From the Broad Notion of Reflection to the Engineering Practice of Object-Oriented Metalevel Architectures.

Ramana Rao
Xerox Palo Alto Research Center
3333 Coyote Hill Road; Palo Alto, CA 94304
Email: rao@parc.xerox.com

Position paper for the OOPSLA-91 Workshop on Reflection and Metalevel Architectures

The chief position I take in this paper is motivated by my own personal experiences in building the Silica window system. My interest in this project was more in the process of building a malleable system using object-oriented programming than in the details of window systems per se, though the project did in the end require a deep understanding of the domain (afterall, all of us who have a firm, though perhaps inarticulated, grasp of reflection know that reflection doesn't let the system designer off the hook of careful, informed design).

Given my background and requirements, much of my early formulation of what I was doing on the Silica project attempted to characterize my intuitions and experiences in the language of more traditional system-building and focused on what the medium of object-oriented programming offered in this enterprise. Though the metaobject approach of CLOS had from the beginning been my most significant model, only quite late in the project did I fully embrace reflection as a descriptive framework. My paper at ECOOP91 represents the culmination of this effort. In the end, though I feel the intellectual scaffolding provided by the reflection framework is of value in itself, many computer science practioners are less willing to pay system-building money for what is perceived as philosophizing about intangibles.

This leads me, for the sake of argument perhaps, to adopt the following position:

> The OOPSLA Reflection Workshop as a series should devote as much attention to highlighting and encouraging the development of cross-domain, but specific, engineering techniques.

Another way of saying this is that I feel it is time to generate a playbook of specific techniques to supplement the broad reflection game plan. Furthermore, I posit that object-oriented metalevel architectures (OOMLA) is a particular case of reflection that can in a more-or-less immediate timeframe offer quite concrete and specific techniques that can be applied across subject matter domains.

Of course, having made this call for focus on the engineering practice of OOMLA, I (remaining at a metalevel) will step out into the broad context and come back at this specific position. Afterall,

unless we keep the broad context of reflection in mind, and place more targetted explorations accurately within the field, we will not be able to reflect and adjust where we invest our engineering talents. To make my approach at the stated position, I will start with a broad definition of reflection and specialize it in successive interations to reach the OOMLA case.

Reflection is, if we are to do justice to the spirit of the word, a very general concept. Smith's definitions of reflection from last year's workshop strives to do this justice:

> An entity's integral ability to represent, operate on, and otherwise deal with its self in the same way that it represents, operates on, and deals with its primary subject matter

Most accept that reflection plays a central role in human adaptability, and so naturally, if we are to build malleable software systems, it make sense that they too should be endowed with this ability. To this end, a number of points on what it means for a system to reflect as in Smith's definition need clarification. For example, the boundary between a system's self and its subject matter is problematic. A more specialized and by-now familiar definition of computation reflection that helps in resolving many issues goes as follows:

> A reflective system must provide (i) an account of its own operation which (iia) accurately represents its current state and which (iib) can be manipulated to affect its future operation.

A system is said to reflect when it shifts from computing about its primary subject matter to examining or manipulating the so-called casually-connected self-account of this definition. This shift of focus delineates activities about a system's primary subject matter from those that are about its self. Furthermore, this shift in the system away from its usual operation in service of fulfilling its mission is evocative of the human reflective act of stepping back to consider matters from a higher level in the hope of being able to more effectively select a future course of action.

This shifting act between the base level and a higher level, often called a metalevel, provides grist for our next iteration on specializing the definition.

> A metalevel architecture provides two interfaces to two distinct, but linked, levels of functionality, a base level interface that provides domain functionality and a metalevel interface that allows manipulating, regulating, or otherwise influencing the behavior, the form, or the implementation of the base level interface.

A system with a metalevel architecture meets the two requirements of reflective systems. A metalevel interface in essence specifies a self-account of the base level system that can be used to monitor or alter base level functionality. In many metalevel system, (e.g. CLOS and Silica), the relationship of the metalevel to the base level is, to the first order, one of implementation (i.e. the metalevel interface exposes details of how the base level interface is implemented). The nature of implementation straight-forwardly meets the causal-connection requirement of reflection.

However, note in our definition, we have loosened somewhat the requirement of causal-connection. The main point made by Smith in his introductory presentation at last year's workshop is that he believes that we should put more effort into exploring the fertile middle ground between systems at one end that provide no causal-connection between their metalevel and their base level

(e.g. representational systems) and systems at the other that strictly fulfill the causal-connection requirement. This suggestion seems particularly reasonable in the context of concurrent or distributed reflective systems, where obtaining completely connected representations of the current state is inherently problematic because of issues like synchronization (e.g. uncertain global ordering because of parallel activities) and communciation (e.g. uncertain delays because of distribution).

Also note that reflection does not require that the metalevel interface use the same language as provided by the base level (If this were the case, then only programming languages could be considered reflective). Though Smith's broad definition does use the phrase "in the same way," he meant that the capacity to reflect should be equivalent in power to the capacity to perform in the primary domain, in particular the power afforded by full-fledged programmability. This equivalence of power distinguishes systems that provide a few predefined reflective operations from those that allow end users to write reflective programs. (Admittedly, this distinction is really a continuum, but for the time being we can leave this issue unexamined).

Now, we are ready to introduce object-oriented programming into the picture:

> A system has an object-oriented metalevel architecture if it is a metalevel system that specifies its metalevel interface as a set of objects and protocols that implement some aspects of the base level interface.

An OOMLA system does not require that a base level interface be object-oriented, but rather exploits object-oriented programming at the metalevel interface to provide a usable and useful self-account. To the first order, an OOMLA carves up the implementation of a base level interface as an object-oriented system. By so doing, the OOMLA system provides a self-account which can be adjusted by the end user to build other systems of a kind with it, and thus the OOMLA system is inherently more adaptable.

One attribute of reflection that appears to get lost in the shuffle of this account is the dynamic act of reflection itself, the shift of attention, the stepping-back to take stock from a higher level. Interestingly enough, the reflective act seems to shift from within the software system proper to the overall system that includes the designers, the software, and the end users. End users are allowed to reflect upon the appropriateness of the software as it sits in their context and moreover tailor the software to the needs of their application.

This story doesn't put OOMLA's as far off the main track of reflection as it may appear at first glance. Though it diminishes the role of runtime decision making by the software system itself, it does represent a half-way point toward the goal of making software selection decisions automatically. Characterizing the implementation of a system as metalevel objects and protocols highlights the aspects of the system that can be usefully and, with a little luck effectively, varied. No system can possibly expose all possible aspects at all time. For example, it makes sense that their may be more than one partial self-account that is useful in different circumstances (The thread of work on ABCL/r, ACT/r, ACT/r2 illustrate how reflection on different aspects lead to different forms of metalevel accounts).

The issue of richness or programmability of the metalevel arises for OOMLA's as well. The combinatorics of a carefully-orthogonalized object-oriented representation of implementational and design issues afford productivity in the generation of alternative systems. In essence, just as programming languages allow their users to write programs unanticipated by the language designer, object-oriented metalevel systems allow their users to create systems with features and implementations unanticipated by the system designer.

We can now return to the point where it makes sense to consider what cross-domain, but specific, techniques can be developed that would provide engineering support to those that embrace the broad framework of reflection and the particular formulation of OOMLA. To conclude this paper, I will briefly turn to an area that I believe is a particularly ripe area for such techniques, namely, object-oriented protocols.

The specification of particular metaobjects and protocols probably can't be generated in a domain-independent manner (though we are approaching a stage where we are ready to codify a set of loose guidelines on how to perform this process). However, a characterization of protocols in general can inform the design of programming tools for designing and utilizing protocols and of efficient techniques for implementing protocols. Furthermore, I believe that we are ready to develop techniques for specifying, annotating, documenting, and optimizing object-oriented protocols.

As a short example, consider how a layered object-oriented protocol can be optimized in an OOMLA system. The CLOS Metaobject Protocol defines a protocol which is invoked to select a method for a particular generic function invocation. This protocol specifies a set of generic functions that are invoked to implement the selection process. This presents two problems. First, it means that for every user generic function invocation result in a number of system generic function invocations. Second, if all method selections were really implemented by this protocol then the system would recurse infinitely trying to select a method for the top level function of this protocol (Jim des Riveries observed that this is CLOS's reflective tower in his paper at last year's reflection workship).

Thus, this CLOS protocol seems to violate a principle of "Don't use, don't lose." However, in a real implementation of CLOS, this method selection protocol is only invoked in cases where it would make a difference to invoke this protocol (just as the metacircular tower of 3-Lisp only exists to the extent necessary). So, when your reach a standard case method dispatch, the standard, hopefully efficient, implementation is used, and the virtual recursion of the protocol is capped. (The relationship in these case isn't strictly implementation, but rather the promise to implement if necessary. This is why above I have made the "implementation to the first" order comment a couple of times.)

A technique for accomplishing grounding out of the protocol is absolutely necessary in CLOS, but it would also be useful in any system written in CLOS for eliding protocols when they aren't being used (e.g. Silica).

The class of techniques that I imagine go far beyond the techniques currently used in existing object-oriented systems and language implementations. The problem is that most existing techniques focus on single classes or single generic functions, rather than on sets of classes or the protocols by which they interact. For example, eliding the code for a method dispatch when only one method exist is a specific technique that is familiar in the object-oriented community, but this doesn't come close to addressing the eliding of whole subprotocol because it would make no difference to invoke it.

The example presented is just one example of what I mean by specific techniques. Over the next few months, I intend to work out a more complete account of this area of work and of specific techniques that fit within this account.