Reflective Facilities in Smalltalk-80

Brian Foote
Ralph E. Johnson

University of Illinois at Urbana-Champaign
1304 W. Springfield
Urbana, IL 61801

foote@p.cs.uiuc.edu
(217) 333-3411

johnson@p.cs.uiuc.edu
(217) 244-0093

Reflective Facilities in Smalltalk-80

Brian Foote
Ralph E. Johnson

Dept. of Computer Science
University of Illinois at Urbana-Champaign
1304 W. Springfield
Urbana, IL 61801

foote@p.cs.uiuc.edu (217) 333-3411
johnson@p.cs.uiuc.edu (217) 244-0093

Abstract

Computational reflection makes it easy to solve problems that are otherwise difficult to address in Smalltalk-80, such as the construction of monitors, distributed objects, and futures, and can allow experimentation with new inheritance, delegation, and protection schemes. Full reflection is expensive to implement. However, the ability to override method lookup can bring much of the power of reflection to languages like Smalltalk-80 at no cost in efficiency.

Introduction

One of the attractions of object-oriented languages is their extensibility. Programmers gain confidence that facilities that are not already available in a given environment can easily be constructed. Current systems make it easy to define data types like complex or arbitrary precision numbers. New object types become as "first class" as those built into the language. A big part of the appeal of object-oriented programming is that programmers believe that, if they really wanted to, they could redefine the world.

However, there are a few kinds of objects that are difficult to implement in conventional object-oriented programming languages such as Smalltalk-80 [Goldberg 1983] and C++ [Stroustrup 1986]. For example, a future object [Halstead 1985] acts as a surrogate for a result that is in the process of being computed. Messages to the future cause the sender to wait until the result is computed, at which time the message is relayed to the result. Object-oriented languages make it easy to redefine simple methods, but futures must intercept any message sent them. Monitors, atomic objects, and distributed objects present similar difficulties. These applications all require that an object be able to intercept any message sent to it on a per-object rather than a per method-basis. In Smalltalk-80, the method dispatch mechanism is buried within the Virtual Machine, beyond the programmer's grasp.

Recent work on object-oriented computational reflection [Maes 1987b] has provided a general framework for addressing problems like those discussed above. Reflection is intriguing because it seems to allow programmers to make open-ended, localized extensions to the languages and systems with which they are working.

Reflective systems are frequently constructed using metacircular interpreters. This approach provides the highest possible degree of flexibility at the expense of performance. We believe that full reflection may not be necessary to address many of the kinds of problems that have attracted so much attention to it. For this reason, we have focused on how a handful of *reflective facilities* can efficiently bring much of the power of full reflection to Smalltalk-80. In particular, this paper discusses redefining the default method lookup mechanism, and looks at how lightweight classes can be used to implement *metaobjects*.

Reflection

A reflective program is one that reasons about itself. A fully reflective procedural architecture [Smith 1983] is one in which a process can access and manipulate a full, explicit, *causally connected* representation of its own state. "Causally connected" means that any changes made to a process's self-representation are immediately reflected in its actual state and behavior.

[Maes 1987a] [Maes 1987b] describes an object-oriented reflective language called 3-KRS. 3-KRS partitions every object into a domain specific *referent* and a reflective *metaobject*. The referent contains information describing the real-world entity that the object represents. The metaobject contains information describing its referent as a computational entity in itself. Enforcing this separation encourages the emergence of standard protocols for communicating between objects and metaobjects. The distribution of the system's self-representation among the system's metaobjects makes this self-representation easier to think about and manipulate.

Manipulation of an object's metaobject can affect the way in which computation for its referent object is carried out. For instance, in 3-KRS, an object can have method lookup redefined by giving it a new metaobject that defined a "deviating" interpreter. Such changes can be made on a per-object basis, since each object has its own metaobject. Specialized metaobjects can be introduced dynamically. The ability to introduce temporary, localized changes to an object's semantics without changing the object itself accounts for a great deal of reflection's power.

Maes distinguishes between systems with *reflective facilities* and full-fledged *reflective architectures.* Reflective facilities allow a program to reflect upon certain aspects of itself. For example, in Lisp, **eval** permits reflecting upon programs given as data, and **catch** and **throw** allow reflecting upon the runtime stack.

Actor Languages

An Actor [Agha] is a process that explicitly decides whether and how to handle or delegate any messages it receives. This ability is used to construct futures, monitors, etc. Actor languages are designed for multiprocessors; the requirement that each actor be a separate process makes actors inefficient on single processors. Further, using a parallel programming language for sequential programming causes a certain mental overhead.

Smalltalk-72 [Goldberg 1976] was similar in some respects to the Actor languages. In Smalltalk-72, an object's class was a process that repeatedly examined the object's message stream and then dispatched on the message. As the language evolved, message lookup was subsumed into the Smalltalk Virtual Machine for efficiency's sake and because deviations from case-style method dispatching were rare and usually hard to understand. Smalltalk-72 permitted more reflection than Smalltalk-80 because objects could manipulate the message stream.

Replacing user-defined message-handling with standardized method lookup was a success. It led to efficient Smalltalk implementations and a large body of reusable software. However, the power of the Actor language is needed to deal with parallelism and other aspects of modern applications. We believe that object-oriented systems should provide primitive facilities that allow actor-like objects to be constructed out of more rudimentary components. Thus, the full power of actors will only be used when appropriate.

Lisp Based Systems

The Common Lisp Object System (CLOS) [Bobrow 1988] [Keene 1989] is a marriage of Common Loops and New Flavors that provides a very powerful generic function-based method combination mechanism. The CLOS Metaobject Protocol provides a metacircular definition of the entire CLOS system. The MOP is intended to permit open-ended experimentation with new object-oriented programming paradigms, and has been used to successfully implement a number of distinctly un-CLOS-like object-oriented programming systems. The MOP is designed to permit the modification of basic language mechanisms such as method lookup from with CLOS itself.

The mechanisms provided in CLOS's standard method combination scheme are quite powerful. CLOS generic functions can specialize on any of their arguments, not just the first, and may specialize on instances as well as classes and types. The **:before**, **:after**, and **:around** method types modify particular primary methods. Class-wide before, after and around methods for arbitrary messages cannot be defined without making alterations at the meta-level.

Cointe's ObjVLisp system [Cointe 1987] permits a Metaclass hierarchy distinct from the class hierarchy. This allows arbitrarily deep meta-regress, where appropriate. This system allows the lookup method used by the system's send primitive to be changed by manipulating the metaclass. Metaclass customization can also be used to control internal object representations, method caching, and access to instance variables.

Reflecting on the Smalltalk Virtual Machine

Most of the existing object-oriented systems that permit meta-level system manipulation, such as 3-KRS, CLOS, ObjVLisp, and ABCL/R [Watanabe 1988] have been constructed using Lisp-based metacircular interpreters. Reflection is then implemented by modifying the language's interpreter. Smalltalk-80, on the other hand, uses a virtual machine.

Although the Smalltalk-80 virtual machine is a byte code interpreter that is usually implemented in machine language or C, the official definition of the virtual machine is written in Smalltalk[Goldberg 83]. Moreover, the Smalltalk-80 debugger uses a byte-code interpreter written in Smalltalk. Indeed, debugging and tracing are frequently cited as applications that are well addressed using reflection. In general, though, secondary interpeters are probably too inefficient to serve as a practical mechanism for the implementation of reflective features. Hence, we have focused instead on adding certain reflective facilities to the Smalltalk virtual machine.

Smalltalk already exhibits many reflective facilities; programs can examine their run-time stacks and can redefine their methods. These facilities were exploited by LaLonde and Van Gulik [LaLonde 1988] to construct a backtracking facility in Smalltalk. We can add more reflective facilities by examining each part of Smalltalk and seeing how each could be overridden.

1) Variable read and write

Variable accesses can be reflected upon in a number of ways. If variable accesses are implemented using messages, as in Self[Ungar 88], then reflective mechanisms for messages will also work for variables. Another alternative is to introduce active variables, as was done in Smalltalk-80 by Messick and Beck [Messick 1985]. This approach involved modifying the Smalltalk-80 compiler to convert variable accesses into message sends to ActiveVariable objects, which in turn regulated access to their contents.

2) Sending a message

Messages sent by an object can be intercepted by modifying the Smalltalk-80 compiler to wrap code around each send operation. Mechanisms that allow a specialized interpreter to preempt the default one can also be used to intercept message sends. The Smalltalk debugger is implemented using an interpreter that runs on top of the Smalltalk virtual machine. Intercepting each message sent by an object seems useful primarily for applications like tracing and debugging.

3) Receiving a message

We have concentrated on the mechanism for handling messages sent to an object. The message send is one of the fundamental notions upon which object-oriented systems are based, so redefining its meaning is very powerful. Later, we will show how to efficiently redefine message look-up and some of its many uses.

4) Returns

If an object can intercept messages sent to it before dispatching them, its dispatching routine can inspect results returned to it before returning them itself. Thus, redefining returns falls out of redefining message dispatching.

Changing Method-Lookup in Smalltalk

Pascoe [Pascoe 1986] described a clever scheme for customizing method dispatch in Smalltalk-80. His approach, rather than relying on compiler and code modifications, involved the creation of a parallel hierarchy of classes called Encapsulators. An Encapsulator is an object that surrounds another (single) object. Pascoe exploited Smalltalk-80's **doesNotUnderstand:** mechanism to construct these objects.

When a message is sent to a Smalltalk-80 object, the method dictionaries associated with that object's class and its superclasses are searched at runtime. If none of these classes implement a method for a given message, the Smalltalk virtual machine sends the object the message **doesNotUnderstand:.** The original message selector and message arguments are bundled together in a Message object and passed as the argument to **doesNotUnderstand:**. The default method for this message is stored in class Object. This method invokes the Smalltalk debugger, since sending an object a message it does not implement is usually a sign of programmer error. However, objects that override **doesNotUnderstand:** can intercept unimplemented message at runtime, and process them as they see fit.

Encapsulators are "wrapped around" the objects they encapsulate. Unadorned Encapsulators forward every message sent to them to their client object. However, pre- and post- actions (**EPreAction** and **EPostAction**) as well as initialization code can be specified by overriding certain default Encapsulator methods (**EPreAction**, **EPostAction**, and **EInitialize**). **EPreActions** can include message screening, synchronization, and the like. **EPostActions** can gain access to the value returned, perhaps altering the returned value, if appropriate.

Pascoe's approach, in effect, uses message forwarding to an encapsulated component to effect method lookup customization. This is message forwarding, and not true delegation, because the **self** problem [Liebermann 1986] is not addressed (see below).

Encapsulators are good for building objects like distributed objects [Bennett 1987] [McCullough 1987], since the object being forwarded to is in another address space from the original object receiving the message. However, they do not work well for atomic objects, since it is possible for the object being encapsulated to escape from the encapsulator and allow other objects to send it messages directly.

Messick and Beck discuss a tool to insert code before and after a given method without having to recompile the method. They call such methods advised methods, after a related feature in Interlisp. They used this facility for setting conditional breakpoints, for argument and result type checking, for coercion, for tracing, and for changing method interfaces. Advised methods show the value of being able to wrap a piece of code around an existing method without changing it.

One interesting aspect of Messick and Beck's work is that it illustrates how manipulation of the Smalltalk-80 system's self representation can be undertaken at several different levels of the system. The implementation of active variables took advantage of the fact that Smalltalk-80 allows the Smalltalk compiler itself to be modified by users of the system. The advised method feature involved the manipulation of compiled methods. It is instructive to contrast these approaches with those that use only pre-existing reflective facilities (such as the backtracking work) and with those that entail modification (or replacement) of the virtual machine itself (see below).

Method Lookup in TS

Customized method-lookup routines can be efficiently implemented. One technique was used by Chris Lanier as part of TS, an optimizing compiler for Smalltalk that we are building [Johnson 1988b] This technique employs a class instance variable that refers to a method lookup routine for that class. It uses a variation of in-line caching, as originally implemented in PS, ParcPlace Smalltalk [Deutsch 1983] that allows custom method lookup routines to be executed with almost no performance penalty.

Note that the vast majority of classes will not need to override method lookup. A mechanism for allowing method-lookup to be overridden should impose no overhead on classes that do not use it.

In-line method caching is a technique for optimizing method lookup that takes advantage of the dynamic locality of type usage by caching the method last associated with a given message send in-line. Deutsch and Schiffman report that this strategy is effective about 95% of the time.

In-line caching works as follows. A message send with no cache entry is an*unlinked* entry. For example, consider this message send:

```
push    receiver            ;Stack the receiver...
push    arg1,...,argn       ;Stack the args...
call    unlinkedDispach     ;Lookup and fill in cache
<message selector>          ;Selector to send...
<unused>                    ;Unlinked entries don't use.
                            ;Send will return here...
```

When the unlinkedDispatch routine is called it performs a standard method-lookup, and associates a method with the given receiver and selector. It then modifies the call in place as follows before completing the send:

```
push    receiver            ;Stack the receiver...
push    arg1,...,argn       ;Stack the args...
call    linkedDispach       ;Try previous method first...
<last class>                ;Class of cached method...
<method>                    ;Method to call first...
                            ;Send will return here...
```

The modified call is said to be *linked*. (The changes made to create the cache entry are shown in italics.) A linked cache entry contains the class of the method stored in the cache, together with the address of the method itself. The linkedDispatch routine must determine whether the class of the cached entry matches the class of the receiver of the current message send. If this is the case (as it usually will be) the method can be dispatched to directly. When there is a mismatch, the cache entry must be relinked by performing a full method-lookup. Some implementations of in-line caching call the method itself directly, instead of using a dispatching routine. The code to check the validity of the cache entry must then be prefixed to every compiled method.

Deutsch and Schiffman note that this sort of dynamic code modification is generally condemned in modern programming practice. It is, however, very much in the spirit of the work on reflection.

Under TS, each class stores its method look-up routine in one of its instance variables. Since the linkedDispatch and unlinkedDispatch routines are the method-lookup routines, redefining method lookup means creating new dispatch routines. There is a new version of the linkedDispatch routine for each redefinition of method lookup. The code that links cache entries must check whether the dispatch routine has changed, too, and might need to jump to a new dispatch routine. Classes that use the default method lookup routine will behave exactly like PS, so method-lookup will continue to be fast. The overhead associated with this method lookup scheme was about the same as that of method caching alone in PS. We plan to eventually write method-lookup routines in Typed Smalltalk, but at the moment all of them are written in machine language.

Reflective Facilities in Smalltalk

Under ParcPlace Smalltalk (PS) we implemented our specialized method lookup routines in Smalltalk-80 itself. Whenever an object belonging to a class designated as a *dispatching class* (using a bit in the class object's header) is sent a message, that object is instead sent **dispatchMessage: aMessage.** This feature is implemented using essentially the same mechanism as **doesNotUnderstand:**. The method associated with **dispatchMessage** is then free to handle the message as it sees fit. A mechanism for allowing a dispatching class's methods to call other class methods without reinvoking the dispatching mechanism is provided.

A related class type is the *parentless class.* These classes invoke the message dispatching mechanism if the message sent to them is not found in the method dictionary of the first class searched. The effect achieved is the same as that of removing the class's superclass chain.

These features are augmented with mechanisms for supporting lightweight anonymous classes and *metaobjects.* Metaobjects are class objects that refer to a single instance, their referent. They are similar in this respect to Smalltalk's Metaclass objects, which describe a single class object. We currently implement lightweight classes and metaobjects as subclasses of Behavior. We allow these objects to be introducted and removed from the inheritance paths of individual objects dynamically. These class objects are not made visible to the parts of the Smalltalk system that are involved with the management of the system's class hierarchy, and have no global names. The figure below illustrates how a metaobject that specializes a single instance of class Point fits into the class hierarchy.

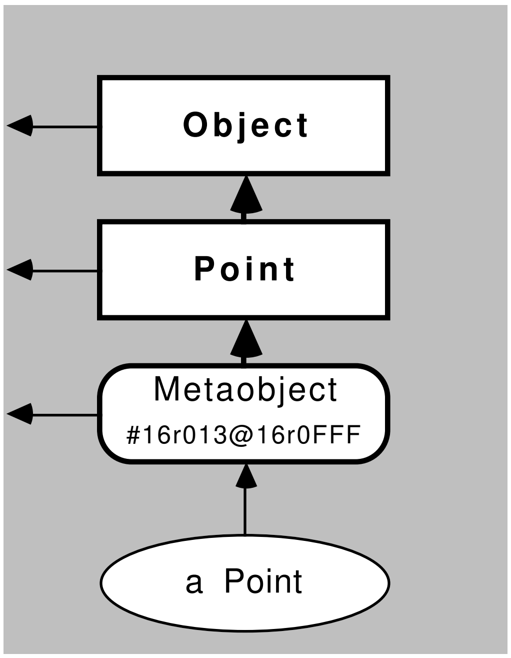

Figure 1

The use of a **doesNotUnderstand**-based dispatching mechanism limits the efficiency of the code that uses it. Our TS based implementation of specialized method lookup routines based on in-line caching can be very efficient. The only overhead is the extra time needed to link dispatch routines, which is very small. We believe that a synthesis of the two approaches discussed above can retain the flexibility associated with programming reflective code in Smalltalk-80, as well as the efficiency of the TS-based lookup customization scheme.

Applications

A classic example of the need for redefining message dispatching is making an object that only responds to one message at a time, i.e. an atomic object [Yokote 1986]. An atomic object has a semaphore, and every process that sends a message to it must first acquire that semaphore. When the message returns, the semaphore is released. This is easily implemented by having the message dispatch routine acquire the semaphore before executing the method and releasing it afterwards. Our implementations of monitors and atomic objects bear a fairly strong resemblance to those given in [Pascoe 1986].

A future object differs from an atomic object because the final methods that are executed belong to the value of the future, not the future object itself. The future has a semaphore and a value. The semaphore is originally locked. Once released, the semaphore is never locked again, and messages are relayed to the new value of the future instead of being executed by the future itself.

The Smalltalk-80 code that follows illustrates a simple implementation for Future objects. The code below shows how a future object for the result of a block that calculates 2+2 might be created. The given block is executed in parallel with the active process. The **promising:** method immediately returns a Future object, which can be thought of as a place holder for the result being computed in parallel. Messages sent to the future (such as the **printString** message below) while the process computing the Future's result is still in progress cause the current process to be suspended until the Future arrives.

```
|f|
f <- Future promising: [2+2].
f printString '4.0'
```

Future class methodsFor: 'instance creation'

The instance creation message below creates an instance of Future promising the result of the given block. Classes that implement a **dispatchMessage:** method (other than the default one for class Object) use a bit in the class object's header to indicate that this method should be executed any time a message is sent to an instance of that class. We allow dispatching classes to execute **perform:withArguments:** without reinvoking the **dispatchMessage:** method. This is how the default **dispatchMessage:** method dispatches messages.

promising: aBlock

```
| aFuture |
aFuture <- self new.
^aFuture promising: aBlock
```

The Future instance method below first allocates a semaphore. It then creates and starts a process that calculates a result for the given block and signals this semaphore when this calculation is complete. The Future object itself is the immediate result of this method.

Future methodsFor: 'initialization/dispatching'

```
promising: aBlock
    "Create a semaphore, and fork a block that
    will signal it.  The result of this block is stored
    in result..."

    semaphore <- Semaphore new.
    [result <- aBlock value.
    semaphore signal] fork.
    ^self
```

The **dispatchMessage:** method is invoked when the Future object is sent (nearly) any message whatsoever. In the example given above, the message sent it is **printString.** This method first tests to see whether the message sent the Future is a **promising:** message. If so, it is dispatched using the inherited message dispatching method.. Otherwise, the method waits until the Future arrives. It then tests the result to see if it is a SmallInteger. Since these do not have unique object pointers, they do not work with the **become:** message. In such cases, the value is converted to a floating point value. Finally, the **become:** message is executed to exchange the identities of the Future and result objects.

The beauty of using **become:** here is that all subsequent references to the object pointer originally returned when the Future object was created will now refer directly to the result object. Subsequent references to this object will no longer incur any dispatching overhead. In systems where **become:** is slow and dispatching is fast (e.g. TS) it would be better to omit the **become:**. The final act of the method below (and hence of the Future object itself) is to forward the message that invoked **dispatchMessage:** (the **printString**) to the result object.

```
dispatchMessage:  aMessage

    "If this is our init message, let it by..."
    aMessage selector == #promising:
        ifTrue: [^super  dispatchMessage:
                    aMessage].

    "Wait until our result is available..."
    semaphore wait.

    "If our result is a SmallInteger, it has no oop.."
    (result isKindOf: SmallInteger)
        ifTrue: [result <- result asFloat].

    "Become the result and do the
    deferred message..."
    result become: self.
    ^super dispatchMessage: aMessage
```

Remote objects are like future objects in that they do not need to perform method look-up, but transform the message into another form. In the case of remote objects, messages get translated into network packets and sent to the machine at which the objects are located. [Bennett 1987] [McCullough 1987].

Actors can also be implemented easily. Each actor has a message queue and a process. The process repeatedly reads a message from the message queue and then dispatches on it. The message dispatch routine for the actor simply turns the message into an object and places it onto the queue.

Languages based on delegation and prototypes let each object have its own specialized methods. We implement per-object behavior modification using Metaobjects. True delegation [Lieberman 1986] requires that the original self be maintained somehow when a message is delegated to one of an object's components. Efficiently rebinding self in this fashion at runtime would seem to require modifications to the Smalltalk virtual machine.

We have investigated two strategies that can be employed at the Smalltalk level to implement this behavior. One approach involves passing the "outer self" to an object's component as part of a message delegating the original message to that component. Another approach entails augmenting an object's components with a Component metaobject. This metaobject screens all its messages for messages sent to self from itself, and forwards these to the object designated as the component's container.

There is a very strong similarity between the notions of delegation to an object's components and inhertance [Stein 1987]. We contend [Johnson 1988a] that inheritance, and multiple inheritance in particular, are overused in situations where forwarding or delegating messages to components would be more appropriate. One reason for this might be that inheritance is frequently the best supported code sharing mechanism available. In contrast, message forwarding must be implemented explicitly on a method-by-method basis.

Dispatching classes let the Smalltalk programmer experiment with a number of alternate approaches to the problem of composing new objects from existing ones. Among these are:

1) Prioritized forwarding to components

It is easy, using dispatching classes, to construct objects that test their instance (component) variables in some prioritized order to see whether their contents responds to a given message, and to forward such messages to these components.

2) Dynamic fields

Message interception (sometimes in conjunction with metaobjects) can also be used to construct objects with dynamic fields. Such objects can respond to the standard external instance variable access protocol, and convert references they don't understand to dictionary accesses. Alternately, they can permit dictionary-style iteration over static, record-like objects. One scheme for doing this using **doesNotUnderstand:** is given in [Foote 1988].

3) Dynamic protection

Specialized dispatching routines could also perform dynamic protection functions, based on the class of the sender of the message. A protected object's class (or metaobject) might contain several dictionaries mapping classes or even individual instances to sets of message selectors. An object's membership (or lack thereof) in one of these sets would determine its elegibility to send the object the given message. Some sort of wildcard conventions, as well as default groups, might be allowed.

4) Multiple views

An alternate way to protect objects is to export references not to the objects themselves, but to objects that forward restricted sets of the original object's protocol to the original objects. Such a scheme could permit different clients to have different "views" of an object. This mechanism might resemble operating system capabilities.

5) Protocol matching

Yet another application is the construction of protocol mapping adapters. These are objects that can be fitted between sets of existing objects to translate between differing sets of protocol assumptions. Such capabilities might prove valuable in constructing open systems [Hewitt 1983].

Customized method dispatching code can also be be used for the collection of performance data, and dynamic optimizations such as result caching and coercion. Another application is the construction of persistent objects.

Conclusions

Computational reflection provides a framework for organizing object-oriented systems that addresses a number of programming problems that are awkward to address in conventional object-oriented systems. In Smalltalk-80, the addition of facilities for overriding the message dispatch process on a class wide basis can provide much of the same power.

An object, during the course of its lifetime, may enter into a number of different relationships with other objects in its environment. Some of these relationships will be permanent, or at least relatively static, and others might be quite ephemeral. Some of these relationships might be quite complex. In conventional object-oriented systems, one such relationship, inheritance, is well supported. However, the facilities for allowing the programmer to construct similarly powerful mechanisms of this sort of his or her own are limited. It remains to be demonstrated that such facilities will lead to the discovery of mechanisms of enduring value. The reflective facilities discussed herein can allow such issues to be explored in Smalltalk-80.

Adding a carefully chosen set of reflective facilities to a language like Smalltalk-80 lets it solve many of the same problems that a fully reflective system can, with much greater efficiency. These facilities can be implemented so that they have little impact on code that doesn't use them.

Smalltalk-80 is not well suited for parallel programming or distributed programming, as can be seen by the number of attempts to extend it for these areas. However, instead of extending the language repeatedly each time a weakness is discovered, it is better to give it enough power to extend itself. This makes it more likely that an existing system can be modified to reuse it in future contexts that cannot now be predicted. By adding the ability to redefine message dispatching, object-oriented languages like Smalltalk-80 will have less need to be extended in the future. Reflective facilities can also assist complex systems in adapting to changing requirements as they evolve by permitting a wider range of mechanisms for fitting existing objects to new situations.

Acknowledgements

The second author was supported by NSF contract CCR-8715752. We would like to thank Peter Deutsch and ParcPlace Systems for providing tools and support for the PS based portion of this work. We are grateful to Brian Marick for helpful comments on an earlier draft of this paper.

References

[Agha 1986]
Gul Agha
ACTORS: A Model of Concurrent Computation in Distributed Systems
MIT Press, 1986

[Bennett 1987]
John K. Bennett
The Design and Implementation of
Distributed Smalltalk
OOPSLA '87 Proceedings
Orlando, FL, October 4-8 1977 pages 118-330

[Bobrow 1988a]
D. G. Bobrow, L. G. DeMichiel, R. P. Gabriel,
S. E. Keene, G. Kiczales, and D. A. Moon
Common Lisp Object System Specification X3J13
Document 88-002R
SIGPLAN Notices, Volume 23,
Special Issue, September 1988

[Bobrow 1988b]
Daniel G. Bobrow and Gregor Kiczales
The Common Lisp Object System
Metaobject Kernel -- A Status Report
Proceedings of the 1988 Conference on Lisp
and Functional Programming

[Deutsch 1984]
L. Peter Deutsch and Allan M. Schiffman
Efficient Implementation of the
Smalltalk-80 System
Proceedings of the Tenth Annual ACM Symposium
on Principles of Programming Languages,
1983, pages 297-302

[Foote 1988]
Brian Foote
Designing to Facilitate Change with
Object-Oriented Frameworks
Masters Thesis, 1988
University of Illinois at Urbana-Champaign

[Goldberg 1976]
Adele Goldberg and Alan Kay, editors
with the Learning Research Group
Smalltalk-72 Instruction Manual
Xerox Palo Alto Research Center

[Goldberg 1983]
Adele Goldberg and David Robson
Smalltalk-80: The Language and
its Implementation
Addison-Wesley, Reading, MA, 1983

[Goldberg 1984]
Adele Goldberg
Smalltalk-80: The Interactive
Programming Environment
Addison-Wesley, Reading, MA, 1984

[Halstead 1985]
R. Halstead
MultiLISP: A language for Concurrent
Symbolic Computation
ACM Transactions on Programming Languages
and Systems
October 1985, pages 501-538

[Hewitt 1983]
Carl Hewitt and Peter de Jong
Analyzing the Role of Description and Actions
in Open Systems
AAAI '83, pages 162-167

[Ingalls 1978]
Daniel H. H. Ingalls
The Smalltalk-76 Programming System
Design and Implementation
5th ACM Symposium on POPL, pp. 9-15
Tucson, AZ, USA, January 1978

[Johnson 1988a]
Ralph E. Johnson and Brian Foote
Designing Reusable Classes
Journal of Object-Oriented Programming
Volume 1, Number 2, June/July 1988
pages 22-35

[Johnson 1988b]
Ralph E. Johnson, Justin O. Graver, and
Laurance W. Zurawski
TS: An Optimizing Compiler for Smalltalk
OOPSLA '88 Proceedings
San Diego, CA, September 25-30, 1988
pages 18-26

[Keene 1989]
Sonya E. Keene
Object-Oriented Programming in Common Lisp
A Programmer's Introduction to CLOS
Addison-Wesley, 1989

[LaLonde 1986]
Wilf R. LaLonde, Dave A. Thomas and John R. Pugh
An Exemplar Based Smalltalk
OOPSLA '86 Proceedings
Portland, OR, October 4-8 1977 pages 322-330

[LaLonde 1988]
Wilf R. LaLonde and Mark Van Gulik
Building a Backtracking Facility in Smalltalk Without
Kernel Support
OOPSLA '88 Proceedings
San Diego, CA, September 25-30, 1988
pages 105-122

[Lieberman 1986]
Henry Lieberman
Using Protypical Objects to Implement
Shared Behavior
in Object-Oriented Systems
OOPSLA '86 Proceedings
Portland, OR, October 4-8 1977 pages 214-223

[Maes 1987a]
Pattie Maes
Computational Reflection
Artificial Intelligence Laboratory
Vrije Universiteit Brussel
Technical Report 87-2

[Maes 1987b]
Pattie Maes
Concepts and Experiments in
Computational Reflection
OOPSLA '87 Proceedings
Orlando, FL, October 4-8 1977 pages 147-155

[McCullough 1987]
Paul L. McCullough
Transparent Forwarding: First Steps
OOPSLA '87 Proceedings
Orlando, FL, October 4-8 1977 pages 331-341

[Messick 1985]
Steven L. Messick and Kent L. Beck
Active Variables in Smalltalk-80
Technical Report CR-85-09
Computer Research Lab, Tektronix, Inc., 1985

[Pascoe 1986]
Geoffrey A. Pascoe
Encapsulators: A New Software
Paradigm in Smalltalk-80
OOPSLA '86 Proceedings
Portland, OR, September 29-October 2 1986,
pages 341-346

[Smith 1983]
Brian Cantwell Smith
Reflection and Semantics in Lisp
Proceedings of the 1984 ACM
Principles of Programming Languages Conference
pages 23-35

[Smith 1987]
Randall B. Smith
Experiences with the Alternate Reality Kit:
An Example of the Tension Between Literalism
and Magic.
CHI+GI 1987 Conference Proceedings

[Stefik 1986a]
Mark Stefik and Daniel G. Bobrow
Object-Oriented Programming:
Themes and Variations
AI Magazine 6(4): 40-62, Winter, 1986

[Stefik 1986b]
M. Stefik, D. Bobrow and K. Kahn
Integrating Access-Oriented Programming into
a Multiprogramming Environment
IEEE Software, 3, 1 (January 1986), 10-18

[Stein 1987]
Lynn Andea Stein
Delegation is Inhertance
OOPSLA '87 Proceedings
Orlando, FL, October 4-8 1977 pages 138-146

[Stroustrup 1986]
Bjarne Stroustrup
The C++ Programming Language
Addison-Wesley, Reading, MA, 1986

[Tiemann 1988]
Michael D. Tiemann
Solving the RPC problem in GNU C++
1988 USENIX C++ Conference
Denver, CO, October 17-21 1988

[Ungar 1987]
David Ungar and Randall B. Smith
Self: The Power of Simplicity
OOPSLA '87 Proceedings
Orlando, FL, October 4-8 1977 pages 227-242

[Yokote 1986]
Yasuhiko Yokote and Mario Tokoro
The Design and Implementation of
ConcurrentSmalltalk
OOPSLA '86 Proceedings
Portland, OR, September 29-October 2 1986,
pages 331-340

[Watanabe 1988]
Takuo Watanabe and Akinori Yonezawa
Reflection in an Object-Oriented Concurrent
Language
OOPSLA '88 Proceedings
San Diego, CA, September 25-30, 1988
pages 306-315