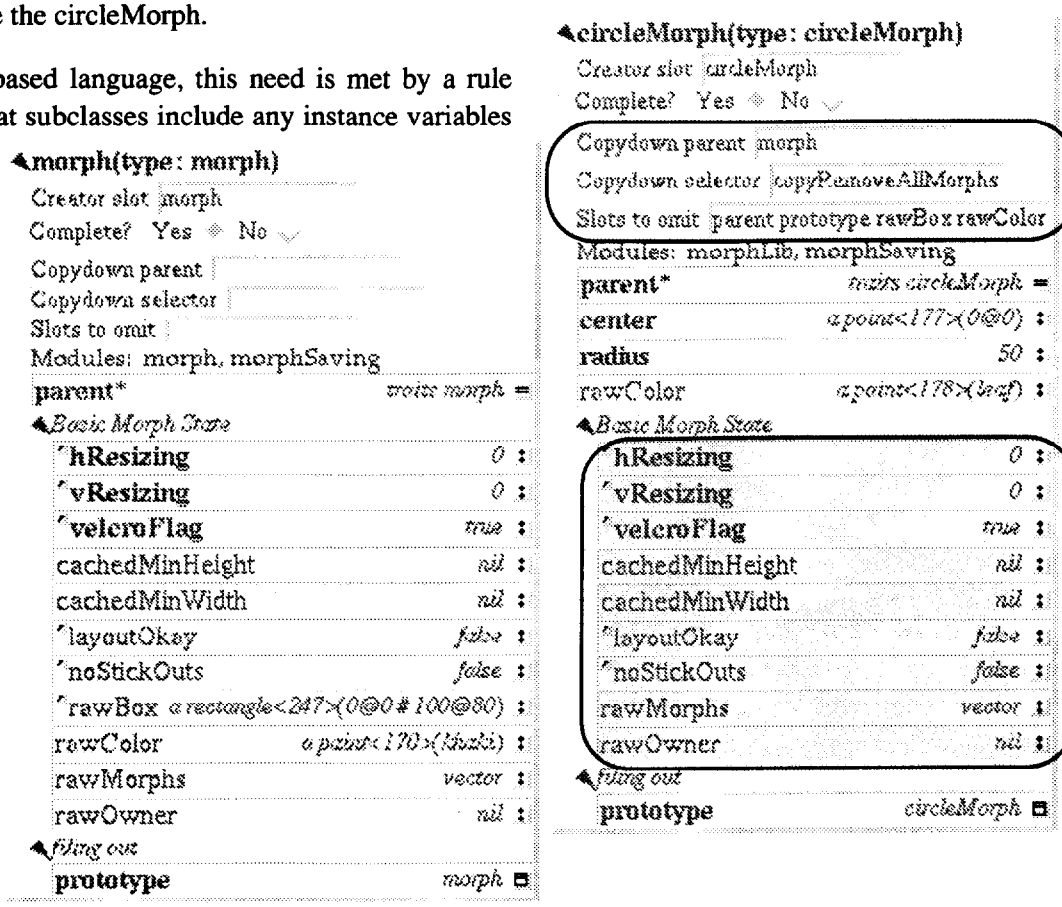

Figure 8. Copied-down slots: Two prototype objects are shown, `morph`, the general graphical object, and `circleMorph`, a more specialized object to represent circles. Since the `circleMorph` prototype needs to include the same slots that are in the `morph` prototype, it is annotated to include slots copied from morph. Although it cannot be seen on the printed page, the copied slots are shown in pink on the screen.

The `Basic Morph State` category of slots has been copied from those in `morph` by first copying the morph and removing all its submorphs (i.e. by sending it `copyRemoveAllMorphs`) and then copying the resultant slots, omitting `parent`, `prototype`, `rawBox` and `rawColor`. The three omitted slots, `parent`, `prototype`, and `rawColor`, have different contents than their counterparts in `morph` and so cannot be supported by the copy-down mechanism. The omitted slot `rawBox` is more interesting; circle morphs do not need this slot at all and so omit it. Most other object-oriented programming systems would not allow a subclass to avoid inheriting an instance variable.

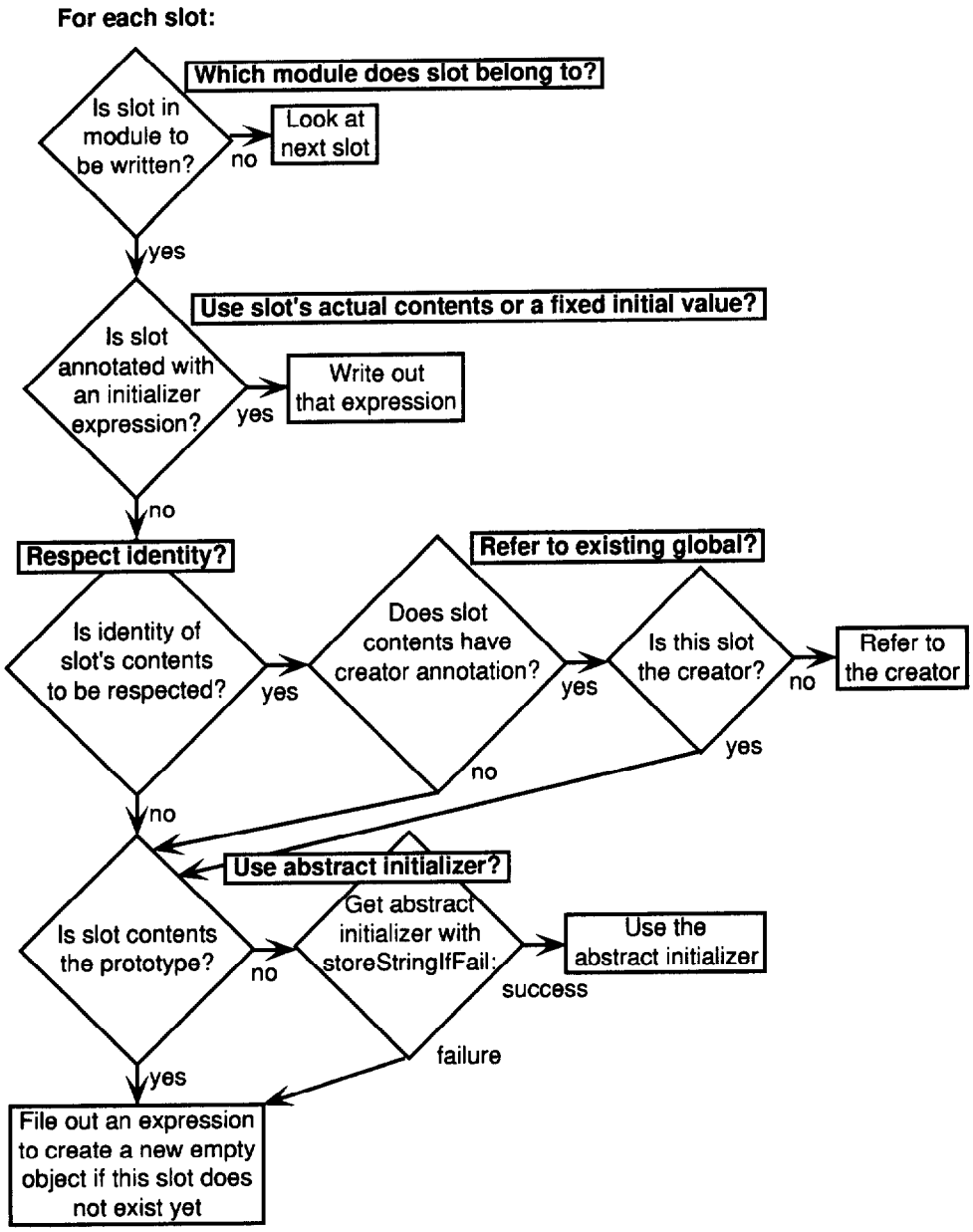

Figure 9. How the transporter uses extra information to write out a slot: This flow-chart clarifies the order in which the transporter queries the descriptive information in order to make its decisions.

The Self 4.0 programming environment uses the copy-down information to allow the programmer to use a classical style when appropriate. For example, if the programmer adds a slot to `morph` the environment will offer to add it to `circleMorph`, too.

3.7 Summary: Missing Information

Although a slot contains a simple reference to an object, the transporter must make many decisions when saving that reference for transport into another snapshot (Figure 9).

These decisions rely on information missing from the original objects which must be supplied by the programmer in annotations and extra attributes, listed earlier in Table 1. It is surprising how much there is—how much is taken for granted in conventional programs. In addition to a renewed appreciation for the information in a conventional program, these items can provide a checklist of capabilities for systems that attempt to save directly constructed programs.

total number of modules	162 modules
smallest module	6 slots
largest module	657 slots
median module	54 slots
mean module	90 slots
total slots in all modules	14,621 slots
time to save median module on a 50 Mhz. SparcStation-10	7.5 seconds
size of saved file for median module on a 50 Mhz. SparcStation-10	680 lines, 26 kilobytes
portion of object heap used for annotations	10%
portion of object heap used for module cache	4%
total size of object heap	7 MB

Table 2. Measurements of a typical Self 4.0 system

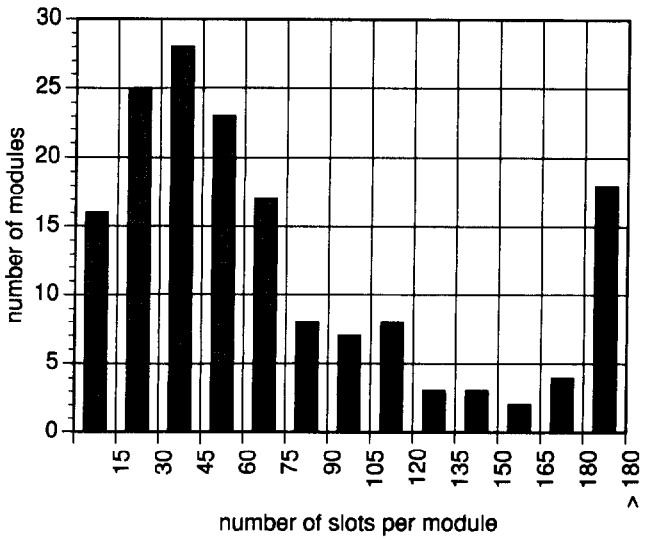

Figure 10. Module size distribution:
This graph shows the number of modules that contain a given number of slots. For example, the first bar indicates that 16 modules in the system contained from fifteen or fewer slots. (The last bar lumps together all the modules that did not fit, from 181 to 657 slots. Many of these contained automatically generated interfaces to C libraries such as xlib).

4. Status and Measurements

The Self transporter has been in daily use by the members of the Self group for approximately two and a half years, ever since we made the leap from editing files to working in the environment. Before that point two different approaches were explored, both more BOSS-like (see section 5.2) than the final design. The third version was the first to be widely used in our group. Approximately two years ago, the transporter was redesigned and implemented in its present form, and has since been in constant use.

Table 2 presents some measurements of the Self configuration most often used, and Figure 10 shows the sizes of the modules. As one might expect, most modules are fairly small, but the size distribution has a long tail. When using the system, the seven seconds required to save a typical module does not disrupt the programmer because he can go on programming in the meantime.[†]

5. Previous work

Many others have decided that classes and modules should not coincide. The designers of Beta also chose to separate modularity concerns from language design. Their fragment system [Madsen] allows a system to be decomposed into logically related, fine-grained pieces, much as the Self transporter chops information up slot-by-slot. The Beta fragment system is more versatile than the transporter, because while the transporter cannot dissect slots, a Beta fragment can be any abstract syntax tree node. For example, a Beta fragment could be a single line in a method. On the other hand, the Beta system embodies a very classical view about the nature of a program: a Beta program is a collection of abstract conceptual patterns that describe concrete phenomena without themselves partaking of the concreteness of phenomena.

[†] At this point, the reader may be wondering "What if the programmer modifies a module while it is being transported in the background? After all, Kirk always stood still." We believe we have put in enough synchronization so that the change either gets saved or not. If not, the module will remain on the dirty list.

Wills' Fresco system [Wills] partitioned Smalltalk images into verifiable units of software. Although most of his work was concerned with verification and lies outside the scope of this paper, he did independently settle on a granularity that was finer than that of a class; a capsule contained a set of instance variables, methods, theorems, and type conformance proofs. In order to model extensions to an object-oriented program, Ossher and Harrison also adopted a fine, per-slot granularity [Ossher], as has Bracha [Bracha].

Although there has been a great deal of research on persistent object systems, these systems either operate in a closed world of objects, or with objects created by conventional, descriptive programs. We are unaware of any work in this area that attempt to transport directly constructed programs between worlds.

5.1 Moving Structures between Smalltalk Images

Vegdahl moved groups of objects from one Smalltalk system to another [Vegdahl]. He discusses several issues: "mapping unique objects," which corresponds to our "reference vs. object" discussion, "mapping abstract objects" corresponding to our level of initialization section, and "mapping redefined objects", which is his attempt at ensuring that class definitions are identical so instance variables can be mapped by position. Since the goal was to move specific data structures and not programs, some of the issues dealt with for the transporter did not arise. For example, the decision to maintain identity was based on whether a reference resided in a global variable.

Although we tried to limit the number of annotations for Self by experimenting with a number of these kinds of heuristics, in the end the requirement for the transporter to handle the entire system forced us adopt the more flexible, annotation-based strategy. The Self transporter's annotations help decouple mechanism from policy; policies can be implemented by automatically setting them. Likewise, since its extra attributes are used only for transport, they could be automatically set to implement the same sort of policies that were bound in more tightly to Vegdahl's work.

5.2 ParcPlace Binary Object Storage System (BOSS)

The Self transporter owes much of its inspiration to the ParcPlace Binary Object Storage System (BOSS) [ParcPlace], although BOSS was designed to move specialized data structures from world to world, and the Self transporter was designed to be the way that every program was saved. Therefore, the Self transporter differs from BOSS in using a representation optimized for bootstrapping and textual change merging, whereas BOSS's representation is a denser binary representation which can be parsed much more efficiently. Nevertheless, as in Self, BOSS confronted the issues of whether or not to maintain identity, whether to initialize concretely or abstractly, and whether to create or reference objects.

Unlike Self, though, BOSS merged the identity and initialization issues into one concept: "manifest objects," and put the initialization-level policy into a separate `BinaryStoragePolicy` class. The standard policy detects references to objects or associations in the Smalltalk dictionary and associations visible by name to a method, classes and metaclasses. This centralization may result in less flexibility slot-by-slot than Self's creator annotations, but may result in more flexibility in that the same objects could be stored with different policy classes. Apparently, later versions of BOSS distributed some of this policy with additional attributes; a per-class method called `representBinaryOn:` could be used to gain more flexibility, although it still merged several issues into a single point of inflection [Steiger]. As compared to BOSS, the Self 4.0 transporter strives for a greater separation of concerns, produces a representation that can be used for merging changes and bootstrapping, and attempts to be more concrete and comprehensible to a wider audience.

5.3 Interlisp

The Interlisp environment [Interlisp], [Medley], and [Sybalsky], also had to save programs that were directly constructed in a world of data (sometimes called a "residential programming environment"). In

this case, data were represented as S-expressions and Loops objects (built out of S-expressions). Unlike Self's annotations, Interlisp maintained a centralized data structure to associate functions and variables with modules. But like the Self environment, changed modules were automatically added to a list of modules to be saved, and if new entities had been added, the user was queried for their module names.

It is very interesting that both systems arrived at some of the same capabilities: both Self and Interlisp allowed for a counterfactual, initial value to be stored, both supported abstract, user-defined initializers, both provide for identity preservation, although for Interlisp a special option had to be used to preserve the identity of non-Loops data that only worked within one module. (Loops objects used unique identifiers to preserve their identity.) In addition, Interlisp had a feature not included in the Self transporter: some variables could be saved with an option that would prevent them from being changed if they were already present in the destination world.

Self's annotations, its tighter integration with the programming environment (it was not so easy to go from a variable to its storage directions in Interlisp as in Self 4.0), and its more general support for identity preservation are probably the biggest differences from Interlisp. Most importantly in the context of this paper, we are unaware of any coherent framework published by creators of Interlisp for identifying the information that was added for the sole purpose of transport. For example, although Self uses separate protocols for printing and for constructing abstract initializers for transport, Interlisp used the same function, specializing its behavior by testing a global variable.

6. Conclusions

The Self 4.0 system strives for a different kind of programming experience, one based upon direct, physical manipulations of concrete objects rather than textual descriptions. Consequently, descriptions must be regenerated in order to move "programs" from one world of objects to another. The Self 4.0 transporter performs this task, and has been in daily use for two years by the Self group at Sun. It is far from perfect, but performs satisfactorily.

Although many object-oriented programming systems modularize programs by classes, we believe that a finer granularity is needed. Accordingly, the Self transporter labels each individual slot (used for both data and methods in Self) with the name of the module it belongs to. In this way a Self program can include additions to standard "classes" such as adding a method inherited by all strings, without having to modify the baseline system. An incremental cache efficiently maps modules back to sets of slots and keeps track of modified modules that need to be saved.

While building the transporter, we gradually came to a profound (and somewhat painful) realization: a directly constructed, concrete program is not complete. Although the objects comprising a program contain all the information needed to run it, they lack information needed to save and reload it into another world of objects. For example, a program including a cache may include a vector with seventeen elements in it that should be saved as an empty vector. The missing information, supplied in Self by annotations and extra attributes, illuminates the distinction between the minimal amount of information needed to run the program and the program itself. Of course, how much extra information is allowed by the framework is a trade-off between simplicity and expressiveness. We believe that the framework described in this paper represents a good compromise.

Most of the extra information expresses intentions about objects in slots. The programmer may intend for a slot to be initialized with a given value regardless of its current state, he may intend for a slot to be initialized with a reference to a particular (global) object regardless of its identity, he may intend for a slot to be initialized to an object with a given value, he may intend for an object to be represented by an abstract expression, or he may be content with simply reconstructing the object, slot-by-slot. In addition to these four cases, there are two more required by the vagaries of Self, but we believe that these basic four must be covered by any system that attempts to reconstruct programs from purely concrete, extensional information.

7. Acknowledgments

Lars Bak helped convert Self source files for the first version of the transporter, implemented the syntactic changes for annotations in the virtual machine, provided a lot of good advice on the design of the transporter, and generally helped make it happen in the time he was with the Self project. Urs Hölzle also helped set the direction for the second incarnation. All the past and present members of the Self group made essential contributions to Self 4.0, and some of them, Bay-Wei Chang, Mario Wolczko, John Maloney, Randall B. Smith, Ole Agesen, and Ole Lehrmann Madsen, even helped by reading early drafts of this paper. I would also like to thank Peter Deutsch, Guy Steele, and John Sybalsky for their help in understanding the related Interlisp work, and Chris Hibbert, Richard Steiger, Michael Van De Vanter, and Mick Jordan for their suggestions.

8. References

[Bracha] Gilad Bracha, Gary Lindstrom, *Modularity meets Inheritance*, I.E.E.E 1992 International Conference on Computer Languages, IEEE Computer Society Press, Los Alamitos, CA, 1992, pp. 282-290.

[Interlisp] *Interlisp Reference Manual*, Xerox PARC, October, 1993.

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

[Medley] *Medley Reference Manual*, Volumes 1-3, 1990, Venue Inc., Oakland, CA.

[Ossher] Harold Ossher and William Harrison, *Combination of Inheritance Hierarchies*, OOPSLA'92, pp. 25-40.

[ParcPlace] ParcPlace Systems, *Objectworks Reference Guide, Smalltalk-80, Version 2.5*, Chapter 36, ParcPlace Systems, Sunnyvale, CA, 1989.

[Smith 1] Randall B. Smith and David Ungar, *Programming as an Experience: The Inspiration for Self.* ECOOP'95, Springer Verlag.

[Smith 2] Randall B. Smith, John Maloney, and David Ungar, *The Self-4.0 User Interface: Manifesting a System-wide Vision of Concreteness, Uniformity, and Flexibility.* OOPSLA'95.

[Steiger] Richard Steiger, private communication, 1995.

[Sybalsky] John Sybalsky, private communication, 1995.

[ParcPlace] Clemens A. Szyperski, *Import is Not Inheritance, Why We Need Both: Modules and Classes*, ECOOP'92, Springer-Verlag, pp. 19-32.

[Tichy] Walter F. Tichy, *RCS—A System for Version Control*, Software Practice and Experience, 15(3), July 1985, pp. 637-654.

[Ungar] David Ungar, Craig Chambers, Bay-Wei Chang, and Urs Hölzle, *Organizing Programs Without Classes*, Journal of Lisp and Symbolic Computation, 4(3), Kluwer Academic Publishers, June, 1991.

[Wills] Alan Wills, *Capsules and types in Fresco, Program verification in Smalltalk.* ECOOP'91, Springer Verlag.

[Vegdahl] Steven R. Vegdahl, *Moving Structures between Smalltalk Images*, OOPSLA'86, pp. 466-471.