self includes: Smalltalk

Mario Wolczko
Sun Microsystems Laboratories
2550 Garcia Ave., MS UMTV29-116, Mountain View, CA 94043, U.S.A.
Mario.Wolczko@Sun.Com

1.0 Abstract

Most object-oriented languages share behavior among objects by having classes inherit from one another. However, the object-based inheritance of prototype-based languages offers greater flexibility than class-based inheritance. Using prototype-based inheritance in place of class-based inheritance does not require any loss of performance. We have constructed a Smalltalk system in Self that demonstrates this.

2.0 Introduction

Traditionally, object-oriented languages have shared behavior among objects by using classes related by inheritance. More recently, an alternative organization has been proposed based on prototypes [L86, US87]. In a prototype-based language, sharing is achieved by having objects directly inherit from each other. In principle, any object can inherit from any other; there are no special objects playing the role of classes.

It has been claimed that the prototype-based approach is more flexible than the class-based approach: prototypes can be used to model classes directly and efficiently, whereas the converse is not true. In this paper we demonstrate that prototypes can indeed model classes directly and efficiently, by describing an implementation of a class-based language, Smalltalk, in a prototype-based language, Self. The implementation of classes is direct, and the resulting Smalltalk implementation runs considerably faster than other Smalltalk implementations running on similar hardware, proving that there is no inherent performance disadvantage to this approach.

3.0 An overview of Self

Self [US87] is a prototype-based language. All computation in Self involves objects. A Self object is composed of a number of named slots. Each slot contains a reference to an object; methods are special objects that can be executed. When a message is sent to an object, the slot in the object with the same name as the message is located. If the slot refers to a method, the method is executed. Otherwise, the object referenced by the slot is returned. A slot can be designated as a parent slot (by appending an asterisk to its name), in which case message lookup will continue to the object referenced by the slot if no local

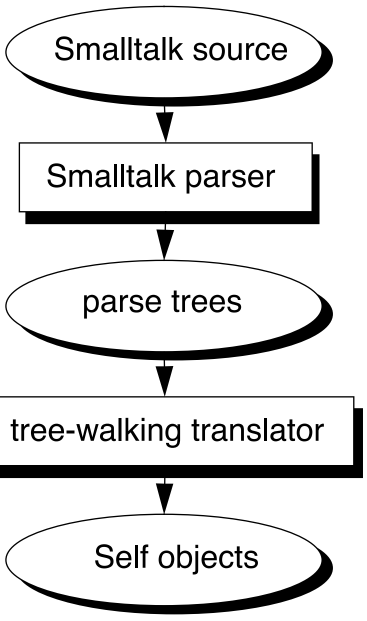


Figure 1. The structure of the implementation of Smalltalk in Self

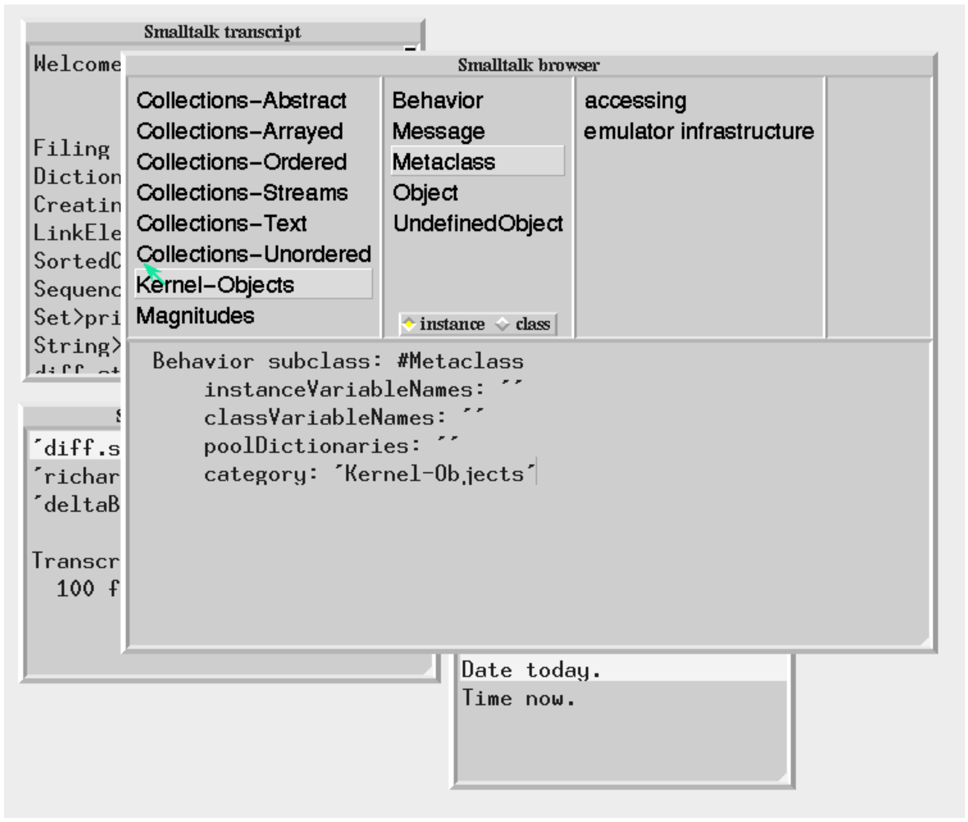
slot is found matching a message. Parent slots are the realization of object-based inheritance.

4.0 A Smalltalk implementation in Self

We have constructed an implementation of Smalltalk in Self. The implementation is written entirely in Self, and did not require any changes to the Self Virtual Machine. The structure of the implementation is sketched in Figure 1.

The implementation includes a Smalltalk parser generated by the Self parser-generator, Mango [A94], and a translator which generates equivalent Self code from the Smalltalk parse trees. The translated code runs in an environment of Self objects which mimic Smalltalk classes.

Additionally, there is a rudimentary development environment, comprised of browsers, workspaces, inspectors and a transcript. These were constructed in Self, and provide the usual facilities of browsing, editing and running code.



```
Welcome

Filing
Diction
Creatin
LinkEle
SortedC
Sequenc
Set>pri
String>
```

```
'diff.s
'richar
'deltaB

Transcr
  100 f
```

```
Collections-Abstract       Behavior          accessing
Collections-Arrayed        Message           emulator infrastructure
Collections-Ordered        Metaclass
Collections-Streams        Object
Collections-Text           UndefinedObject
Collections-Unordered
Kernel-Objects
Magnitudes
```

```
Behavior subclass: #Metaclass
    instanceVariableNames: ''
    classVariableNames: ''
    poolDictionaries: ''
    category: 'Kernel-Objects'
```

```
Date today.
Time now.
```

Figure 2. The Smalltalk-in-Self programming environment

4.1 Semantic issues

In order to mimic Smalltalk execution in Self, the following issues had to be addressed:

- The choice of representation of the various kinds of Smalltalk objects, including classes, metaclasses, instances, primitive objects (integers, floats, characters, etc.).
- The mapping from Smalltalk expressions to Self expressions.

4.1.1 Representing classes, metaclasses and instances

To mimic Smalltalk execution in Self, an organization of Self objects has to be devised that will model all the essential properties of Smalltalk's classes, metaclasses and instances. The Smalltalk Virtual Machine defines some of the relationships between these objects (e.g., that an instance knows about its class, that a class knows about its superclass, about the format of its instances, and has a reference to a method dictionary). Other relationships are mandated by the organization of objects in the Smalltalk world (e.g., the relationships between classes and metaclasses), but are considered part of the definition of Smalltalk.

In contrast, Self imposes minimal constraints on the relationships between objects. An object need not possess any parent slots, in which case it does not inherit from any other object. Or, it may possess one parent slot, or even multiple parent slots, providing a form of multiple inheritance. We will describe an organization of Self objects which mimics the behavior of Smalltalk classes.

The fundamental properties of a Smalltalk class are:

- A class inherits from another class, known as its superclass. A single class, known as Object, does not have a superclass. Hence, the classes form an inheritance hierarchy, rooted at Object. We will call the set of classes that a class inherits from its *ancestors*, and the classes which inherit from a class its *descendants*.
- A class, together with its ancestors, defines the structure of its instances. Each class declares a set of named instance variables. None of these names can be the same as the name of an inherited instance variable. An instance of a class contains the instance variables declared by that class and its ancestors.
- A class defines methods applicable to its instances, in addition to those inherited from its ancestors. A method can override an inherited method.
- Each class has a corresponding metaclass; the metaclass defines the structure and behavior of the class in a manner analogous to the way a class defines the structure and behavior of its instances. The metaclass hierarchy parallels the class hierarchy, except that Object's metaclass has a superclass (a class named Class, which is a descendant of a class named Behavior).
- A class's behavior is defined by its class methods, in addition to class methods inherited from its ancestors. Because of the relationship between a class and its metaclass, the class methods of a class are the instance methods of its metaclass.
- A class can declare a set of named class variables. None of these names can be the same as the name of an inherited class variable. All the methods of a class and its metaclass and its descendants and their metaclasses can access the class variables.
- An instance of a class can be created using a primitive operation, bound to the instance methods basicNew and new in Behavior.

```
class               Date
superclass          Object
instance variables  days
instance methods
addDays: d
  ...
class methods
initialize
  ...
```

Figure 3. The (partial) definition of the Smalltalk class Date

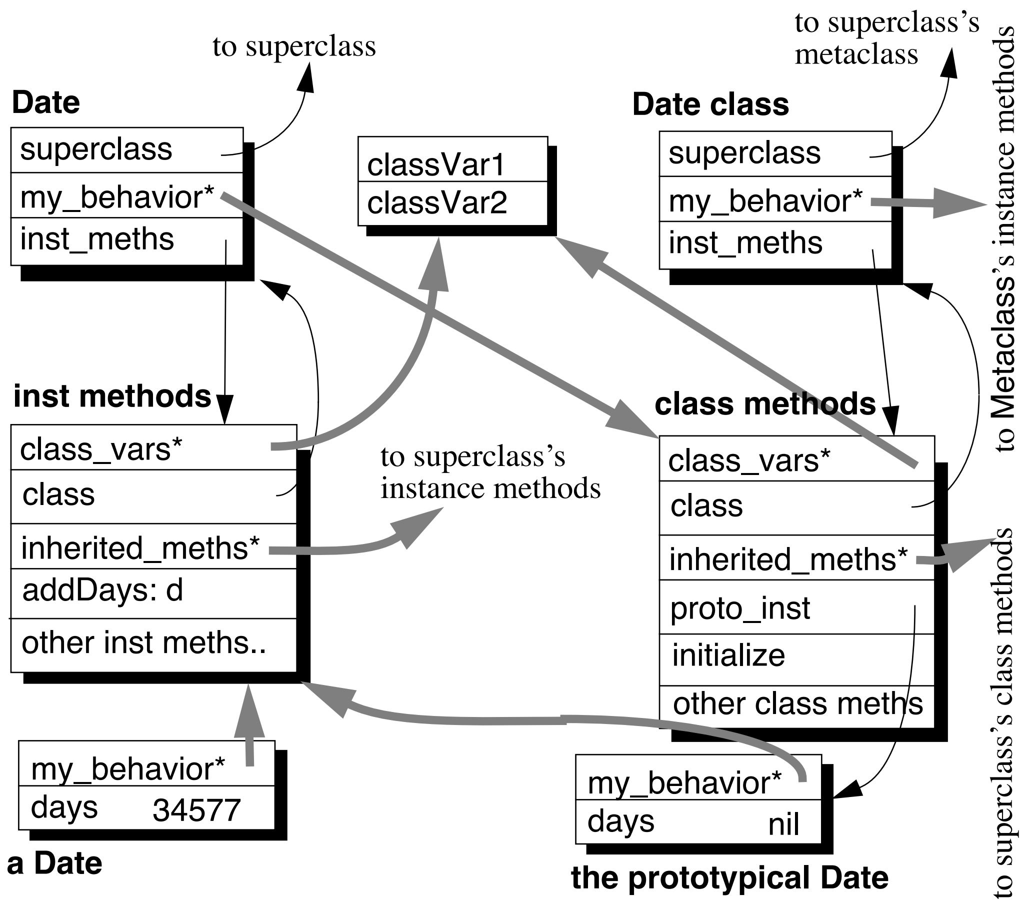


Figure 4. The Self objects representing class Date and an instance

4.1.2 The Self representation of a Smalltalk class

In our implementation, each Smalltalk class is represented by a cooperating group of Self objects (see Figure 4, in which inheritance links are grey; the corresponding Smalltalk class definition is shown in Figure 3). Each Smalltalk instance (e.g., the instance of Date, at the lower left of Figure 4) contains an inheritance link, my_behavior, which refers to the *instance method dictionary* of that class (the object labelled “inst methods” in the figure), which holds the instance methods (addDays:, etc.).

The instance method dictionary also contains these slots:

1. class, which refers to an object that is the class’s *proxy.*
2. A parent slot, inherited_meths, which refers to the superclass’s method dictionary,
3. A parent slot, class_vars, which refers to an object that contains the class variables declared by this class.

The class proxy (at the upper left of Fig. 4) is the actual object which denotes the class during Smalltalk execution, i.e., is the receiver of messages sent to Date. This object has a reference to its superclass in the superclass slot. The remainder of its behavior is defined by the class methods inherited through its my_behavior slot, which refers to a *class method dictionary* similar in structure to the instance method dictionary. In addition to the slots present in an instance method dictionary, the class method dictionary contains a reference to a *prototypical instance* of that class. The prototypical instance has the structure (i.e., the instance variables) required of instances of the class, suitably initialized so that a new instance can be created by simply cloning this object. Note that the class methods also inherit from the class variable pool (upper center), so that class variables are accessible to both instance and class methods.

The metaclass proxy (upper left in the diagram, Date class) is identical in structure to the class proxy.

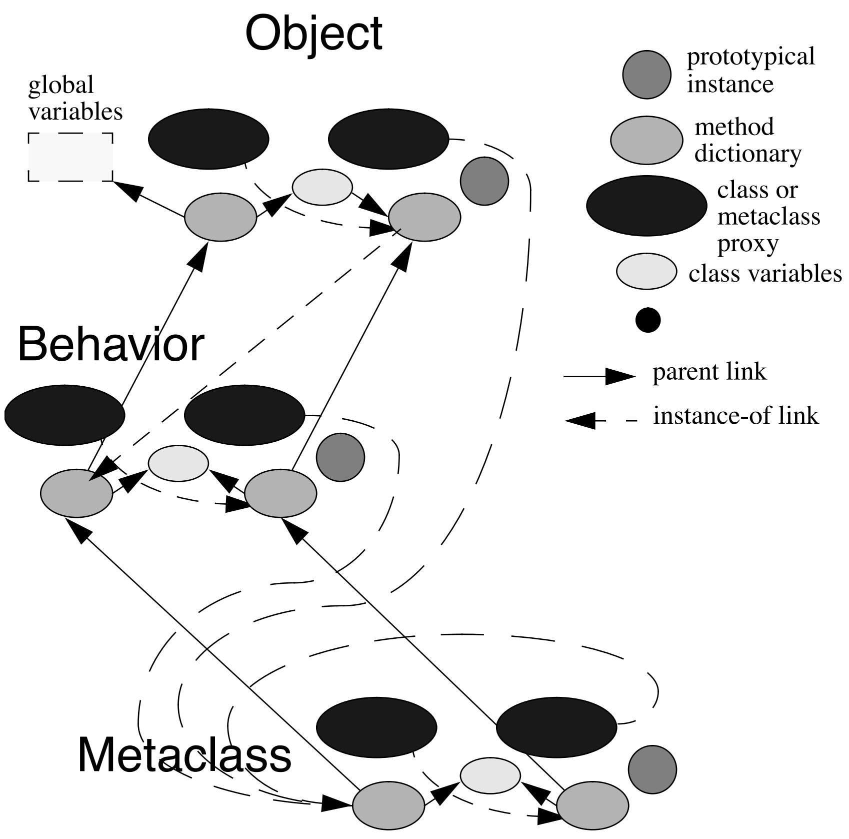


Figure 5. How the basic classes are connected

This constellation of objects is repeated for each class in the system (see figure above). The five outgoing links to the superclass (to the superclass's proxy, the superclass's metaclass's proxy, the superclass's instance methods, the superclass's class methods) are connected in the expected way. The inherited instance methods of the metaclass are the instance methods defined by the class Metaclass. The instance method dictionary of Object has an extra parent link, to the object which is the repository of all Smalltalk global variables.

4.1.3 Translating Smalltalk expressions to Self

Translating Smalltalk source expressions into Self is relatively straightforward; the syntax of Self was heavily influenced by Smalltalk, and bears many similarities. The particular variant of Smalltalk we chose to handle was that of ParcPlace's Objectworks/Smalltalk 4.1. This is arguably the most sophisticated version of the language; its use also made comparison of programs between systems simple, as minimal source changes were required (these are discussed later).

Translation of a message send is complicated only by Self's case convention on keyword messages: all but the first keyword of a keyword selector must begin with an uppercase letter, so that at:put: is translated to at:Put:. (By convention, the first letter of a Smalltalk keyword is lower case; strictly, we should check for this.)

The reverse problem is encountered when dealing with Smalltalk class and global variables. These begin with an uppercase letter, which is not permitted in Self. Instead, we append the suffix _g; the underscore is not allowed in Smalltalk identifiers, so we can be sure that the mangled name will not collide with an existing name. (We also exploit this by having all support methods include an underscore in their names, so that they do not accidentally collide with Smalltalk methods of the same name.)

The names of arguments and temporary variables are left intact. We need to mangle the names of instance variables, because in Self access to an instance variable is via a message. In Smalltalk, it is possible to have an instance variable x and a method x in the same class. The method typically returns the value of x, but it need not; in this case we need to mangle the name of the instance variable when translating to Self. For simplicity, we append the suffix _i to all instance variables when translating. When translating an assignment to an instance variable, x, we must generate a message send which will cause the Self assignment primitive associated with x to be invoked, namely, x_i:.

Smalltalk has a rich repertoire of syntax for literal objects, i.e., objects which are constructed at compile time. In ParcPlace Smalltalk, there are various kinds of integers, floating point numbers, characters, strings, symbols, object and byte arrays, and blocks.

Translation of integers and single-precision floating point numbers was straightforward. Self 4.0 does not support double-precision floating point numbers, so we chose to ignore them; none of the programs we tried to run used them anyhow.

In translating characters, strings, symbols and arrays, none of which has a literal syntax in Self[1], we had to ensure that the appropriate objects were created at compile time. To do this, we utilized the syntax in Self that declares a temporary variable and initializes its value at compile time. Hence, each literal requires an extra temporary to be synthesized in the translated code.

Self has a block facility very similar to Smalltalk's, so the translation of blocks was, for the most part, straightforward. The only difficult problem is presented by Self's treatment of long-lived blocks. In Smalltalk, a block can be executed time at any time after its creation, even after its surrounding context has exited; such a block is known as a *non-lifo block*. In contrast, Self does not allow a block to begin execution if its surrounding context has exited. To make this possible would have required modification to the Self Virtual Machine, which was beyond the scope of this project and deemed inessential to the purpose of the experiment.

Unfortunately, non-lifo blocks are used quite commonly in Smalltalk programs, and so we had to devise a solution to allow us to complete our experiment and compare the perfor-

1. Self has literal strings, but we chose not to use Self strings to represent Smalltalk strings; the reason for this appears in Section 4.1.4

```
nonLifoBlockExample                nonLifoBlockExample
  | c t |                            | c |
  c := SortedCollection new.         c := SortedCollection new.
  c sortBlock: [ :a :b |             c sortBlock: [ :a :b || t |
     t := a + 1.                        t := a + 1.
     t > b ].                           t > b ] asNonLifoBlock.
  ^c                                 ^c
```

Figure 6. How a non-lifo block must be rewritten.

mance of a variety of Smalltalk programs on our implementation with a native Smalltalk system. The solution adopted was to have the programmer annotate such blocks, and then have the Smalltalk-to-Self translator deal with them differently.

The annotation takes the form of a unary message, asNonLifoBlock, which must be sent to the literal block before anything else happens (see Figure 6). The translator detects this annotation, and instead of translating the Smalltalk block into a Self block, it creates a normal object with a suitable value method whose body is the translated body of the block. In order for the block object to be able to access self and its instance variables, when the method is executed a parent slot in the block object is set to the value of self. The block does not have access to arguments and temporaries in enclosing scopes, so some source reorganization may be necessary.

Because these non-lifo blocks use a dynamic parent slot, the Self compilation system is unable to optimize the machine code for them as well as it can for regular blocks, and so they suffer a small performance penalty.

```
a := b := c                        a: ([| :a_t | b: a_t. a_t]
                                       value: c)

a foo ; bar                        [| :c_t | c_t foo. c_t bar]
                                       value: a
```

Figure 7. Translating multiple assignments and cascades

Smalltalk has syntactic support for multiple assignments and cascaded messages; Self does not. To translate these, extra blocks are introduced (Fig. 7). Self's inlining compiler will optimize these blocks away if the expressions are used frequently.

4.1.4 Primitives and primitive types

In Smalltalk, a primitive method contains an integer that identifies an offset into an entry point table in the Virtual Machine. That entry point implements the primitive. In our implementation of Smalltalk, the primitives are written directly in Self. Where Smalltalk requires an integer, in our system you may write a string containing a Self expression. This string will be suffixed with the rest of the method, translated into Self and enclosed in a block, which may be conveniently used as a Self fail block. For example, the Smalltalk method

```
basicSize
  <primitive: '_Size'>
  self primitiveFailed
```

is translated into

```
basicSize = (_SizeIfFail: [self primitiveFailed])
```

Messages may also be sent between Self and Smalltalk objects. To escape the Smalltalk world, a method, `selfLobby`, is provided in `Object`, which return a reference to a key Self objects, the “lobby”.

A variety of Smalltalk objects have analogous representations in the Self world. For each kind, we had to decide whether to use the Self representation, or devise a new one. The benefit of reusing the Self representation is ease of implementation, and, sometimes, better performance. On the other hand, the Self objects are already endowed with behavior which may not be compatible with the expected behavior as Smalltalk objects.

For integers, floats, and the boolean objects, true and false, we chose to use the Self representation. This decision was motivated by efficiency: 30-bit integers and single-precision floats are dealt with specially by the Self Virtual Machine, and making new objects to represent them (e.g, “boxing” an integer in another object) would have made arithmetic much slower. Self’s implementation of integers provides aritrary precision arithmetic in a similar way to Smalltalk’s Integer classes, so that eased our task. As for the booleans, these are known to the Self Virtual Machine so that it can return one or the other directly as the result of relational operators. Hence, when we evaluate x<y, if x are y are integers, the result would be Self’s true.

The biggest problem in using Self’s representation is that the Self system already defines many methods for these objects, and some are incompatible with the expected Smalltalk behavior. For example, /, sent to an integer with an integer argument, returns the truncated quotient in Self, but in Smalltalk returns a Fraction. Similarly, printString returns a Self string describing the receiver, but for Smalltalk execution should return a Smalltalk string.

Another problem is that these Self objects have constrained inheritance relationships. For example, although one can add and remove behavior from the parent of all floats, you cannot change what the parent slot of a float actually refers to. This makes participation in the class hierarchy more difficult. In our system, we chose not to have these objects participate in the Smalltalk class hierarchy (instead we added appropriate hand-written methods into their traits), although we do not believe that such participation would be very difficult to achieve. The solution is to add a class method into traits smallInt, traits float, true and false, referring to an object of the appropriate structure, and add another parent to these objects, referring to the instance method dictionary.

For the remaining Smalltalk classes for which similar Self objects were available, we chose to construct our own representation, to avoid the problems described above. Hence, the representation of Smalltalk's nil is distinct from Self's nil, Smalltalk Strings are distinct from Self mutableStrings, and characters are Self objects containing an ASCII value. Symbols contain an instance variable which refers to a Self canonicalString, which allows us to reuse the hashing and interning mechanism built into the Self Virtual Machine. Smalltalk Arrays and ByteArrays, and other classes which utilize indexed instance variables, are not related to Self's vectors.

Some of the primitive methods are worthy of mention. The become: primitive, used in Smalltalk-80 to exchange object identity, was implemented using Self's _Define: primitive (sometimes known as a "one way become"). The GNU Smalltalk library invokes the become: primitive a lot, mainly to grow collections. This is inexpensive when objects are reached indirectly through an object table, as in the GNU Smalltalk Virtual Machine, but is prohibitively expensive when objects are addressed directly, as exchanging identity requires a traversal of the complete object heap. When we first ran the DeltaBlue benchmark (see below), it was very slow, due to frequent invocation of become:, so we selectively reimplemented the collection classes from GNU Smalltalk to avoid it.

Smalltalk represents method and block activations as objects, creating instances of MethodContext and BlockContext. The currently executing activation is accessible via the pseudo-variable thisContext. This representation of contexts is expensive and complicated to maintain, and is not supported by Self (which instead allow one to query the state of an activation but not to change it), so we decided not to support this feature. None of the programs we tried to run used it.

4.2 Building a Smalltalk programming environment in Self

To make the Smalltalk system more attractive to students and educators, we decided to build a rudimentary programming environment for it, using the capabilities of the Self user interface substrate, Morphic [SMU95], and modeled on traditional Smalltalk development tools [G84]. This approach was chosen in preference to building the environment in Smalltalk because

1. No freely available Smalltalk source for browsers, etc., is available, and
2. There is no widely accepted and well defined primitive interface for graphic primitives in Smalltalk.

Constructing the environment in Self had further benefits:

- Because the environment is external to the Smalltalk class hierarchy, it is possible to take a Smalltalk application and easily separate it from the development tools, a task that has traditionally been difficult in monolithic Smalltalk development systems.
- The Self user interface substrate, Morphic, has some unique and useful properties:

(a) All user interface components can be shared among multiple users on separate workstations, and each can interact with them concurrently. This is useful in a teaching situation, where an instructor may wish to demonstrate the system to a student, or students may choose to collaborate on a project.

(b) The components of the user interface can be composed and decomposed graphically and directly, allowing new user interfaces to be constructed simply. For example, a system browser can be transformed into a class browser by simply extracting the category pane and class list.

5.0 Performance

To assess the performance of Smalltalk code in this system, we took three medium-sized benchmarks and measured their runtimes in this system and in ParcPlace's ObjectWorks/Smalltalk 4.1, running on the same hardware (a Sun SPARCstation-10).

The selection of benchmarks was limited by incompatibilities in class libraries; our implementation had no graphics classes, and so any benchmark had to be entirely non-graphical.

The three benchmarks we used were:

1. Richards, an operating system simulator. This benchmark has been widely used in previous implementation experiments. It uses a very small number of library classes and methods, and therefore is a reasonable measure of language performance independent of class library implementations.
2. Deltablue, a constraint solver, with two separate tests. Deltablue uses a large number of Smalltalk classes, and is a reasonable test of the fidelity of execution of the Smalltalk language and class libraries. However, in comparing its performance, one must consider that the implementation of the collection classes might differ between libraries.
3. Diff, a program for computing longest common subsequences. This is the core algorithm in some versions of the "diff" program made popular by Unix. It uses a small number of collection classes quite heavily.

In measuring Self programs, one has to be cognizant of Self's dynamic optimization system. This system instruments the initial version of a compiled method, and uses the measurements to recompile hot spots with a much more sophisticated optimizing compiler. Hence, the performance of a program is not constant, but improves asymptotically. The results we quote here are based on the best of 20 runs, which gives the optimizing compiler a chance to generate code close to its optimum. ParcPlace Smalltalk is based on a dynamic translator; the first run of a program will be slower due to translation taking place, but subsequent runs will be very similar (as was observed). We took the best of 20 runs, just to make sure.

Table 1 lists the results. As can be seen, all benchmarks run faster in the Self system, some significantly more so.

Benchmark	ParcPlace Smalltalk	Smalltalk in Self	Speedup
Richards	1100	410	2.7
Deltablue chain	1200	830	1.4
Deltablue projection	980	450	2.2
Diff	7500	7000	1.1

TABLE 3. Performance results (times in ms, all results to two significant figures)

Richards shows the Self compilation system in its best light. Self's compiler optimizes and re-optimizes code adaptively, based on the observed types of message receivers. Very little of the GNU class library is used, and so the performance can be directly compared.

The Deltablue tests spend a significant part of their execution in the Smalltalk collection classes. The GNU classes are not implemented in the same way as the ParcPlace classes, and therefore these times are hard to compare.

The Diff program spends most of its time in a single loop, doing binary-chopping searches down a list of sorted integers. While the Self compiler discovers this loop, and inlines all the blocks involved, in Smalltalk this loop is written in terms of messages such as ifTrue: and whileTrue: which the compiler treats specially and inlines anyway. The machine code generated for these loops is similar in quality in both systems, and hence the times are similar.

6.0 Future work

We feel that the system demonstrates that a class-based language, Smalltalk, can be emulated in a prototype-based language, Self, with good fidelity and performance. However, there are a few limitations which would preclude its use as an industrial-strength implementation. Foremost of these is the poor support for non-local blocks, which would require some changes to the Self Virtual Machine.

While the performance of translated code is acceptable, the translation itself is not especially fast. It could be improved by going directly from Smalltalk to Self bytecodes, bypassing the intermediate stage of Self source.

7.0 Conclusions

We have shown that a class-based language, Smalltalk, can be implemented in a direct and straightforward way in a prototype-based language, Self, with no loss of fidelity, generality or performance.

The entire Smalltalk implementation described in this paper was constructed by the author in approximately three weeks. The ease of construction is a testament to Self's incremental programming environment and user interface construction facilities.

8.0 Availability

The Smalltalk implementation described in this paper is part of the Self 4.0 release, available from `http://self.sunlabs.com`.

9.0 Acknowledgments

The author would like to thank Ole Agesen, Ivan Moore, Randy Smith, Dave Ungar and Phillip Yelland for comments on earlier versions of this paper. John Maloney helped construct the initial implementation of our Smalltalk browser. Thanks also to Steve Byrne for creating the GNU Smalltalk class library and making it freely available.

The task of designing the emulation system was made much easier by the author's involvement, while at the University of Manchester, in various Smalltalk translation projects. I am indebted to Borek Vokach-Brodsky, Neil Cope and Ivan Moore for teaching me how to deal with the subtleties of translating Smalltalk to other languages. Their work is described in [VBW90] and [MWH94].

10.0 References

[A94] Ole Agesen, Mango: *A Parser Generator for SELF*, Technical Report, Sun Microsystems Labs, SMLI TR-94-27, 1994.

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

[G84] Adele Goldberg, *Smalltalk-80: The Interactive Programming Environment*, Addison-Wesley, 1984.

[H95] Urs Hölzle, *Adaptive Optimization for SELF: Reconciling High Performance with Exploratory Programming*, Technical Report, Sun Microsystems Labs, SMLI TR-95-35.

[L86] Henry Lieberman, *Using Prototypical Objects to Implement Shared Behavior in Object Oriented Systems*, Proc. OOPSLA 1986, Portland, Oregon, Oct. 1986, pp. 214–223.

[MWH94] Ivan Moore, Mario Wolczko, Trevor Hopkins, *Babel – A Translator from Smalltalk in to CLOS*, Proc. TOOLS USA, Santa Barbara, CA, Aug. 1994.

[SMU95] Randall B. Smith, John Maloney, David Ungar, T*he Self-4.0 User Interface: Manifesting a System-Wide Vision of Concreteness, Uniformity and Flexibility*, Proc. OOPSLA 1995, Austin, Texas, Oct. 1995.

[US87] David Ungar, Randall B. Smith, *Self: The Power of Simplicity*, Proc. OOPSLA 1987, Orlando, Florida, Oct. 1987, pp. 227–242.

[VBW90] Borek R. B. Vokach-Brodsky, Mario Wolczko, *Smalltalk Application Compilers*, Proc. TOOLS Pacific, Newcastle, Australia, Dec. 1990, pp. 69–78.