Implementational Reflection in Silica

Ramana Rao

Xerox Palo Alto Research Center

3333 Coyote Hill Road; Palo Alto, CA 94304

Email: rao@parc.xerox.com

Abstract

The value of computational reflection has been explored in a number of programming language efforts. The major claim of this paper is that an ostensibly broader view of reflection, which we call implementational reflection, can be applied to the design of other kinds of systems, accruing the same benefits that arise in the programming language case. The domain of window systems in general, and the Silica window system in particular are used to illustrate how reflection can be applied more broadly. Silica is a CLOS-based window system that is a part of the Common Lisp Interface Manager, an emerging user interface programming standard for Common Lisp.

Introduction

One meaning of the word reflect is to consider some subject matter. Another is to turn back something (e.g. light or sound). Punning on these two meanings, we get the notion of turning back one's consideration or considering one's own activities as a subject matter. Our ability as humans to reflect in this sense has been credited, since Aristotle, with our success in adapting to new situations and mastering our environment. Naturally, it was widely conjectured in the artificial intelligence community that by providing reflective capabilities to computational systems, we would obtain systems with greater plasticity and consequently, enhanced functionality.

Hence, this notion was introduced in a number of languages including the procedural language 3-LISP[Smi84], the logic-based languages FOL[Wey80] and META-PROLOG[Bow86], and the rule-based language TEIRESIAS[Dav80]. These various efforts have shown that facilities for reflecting on the computational process can offer users the ability to control or monitor a language's behavior and to extend or modify its semantics in an elegant and principled way.

More recently, reflection has been gaining momentum as a major topic in the design of object-oriented languages. A number of object-oriented languages including CLOS[BKK+86, KdRB91], 3-KRS[Mae87], ObjVlisp[Coi87], ABCL/R[WY88], and KSL[IC88] have embraced reflection as a first class concern. Besides adding to the general understanding of reflection's benefits, these efforts have elaborated on the use of object-oriented programming technology for building reflective systems.

The primary purpose of this paper is to establish that reflection can be applied to the design of systems other than programming languages and that this endeavor can attain the same benefits for the users of such systems. We reformulate the framework of reflection in terms of a system's implementation. In particular, we introduce the concepts of implementational reflection (as opposed to computational reflection) and open implementation (as opposed to a reflective architecture) in the next section. This reformulation helps clarify what it means for a system other than a programming language to support reflection.

We substantiate our thesis by considering the domain of window systems, and within this domain, offer the Silica system, designed and implemented by the author, as an example. Silica is the portable window system layer of the Common Lisp Interface Manager (CLIM) [RYD91], an emerging standard user interface programming interface for Common Lisp (which includes CLOS). Silica can be viewed as a window system for a single address space environment (analogous to Interlisp-D[BKM+80], Symbolics[Sym], or Smalltalk-80[KP88, LP91]), or alternatively as an extended window system model for an application address space in a multiple address space environment that provides a window library (e.g. SunWindows[Sun86]) or a window server (e.g. X 11[SG86]). Most pertinent to this paper's purposes, Silica supports reflection on its implementation, thus providing a structured framework for allowing users to explore various window system semantics or implementations.

After presenting an account of implementational reflection, we describe the functionality provided by window systems and two scenarios where reflection would be useful. Following that, relevant features of Silica's open implementation are described. Then the two scenarios are revisited to explain how Silica's reflective facilities can be used. The paper concludes with a discussion of various questions and issues that arise when building systems with open implementations, particularly in object-oriented languages.

1 Implementational Reflection

The notion of reflection describes a wide range of activities loosely characterized as some form of self-analysis, often in service of initiating or informing subsequent actions. Hence, a reflective system, besides computing about some base domain, must compute about itself. But what does it mean for a system to compute about itself? Since a system is represented as a program and is embodied in a computational process that arises from the execution of that program, the following view on reflection is typical:

> *Computational Reflection.* Reflection involves inspecting and/or manipulating representations of the computational process specified by a system's program.

Thus, computational reflection allows a system to participate in how its program is executed. For example, many of the language systems cited above allow a program to alter control flow in response to analysis of various runtime interpretation structures. In particular, these language systems provide a separate level, often called a *metalevel*, for computing about the current state of the *base level* computation and allow this metacomputation to alter the control flow of the base level computation.

Though the computational reflection view adequately describes the application of reflection to programming languages and their builtin mechanisms, most significant systems depend not only on these constructs, but also on other systems that they utilize. This suggests another view on reflection:

> *Implementational Reflection.* Reflection involves inspecting and/or manipulating the implementational structures of other systems used by a program.

Implementational reflection allows a program to participate in the implementation of systems that it utilizes. For example, the CLOS Metaobject Protocol specified in [KdRB91] allows users of CLOS to control the implementation of instance representation. This capability can be used to select an instance representation that is appropriate for a given specific situation. For example, Figure 1 illustrates two different instance representations tuned to different requirements: the point class has a small number of slots that need to be accessed quickly, whereas the person class has hundreds of slots, many of which may not be used in any given instance.

Two observations suggest that computational and implementational reflection are, in fact, just different characterizations of the same essential capability. First, a language interpreter, which generates a computational process from a program, is the implementation of a language. And second, the interface of any system can be seen as a language, and the system's implementation as

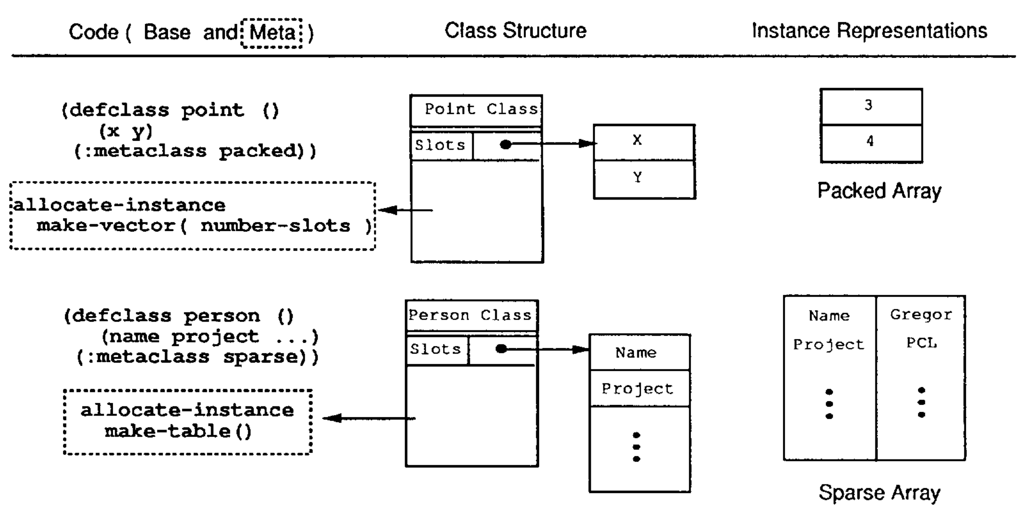


Figure 1: A metaobject protocol for CLOS allows the user to select instance representations tuned to their needs. The point class uses fixed storage, which is appropriate since slot access speed is important. The person class uses variable storage since for a given person object, many slots may not have a value and hence, space savings are the dominant factor.

an interpreter for that language. The first observation indicates that computational reflection is a special case of implementational reflection, and the second observation indicates the converse. In any case, the implementational reflection view allows us to recast much of the framework provided in the reflection literature in terms more familiar to system builders.

1.1 From Implementational Access to Open Implementations

To a certain extent, support for reflection is a matter of degree: many existing systems provide limited cases of reflective capability. In [Mae88], a valuable distinction is made between reflective facilities and a fully reflective architecture. A reflective facility is one that allows the user to query or manipulate some aspect of implementation using a set of predefined operations. As Smith and Maes have pointed out, many programming languages provide access to implementational constructs. For example, Common Lisp provides access to its interpreter (`eval`), its compiler (`compile`), its control stack (`unwind-protect`, `catch` and `throw`), and its special binding environment (`boundp` and `makunbound`).

Similarly, a number of existing non-language systems support operations which are about the implementation itself. The GKS graphics standard[ANS85] allows the user to query whether certain features are supported and thus adapt to different GKS implementations. The X window system[SG86] supports a more general version of this feature. The X11 protocol provides a request for determining whether a desired extension is supported. If the extension is supported, this request returns the information necessary to use the extension. This request is reflective because it supports a dialogue about functionality and implementation.

A *reflective architecture*, on the other hand, according to the discussion in [Mae87, Mae88], allows much more open-ended access to a language's implementation. In particular, a reflective architecture allows writing code that is invoked by the language interpreter. This reflective code participates in language interpretation by manipulating *causally connected* representations of the computational

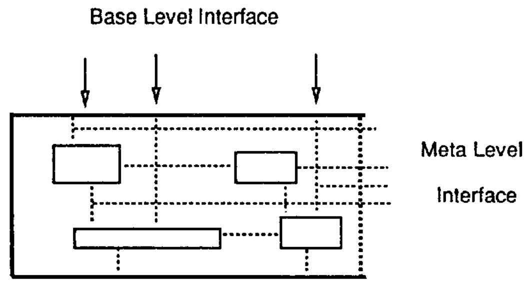


Figure 2: A system with an open implementation, besides providing a familiar interface to its functionality called a base level interface, reveals aspects of its implementation through a metalevel interface. The metalevel interface defines points in the implementation that can be tailored by the user.

process. Causal connection means, on one hand, that the representations accurately render the state of the computation and, on the other, that mutating the representations will influence subsequent computation.

The concept of a reflective architecture can be reformulated in terms of the implementation of a system. A system with an *open implementation* (depicted in Figure 2) provides (at least) two linked interfaces to its clients, a *base level interface* to the system's functionality similar to the interface of other such systems, and a metalevel interface that reveals some aspects of how the base level interface is implemented. In particular, the metalevel interface specifies points where the user can add code that implements base level behavior that differs in semantics and/or performance characteristics from the default base level behavior. Since this metalevel code directly implements aspects of the base level, the causal connection requirement of reflection is straightforwardly met.

Whereas features that provide limited access to a system's implementation provide some measure of system flexibility, an open implementation provides a more open-ended framework for exploring a space of implementations or semantics. Returning to earlier examples, whereas the primitives in Lisp do not support exploring alternative stack or environment implementations or semantics, a metaobject protocol for CLOS does allow implementing a range of instance representation strategies. And whereas the one request in the X window system does not facilitate exploring alternative window system implementations or semantics, a window system with an open implementation would.

1.2 Designing the Metalevel Interface

An obvious consequence of providing an open implementation is that a system is forced to make commitments to particular implementation details. This does not, however, mean that every aspect of its implementation is specified or that users can alter the implementation arbitrarily. In the parlance of the reflection community, an open implementation *reifies* some aspects of implementation, and *absorbs* others, meaning some aspects are made explicit and other are left implicit. In the sense of [Smi84, Smi82], an open implementation is based on a *theory* (i.e. model) which determines the reach of the system's metalevel, i.e. the extent to which base level behavior can be altered by the user. Smith calls this the *theory relativity* of reflection. This concept is illustrated in [Mae88] by contrasting several metacircular interpreters for Lisp, each of which reifies different aspects of Lisp interpretation.

In more traditional terms, just as a system provides an interface to its base functionality, an open implementation provides a well-defined interface (i.e. the metalevel interface) to the implementation of the system. The elaboration of a metalevel interface must address two, sometimes competing, sets of concerns. On one hand, the architecture or facilities prescribed by the metalevel interface must not prevent efficient and effective implementation of the base level. On the other, the metalevel interface must give the user access to aspects of the implementation which can be exploited to create either useful semantical variations or more efficient implementations for particular situations. Balancing these two sets of concerns is the major challenge in designing a system that supports implementational reflection.

2 Window Systems

In this section, we present an account of the functionality provided by window systems (i.e. what they are about) that provides the basis of a theory for Silica's open implementation. We also present two scenarios in which reflection on a window system's implementation would be useful. Later in the paper, we will explain how Silica can be used in these scenarios.

A window system allows multiple applications to share the bounded interactive resources of an computer system, in particular, its input devices and screen(s). The fundamental concept in window systems is a ***windowing relationship.*** A windowing relationship defines how a region in one coordinate system, either a real piece of screen real estate or a virtual region (arising from another windowing relationship), is divided or shared amongst a number of independent virtual regions called *windows*, each of which has its own coordinate system. Window systems often make assumptions in their support for windowing relationships. For example, most window systems provide windows that occupy a "two and a half" dimensional space that are stacked and thus may appear to overlap other windows, though some window systems just support tiled windows.

Many early window systems did not support the broad use of windowing relationships within an application, but rather focused on the desktop level as the primary client. However, others, especially more recent window systems (e.g. X and NeWS[Sun87]), allow the nesting of windows within other windows, thus creating many-level hierarchies of windowing relationships.

Besides managing one or more windowing relationships, window systems also provide output and input functionality. On the output side, window systems implement the graphics capabilities of windows, ensuring that output on one window does not affect the area allocated to other windows.[1] They generate repaint events on a window when window management causes it to be exposed. On the input side, window systems determine which windows to distribute input device events to, and how to deliver the events to the clients of those window. In short, a window system provides a basis for building graphical user interfaces by providing windows which support nesting or sharing of space, and output and input operations.

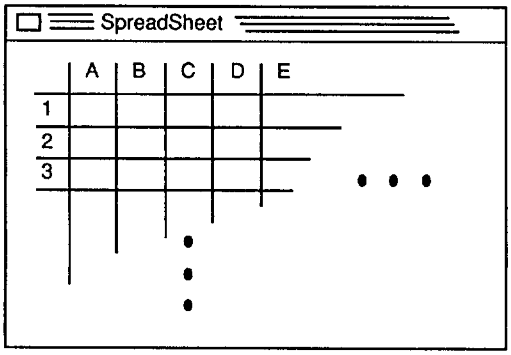
The first scenario illustrates that window system functionality is very similar to the functionality needed within applications for managing space, input, and output. Unfortunately, even though the needed functionality is of a kind provided by the window system, the user must often abandon the window system and build ad hoc support because it is impractical to use the monolithic window system implementation.

2.1 Building a SpreadSheet

In this scenario, we explore building a spreadsheet. In a spreadsheet, an array of cells is nested within a grid as illustrated in Figure 3. The relationship between the cells of the spreadsheet and the spreadsheet itself can be seen to be a kind of windowing relationship. The spreadsheet needs functionality for managing this nesting of regions, for generating repaints of subsets of the cells, and distributing input to cells.

Given these observations, the code shown in the figure accurately reflects the essence of the desired behavior. Unfortunately, this code is not likely to be practical in a window system with a

[1]If windowing is viewed as the virtualization of display space, this integrity constraint is analogous to not corrupting another application's memory in a virtual memory system.



```
for row from 1 to 100
 for col from 1 to 100
  make-window( row * cell-width,
               col * cell-height )
```

Figure 3: A spreadsheet application divides a grid region into a set of cell regions that display values and receive input. The code on the right succinctly captures the essence of this behavior.

closed implementation. The problem is that windows in the typical general-purpose window system implementation must support at least the desktop level of a window environment and maybe others. This requirement places demands on the algorithms and storage structures used within the window system that are not necessarily appropriate for spreadsheet cells. For example, X supports arbitrary overlapping of windows, which affects the implementation of various internal operations, whereas in this case, the cells never overlap. Furthermore, much of the storage associated with an X window is not necessary for every cell since they all, in general, can share a number of properties.

One possible solution to this problem is to provide other kinds of objects that address different. In fact, support for extremely light-weight window-like objects has been provided in a number of X toolkits including InterViews[CL90] and Motif[Ope89]. One problem with approach is that the new objects are now tuned to a different, but still quite specific set of needs.

A more fundamental problem is that the different implementations do not share structure, in code or necessarily in conception, and hence they do not lend support to one another. Besides the loss of conceptual clarity, this has the material consequence that code based on one system can't be easily mixed with that written for another. For example, suppose we now wanted to nest window system windows within the cell windows, so that window-based code, even entire applications, could be used within the cell. The problem is that this requires adding support to the cell window type for embedding window system windows.

An open implementation, on the other hand, provides an open-ended framework for introducing new types of windows and capturing commonalities across various types. It does this by allowing the user to participate in a well-chosen set of system design choices. Rather than providing a single implementation or a number of disconnected implementations, the user is allowed to specify a tailored solution within a design space.

2.2 Regenerating Output

As a second scenario, consider the problem of redisplaying a window when it becomes exposed by a user- or application-initiated window manipulation operation. Suppose the output on an application's window is particularly complex and that it takes substantial computation to regenerate. As a consequence, when the window is fully or partially exposed, this computation may lead to sluggish repaint of the window.

One solution is to write code for the application that caches the graphical results of this computation so that re-executing it during redisplay can be avoided. This solution places an additional requirement on the application and unless the programmer is disciplined, the redisplay issue can get

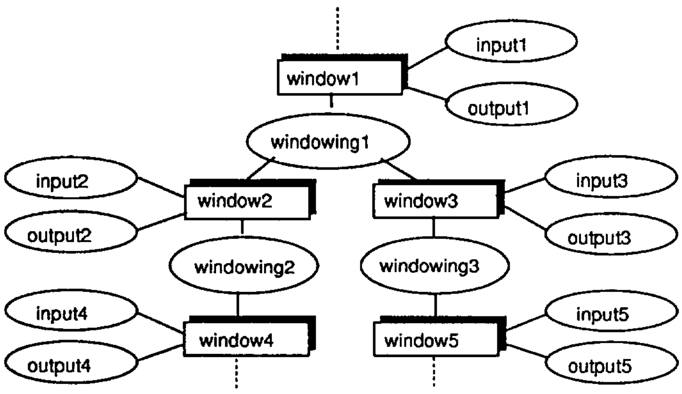


Figure 4: A representative Silica hierarchy. The rectangles represent windows. The ellipses represent contracts which implement part of the functionality of one or more windows. There are three kinds of contract: windowing, input, and output. A windowing contract implements the windowing relationship between a parent window and a set of children windows.

entangled with other application issues.

Moreover, the need for output recording or the choice of an appropriate output recording semantics or implementation may not be the same for all situations. For example, the same application may be used on machines with varying processor speeds or memory capacity, or even display architectures (e.g. one that supports display lists but not bitmaps). This could mean that the application programmer would need several mechanisms.

This type of output regeneration functionality is useful in a broad range of cases, and it seems a natural part of the window system. Specific functionality can clearly be built into the window system, but this approach will eventually lead to a bloated window system that is hard to implement, maintain, and use. On the other hand, we will show later that an open implementation allows adding this behavior in a modular and elegant way as a metalevel abstraction.

3 Silica's Open Implementation

In the last section, we presented an account of window systems and two scenarios that could potentially benefit from an open implementation. In this section, we describe the open implementation of the Silica window system. Silica is implemented in Common Lisp, which includes CLOS, though the aspects described here can be readily implemented in other object-oriented languages.

Silica provides a *base level interface* that is similar to the interface of other window systems. In addition to this base level interface, Silica specifies an architecture that prescribes the skeleton of the base level interface's implementation. In particular, this architecture is specified as a *metalevel interface* that consists of two parts: (i) the *components* that implement the base level behavior and (ii) the object-oriented *protocols* that govern how these components interact to achieve this end. We cover each of these aspects in turn.

Silica's base level interface is based on the same fundamental abstraction as many existing window systems: a tree of light-weight windows that support input and output operations. This interface, among other things, supports constructing and managing window trees;[2] querying their structure; and performing output operations on and receiving input from windows.

[2]Windows are actually called sheets in the existing version of Silica.

- output contract—determines the output capabilities of the window (i.e. how the client can draw images on the window) and how repaint of the window is invoked (e.g. how the client is notified of repaint requests).
- input contract—determines the input interface of the window (i.e. how the client is notified of input events which are distributed to this window).
- youth windowing contract—determines how the window's youth windowing relationship, the one in which it is a child, is managed (i.e. determines how a window behaves as a child). It must be the same or compatible with the adult windowing contract of the window's parent.
- adult windowing contract—determines how the window's adult windowing relationship, the one in which it is the parent, is managed (i.e. determines how a window behaves as a parent). It must be the same or compatible with the youth contracts of the window's children.

Table 1: The responsibilities of a window's four contracts.

In previous window systems, the input, output, and windowing functionality provided for all windows is the same. Silica departs from these window systems by implementing the three areas of window system functionality as distinct manipulable metalevel objects called *contracts*,[3] that can be selected for each window independently. This departure allows clients to select the functionality and implementation of their windows according to their needs, either statically or dynamically to accommodate changing runtime needs.

Figure 4 depicts a metalevel view of an exemplar Silica window hierarchy. Each rectangle represents a window; and each ellipse, a contract of one or more windows. As is shown for `window2` or `window3` in the figure, each window has four contracts, each responsible for a different portion of the window's implementation as described in Table 1. The specific mechanism that associates windows with their four contracts is described in Appendix A.

Contracts are Silica's primary metalevel objects. Input and output contracts implement the input and output functionality respectively of a single window. The case with windowing contracts is more complex (and more interesting) since they implement a windowing relationship involving more than one window. In particular, a windowing contract implements the functionality of one window vis-a-vis its role as a parent, and also the functionality of a number of other windows vis-a-vis their roles as children. A windowing contract, an input contract, and an output contract, thus, comprise the bulk of a local window system for a single window.[4]

The implementational responsibilities of each metalevel component as well as the interactions among components are specified as a set of object-oriented protocols. A *protocol* consists of one or more functions that together perform some subtask in the implementation of the base level interface. In CLOS, protocols include both ordinary Common Lisp functions for fixed portions of the protocol and CLOS generic functions that take one or more of the metalevel objects as specializable arguments for specializable portions of the protocol. Some protocol functions may actually be part of the base level interface, while others implement necessary supporting services. Table 2 presents most of Silica's major protocols, along with the metalevel objects involved in the protocol and a description of the protocol. The details of these protocols are not relevant here; what is important is that each of these protocols performs some task in the implementation of the base level interface and that they circumscribe the aspects of Silica's base level interface that can be changed at the metalevel by the

[3]This term may be unfortunate, since Silica contracts are actually real objects that provide methods rather than declarative specifications. The contracts of [HHG90] are actually more related to what we call a protocol here.

[4]The rest is provided by components that manage global resources. In particular, two other key components are ports and event distributors. Ports manage a connection to a host or remote window system. A port can also be seen as a software port of Silica to a particular host window system or display architecture. Event distributors oversee the distribution of raw input events coming in from ports.

Protocol *implementing objects*	**Responsibility**
Window Construction *global*	Provides a simple interface for constructing windows and establishing windowing relationships. This interface hides the details of realizing a window's contracts. This protocol consults all contract classes to obtain implementation parts for the window.
Windowing Relationship *windowing contract*	Provides functions for adopting, disowning, and "enabling" children; and query methods for asking about parents and children. It also provides functionality for managing window region and coordinate system mapping.
Viewing Parameters *windowing contract*	Provides functions for calculating clipping regions and composing transformations from a window to any of its ancestors (and vice versa) and maintaining a cache for these values. These values are used by, among others, the output protection and the input distribution protocols.
Mirroring *port, windowing contract*	Provides the means for allocating and managing host (or remote) window system windows. Allows implementing top level windows and other kinds of windows that can benefit from the full functionality of a typical heavy-weight window.
Output Protection *output*	Ensures that graphics operations applied to a window are transformed and clipped as appropriate for the window's region and position in the window hierarchy; and that they are appropriately synchronized with changes to the window hierarchy.
Graphics Functionality *output contract*	Provides graphics routines that can be applied to windows and ancillary functionality (e.g. drawing state or graphic context construction and manipulation).
Repaint *windowing contract, output contract*	Provides mechanism for repainting a window when portions are exposed that were previously covered or otherwise not visible.
Input Distribution *port, distributor, windowing contract, input contract*	Determines which window should be the recipient of user input events. Protocol supports extensive participation by local windowing and input contracts.
Input Delivery *input contract*	Translates from port specific event representations (the lowest level representation available) to a representation appropriate for the recipient window. Defines how input is delivered to the client of the recipient window.

Table 2: An Overview of Major Silica Protocols.

user.

Each of Silica's metalevel objects plays a well-defined role in the architecture that is specified by the protocols in which they participate. Some areas of functionality are largely implemented by a single metalevel component, but others are the shared responsibility of several metalevel objects. For example, access to graphics primitives and the delivery of input are primarily the responsibility of the output and input contracts respectively, whereas hit detection and window repainting involve interactions between these contracts and windowing contracts.

An important and familiar technique used to allow responsibility to be divided amongst a number of objects is the *layering* of protocols. A layered protocol invokes subprotocols which implement various subtasks within the protocol's overall task. For example, the repaint protocol invokes subprotocols to calculate what subwindows need repainting and in what order, and to repaint them. Another important benefit of layering is that it allows users to specialize protocols at a level of granularity appropriate to and a cost commensurate with their requirements (discussed further below).

In summary, Silica's metalevel components and protocols provide an architecture for implementing Silica's base level interface and for using this implementation to build window systems with specialized functionality or implementations. Since most Silica protocols are implemented by or consult one or more of a window's contracts and each window has its own contracts, a window's implementation can, to a large degree, be altered locally without affecting distant windows or windowing relationships.

4 Reflection in Silica

An important property of Silica is that its objects and protocols divide into two separate levels, one implementing the other. The collection of windows (the rectangles in Figure 4) form the base level and the collection of contracts (the ellipses) form the implementational or meta level. Similarly, Silica's protocols are layered with their base layer providing standard functionality to the user, and lower meta layers performing various subtasks in the implementation of that functionality.

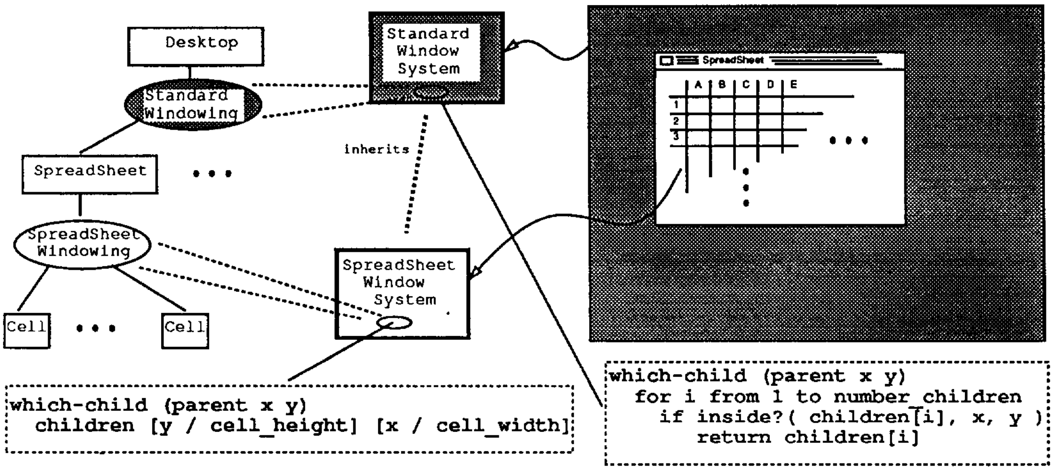
This separation of Silica into two levels is the basis for allowing users to participate in design and implementation decisions, either statically or dynamically, either implicitly or explicitly. A program that uses Silica reflects when it provides contracts which variously define or implement some aspect of the base level interface or an extended or reduced interface. This form of reflective act, where a system (an interpreter) invokes user code at specific points during its implementation (the interpretation process), has been called implicit in [Mae88]. Silica also supports so-called explicit reflective acts by allowing users to manipulate window system functionality explicitly at runtime. This, for example, means that window system behavior like logging input can be added temporarily.

The two scenarios described above can each be handled in Silica by providing reflective code that defines a window system more suited to particular needs. Both cases involve defining a new contract (a windowing contract in the first and an output contract in the second) that specializes one or more Silica protocols. Though in both cases, the code written for the new contracts could clearly be written outside of a window system, Silica allows the framework provided by the window system to be reused by adding such code inside the window system.

4.1 SpreadSheet Revisited

In Silica, the spreadsheet scenario can be handled by implementing a windowing contract specifically tuned to its needs. In particular, this contract can make assumptions about its windowing relationship that can be exploited in its implementation. For example, a windowing contract for this scenario can make two assumptions: first, its children are laid out in a uniform 2-d grid and second, they do not overlap. An appropriate storage representation for storing the children of this contract is a 2-d array indexed by location in the grid.

As part of picking a tuned storage representation, the contract provides specialized methods for various protocol functions. For example, as illustrated in Figure 5, the general purpose method for



```
which-child (parent x y)
  children [y / cell_height] [x / cell_width]
```

```
which-child (parent x y)
  for i from 1 to number_children
    if inside?( children[i], x, y )
      return children[i]
```

Figure 5: A tailored window system that exploits the regularity of the spreadsheet layout is created by combining a specialized windowing contract with the default input and output contracts. In this way, the spreadsheet window system inherits much of the behavior of the standard window system—only the windowing behavior is specialized. (The projection of the windowing contracts into the window system indicates that windowing contracts form only part of the total window system behavior.)

hit detection (invoked by the input distribution protocol) can be replaced by a special one that uses simple arithmetic and array reference. Similarly, a special method like the following can be provided for a function in the repaint protocol.

```
    repaint-children (parent x y w h)
     ...
     for col from (y / cell-height) to ((+ y h) / cell-height)
      for row from (x / cell-width) to ((+ x w) / cell-width)
       repaint( children[col][row] )
     ...
```

An important point to note is that the methods provided by the specialized contract are metalevel code: they are about the implementation of a new kind of windowing relationship, not about the use of a windowing relationship. Metalevel code is like the code system implementors write to implement a window system rather than the code that users typically write to use one.

Using the specialized windowing contract, the spreadsheet application can efficiently create a window for each cell. However, this implementation does require each cell of the spreadsheet to be explicitly created as an object. Though the overhead associated with the window-ness of the cell was reduced significantly, there is still overhead associated with its object-ness. This issue can be addressed in two ways. One, the reflective capabilities of CLOS can be used to reduce unnecessary overhead associated with the cell window as an object.[5] An alternative approach would be to implement a windowing contract which avoids explicitly creating children objects unless or until they are necessary (e.g. for cells that actually have subwindows).

This spreadsheet example suggests an even more ambitious reuse of the window system design and implementation. Traditionally, window systems and toolkits have dealt with the part of the hierarchy

[5]This suggests that reflection on at least partially orthogonal system's used by a program can accumulate benefits.

that deals with objects that act as surfaces to be drawn on, and applications have built their own local mechanisms for dealing with their own hierarchies of objects. Alternatively, window systems have provided a number of predefined window types or user interface toolkits have provided extensive libraries of new window-like objects outside the window system. In contrast, Silica supports adding such facilities (e.g. display lists) as metalevel abstractions by defining new windowing contracts.

4.2 Output Regeneration Revisited

Output capture facilities are not provided primitively in Silica, but rather can be implemented as special output contracts. Such output contracts can extend the standard output semantics to manage the process of output recording. Different implementations of these semantics that optimize this behavior variously can be provided. If the code is running on a very fast processor and performance is not a problem, then a simple implementation that ignores output recording operations can be used. Other output contracts can make other implementation choices: maintaining a backing store (a pixmap) to regenerate the output; recording the output primitives as a display list; and recording output as host display lists in systems that directly support them. Moreover, since Silica supports switching contracts at runtime, different output recording mechanisms can be selected based on changing situations.

Though support for output recording or backing stores could be built directly into the window system, building them as metalevel extensions provides a better solution for several reasons. First, new implementations and semantics for this extension can be explored not just by the implementor of the window system, but also by users. Furthermore, the extensions can be used in cases where they are indeed useful without complicating the base level window system or increasing the builtin overhead for all users. Finally, there are a large number of other possible extensions which may be of equal value to other users. Supporting all such extensions would make any single interface and implementation extremely complex and even then, the single interface could not address the range of needs that an open implementation could.

The metalevel interface provides a separate level for introducing new abstractions. Just as 3-Lisp allows constructs like `catch` and `throw` to be implemented as metalevel abstractions rather than being built into the language, Silica allows constructs that would ordinarily be built into the window system to be introduced as implementation extensions. The primary advantage of having two separate levels for building such libraries is that each of the levels can be used to implement different portions of the required behavior. Window system metalevel code is for handling issues that window systems typically deal with, and base level code handles application computation.

5 Discussion

Many object-oriented languages that support reflection on their implementation—including CLOS, 3-KRS, and ObjVLisp—represent their metalevel as objects in the same object-oriented language as used at the base level. This metacircularity is advantageous for several reasons. The reason of primary concern here is that the benefits object-oriented programming provides to the user at the base level can be equally valuable at the metalevel. For example, the user can localize changes to the implementation to specific base level objects or to specific aspects of base level objects.

Silica's metalevel, also, is implemented using the same object-oriented language as is used at its base level interface. However, unlike the reflective object-oriented languages mentioned above, Silica is not metacircular. Silica's metalevel is not written in the base level language that it provides, nor does it even make sense to write a window system in the "window system language" it implements.

Silica's use of object-oriented programming raises the following question. What distinguishes Silica's design from that of other object-oriented systems? After all, any object-oriented implementation can be said to provide a representation of its implementation since it contains objects that provide implementation methods in addition to interface methods. However, object-oriented programming in itself does not guarantee an open implementation. For example, neither the Smalltalk window system nor the Symbolics window systems had the expressed goal of exposing aspects of the window system implementation, though to a certain degree both do.[6] Rather, object-oriented programming is a technology that is particularly well suited to our purpose; three aspects of object-oriented languages, in particular, are relevant.

1. Object-centered specification of behavior allows building an implementation that closely maps onto an understanding of the important design and implementation issues. Object-oriented languages are similar to knowledge representation languages in this regard.

2. Polymorphism helps partition the world in a manner that allows multiple implementations to peacefully coexist. This means users can implement their own version of the system without disturbing existing versions of the system used by other users.

3. Inheritance provides a powerful mechanism for incrementally specifying new or different behaviors, such that clients can reuse portions of the standard implementation or incorporate stock behaviors from available metalevel libraries.

The framework of implementational reflection provides a particular conception of how to build a malleable system and to some degree a prescription that guides the use of object-oriented programming. Many of the benefits generally attributed to object-oriented programming are, in fact, benefits arising from opening up implementation or more precisely structuring the implementation well enough to allow users to benefit from access to it. Many object-oriented techniques can be cast more specifically in terms of how they give access to a system's implementation.

A salient example of this is the layering of object-oriented protocols, which greatly increases their utility. Layering involves elaborating the substructure of a protocol by specifying auxiliary functions that are invoked by the protocol to perform subtasks within the protocol's overall task. The various layers of a protocol can provide, on one hand, differing degrees of predefined behaviors (and hence structure or functionality), and, on the other, greater latitude in specializing or recombining behaviors at varying degrees of granularity. However, layering has costs and consequences. The process of layering a protocol is exactly the process of refining, and hence further constraining, a system's implementation. Hence, the definition of a layered protocol has to take into account concerns of effective implementation as well as potential utility. Many of these issues are discussed by [KdRB91] in the context of the CLOS Metaobject Protocol and CLOS implementation.

A related point is that providing explicit representations of any aspect of a system's implementation may have consequences for the system's efficiency. Lazy reification or reification on demand is a typical strategy used for making implementation state explicit. This approach can help ensure a "Don't use, don't lose" policy.[7] However, there is still potentially a disadvantage in this area if a promise to reify some aspect of the implementation has a constant or continual cost.

Layering a protocol is, in fact, a form of reification. It involves making explicit various internal places for attaching or installing tailored behaviors or decision-making machinery. An interesting observation is that lazy reification techniques for layered protocols can be implemented using the reflective capabilities of an underlying object-oriented language. In CLOS, this involves developing optimization techniques that bypass entire subprotocols in cases where it is known the standard behavior applies. Such techniques are in fact used in later versions of the PCL implementation of CLOS[KR90].

[6]Similarly, Smalltalk or Flavors do not give the same level of access to the object system implementation as CLOS does.

[7]Of course, a "Use, don't lose" policy is also important.

Conclusion

In this paper, we have argued that reflection can provide a conceptual framework for building not just programming languages but malleable systems of all kinds. A typical consequence of broadening a framework is that it can start to lose some of its resolution and hence may not seem much different from some other set of concepts. Many of the ideas discussed, especially in the terms used here, are probably familiar. Many systems do, in fact, open up aspects of their implementation. For example, the Mach operating system allows users to write code that participates in an open-ended way in decisions regarding secondary storage management and page replacement[YTR+87].

What, then, is the benefit derived from the framework of implementational reflection? Though, we ourselves are not yet convinced that it is the only or even the best way of thinking about systems that open up their implementation, we do believe that, for the time being, it provides intellectual scaffolding in a number of significant ways. First, explicitly focusing on the metalevel as a separate and first class interface to export to the user forces a greater attention to exposing important design and implementation choices. For example, Silica carefully separates the various aspects of windows—its roles as an output surface, as an input stream, and as participant in two different windowing relationships. Second, this benefit is also exported to the client. The disciplined division of the window system into its base level and a partial implementation as specified by a metalevel interfaces provides the programmer with two separate levels for introducing abstractions.

Perhaps most important in the long run is that the separation of the implementation methods into a metalevel allows a more radical shift from procedural reflection to declarative reflection. Currently, a Silica client programs at the metalevel by specializing various methods on their own contract types (i.e. in the same way that they program at the base level). A natural next step is to move from this procedural specification of metalevel statements to one that is more declarative. For example, users could state their requirements for a window system in a higher level language, and the system could then automatically pick or even construct appropriate contract types. Though the causal connection requirements are hard to meet in such a system, and some may argue should be loosened, such an approach promises great value.

Acknowledgements

I am greatly indebted to Richard Burton, Gregor Kiczales, and John Seely Brown who have actively (and patiently) supported my work on Silica in many ways. Other colleagues at PARC, especially Jim des Rivières, Mike Dixon, John Lamping, Luis Rodriguez, and Brian Smith, have helped develop these ideas on reflection and their presentation in this paper.

References

[ANS85] ANSI, New York, NY. *Graphical Kernal System (GKS) Functional Description*, 1985. ANSI X3.124-1985 and ISO 7942.

[BKK+86] D.G. Bobrow, K. Kahn, G. Kiczales, L. Masinter, M. Stefik, and F. Zdybel. CommonLoops: Merging Lisp and Object-Oriented Programming. In *OOPSLA '86 Conference Proceedings, Sigplan Notices* **21***(11)*. ACM, Nov 1986.

[BKM+80] R. Burton, R. Kaplan, L. Masinter, B. Sheil, A. Bell, D. Bobrow, L.P. Deutsch, and W.S. Haugeland. Papers on Interlisp-D. Technical report, Xerox PARC, 1980.

[Bow86] K. Bowen. Meta-level Techniques in Logic Programming. In *Proceedings of the International Conference on Artificial Intelligence and its Applications*, 1986.

[CL90] Paul Calder and Mark Linton. Glyphs: Flyweight Objects for User Interfaces. In *Proceedings of the ACM SIGGRAPH Symposium on User Interface Software and Technology*, pages 92–101. ACM Press, Oct 1990.

[Coi87] P. Cointe. Metaclasses are First Class: the ObjVlisp Model. In *OOPSLA '87 Conference Proceedings, Sigplan Notices* **22***(12)*. ACM, Dec 1987.

[Dav80] R. Davis. Meta-rules: Reasoning about Control. *Artificial Intelligence*, 15, 1980.

[HHG90] Richard Helm, Ian M. Holland, and Dipayan Gangopadhyay. Contracts: Specifying Behavior Compositions in Object-oriented Systems. In *OOPSLA/ECOOP '90 Conference Proceedings, Sigplan Notices* **25***(10)*. ACM, Nov 1990.

[IC88] M.H. Ibrahim and F.A Cummins. KSL: A Reflective Object-Oriented Programming Language. In *Proceedings of the IEEE Computer Society International Conference on Computer Languages*, 1988.

[KdRB91] Gregor Kiczales, Jim des Rivières, and Daniel Bobrow. *The Art of the Metaobject Protocol.* MIT Press, 1991.

[KP88] Glenn Krasner and Stephen Pope. A Cookbook for Using the Model-View-Controller User Interface Paradigm in Smalltalk-80. *Journal on Object-Oriented Programming*, August/September 1988.

[KR90] Gregor Kiczales and Luis Rodriguez. Efficient Method Dispatch in PCL. In *Proceedings 1990 ACM Conference on Lisp and Functional Programming*, pages 99–105. ACM, June 1990.

[Lie86] H. Lieberman. Using Prototypical Objects to Implement Shared Behavior in Object-Oriented Systems. In *OOPSLA '86 Conference Proceedings, Sigplan Notices* **21***(11)*. ACM, Nov 1986.

[LP91] Wilf LaLonde and John Pugh. *Inside Smalltalk: Volume II.* Prentice-Hall, Inc, Englewood Cliffs, NJ, 1991.

[Mae87] P. Maes. Concepts and Experiments in Computational Reflection. In *OOPSLA '87 Conference Proceedings, Sigplan Notices* **22***(12)*. ACM, Dec 1987.

[Mae88] Pattie Maes. Issues in Reflection. In P. Maes and D. Nardi, editors, *Meta-Level Architectures and Reflections.* North Holland, 1988.

[Ope89] Open Software Foundation, Cambridge, MA. *OSF/MOTIF Manual*, 1989.

[RYD91] Ramana Rao, William York, and Dennis Doughty. A Guided Tour of the Common Lisp Interface Manager. *Lisp Pointers*, 4(1), 1991.

[SG86] R.W. Scheifler and J. Gettys. The X Window System. *ACM Transactions on Graphics*, 5(2), 1986.

[Smi82] Brian Smith. Reflection and Semantics in a Procedural Language. Technical Report 272, Massachusetts Institute of Technology, Laboratory for Computer Science, Cambridge, MA, 1982.

[Smi84] Brian Smith. Reflection and Semantics in Lisp. In *Proceedings of the 1984 ACM Principles of Programming Language Conference*, pages 23–35. ACM, Dec 1984.

[Sun86] Sun Microsystems, Mountain View, CA. *Programmer's Reference Manual for the Sun Window System*, 1986.

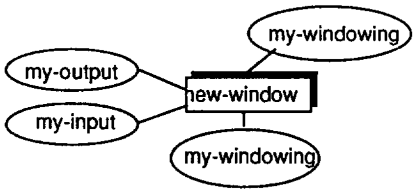
[Sun87] Sun Microsystems. *NeWS Technical Overview*, 1987.

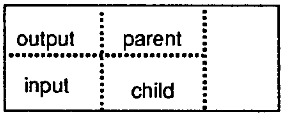
[Sym] Symbolics, Inc, Burlington, MA. *Programmer's Reference Manual Vol 7: Programming the User Interface.*

[Wey80] Richard Weyhrauch. Prolegomena to a Theory of Mechanised Formal Reasoning. *Artificial Intelligence*, 13, 1980.

[WY88] T. Watanabe and A. Yonezawa. Reflection in an Object-Oriented Concurrent Language. In *OOPSLA '88 Conference Proceedings, Sigplan Notices* **23***(11)*. ACM, Nov 1988.

[YTR+87] M. Young, A. Tevanian, R. Rashid, D. Golub, J. Chew J. Eppinger, W. Bolosky, D. Black, and R. Baron. The Duality of Memory and Communication in the Implementation of a Multiprocessor Operating System. In *Proceedings of the 11th Symposium on Operating System Principles.* ACM, Nov 1987.





```
(define-window-class new-window-class
        (extra-behavior :parts)
    (... <slots> ...)
  (:youth-contract 'my-windowing-contract)
  (:adult-contract 'my-windowing-contract)
  (:output-contract 'my-output-contract)
  (:input-contract 'my-input-contract))
```

```
(defclass new-window-class
          (extra-behavior
           window
           my-child-part
           my-parent-part
           my-output-part
           my-input-part)
     (... <slots> ...))
```

Figure 6: The conceptual model as shown in the code and picture on the left is actually implemented in CLOS in the manner shown on the right. Implementational part classes (provided by the contracts) are inherited by a window class. However, because of various features of CLOS, no power is actually lost by this choice.

A Implementation of Contracts in CLOS

An inheritance-based strategy is used to allow contracts to provide implementation methods to their windows in the current CLOS-based implementation. Contracts, though conceptually separate objects, are not allocated separately from the windows that use them. Rather contract implementation parts (i.e. classes defined by a contract implementor) are inherited by or "mixed-in" to the window objects. In the case of a windowing contract, different implementation part classes are mixed in depending on whether the window is to be a parent or a child controlled by that contract.

The essence of this strategy for the case of a statically defined combination of contracts is depicted in Figure 6. In addition, a window class can be constructed dynamically by `make-window` which takes the same keyword options as the `define-window-class` form. Automatically constructed classes are cached so that subsequent attempts to use the same combination of contracts will reuse them.

This implementation strategy was appropriate for CLOS, since CLOS does not directly support delegation[Lie86], but it does support changing an object's class dynamically and constructing classes at runtime. This strategy avoids the inefficiency of allocating several objects as opposed to a single larger one and of trampolining functions from windows to their contracts. CLOS's runtime class construction and `change-class` allow selecting and changing contracts at runtime. Furthermore, since CLOS implements classes as objects, contract classes can be manipulated by Silica's metalevel interface.