Applications of Telos

PETER BROADBERY
CHRISTOPHER BURDORF

School of Mathematical Sciences, University of Bath, Bath BA2 7AY, United Kingdom

Keywords: Eulisp, Telos, persistence, finalization, virtual shared memory

Abstract. EULISP has an integrated object system with reflective capabilities. We discuss some example applications which use these facilities to experiment with some advanced and powerful concepts, namely, finalization, virtual shared memory and persistence. A secondary goal is to attempt to illustrate the additional possibilities of metaobject programming over non-metalevel techniques.

1. Introduction

EULISP [24] provides an object system (called Telos [4]) which is fully integrated with the rest of the language, and includes a meta-object protocol (MOP) [19] which allows programs to reflect on the structure and inheritance relationships between classes. Reflection here is the process of taking a system object, such as a class and transforming it into a user-level object which a program can read and perhaps modify. Using this structure, a program is free to change the representation and computation of these aspects to obtain new behaviour by subclassing existing classes and metaclasses. These new classes have the same status as the system-defined classes, so the extensions become a part of the original language—no special code is needed to use them.

This paper comes in five parts. First we introduce the Telos MOP from a user's point of view, with a particular emphasis on the slot access protocol. After this we look at a fairly simple extension of the slot access protocol to add finalization (section 3). Two larger scale applications are virtual shared memory model (section 4) and persistent objects (section 5). The last section (6) differs from the earlier ones in that instead of being illustrative fragments of meta-object programming, it describes a complete application written using EULISP and Telos which depends heavily on Telos in order to create an interface into an object store for simulation programs. A secondary aspect of this last part is that the system in question was originally written in CLOS and had first to be ported to EULISP before the
extensions described here were made.

2. Meta-object protocols and Telos

A detailed description and rationale for the design of Telos appears in [4]. Two principles of the design are: (i) a program should not pay for the cost of a feature that it does not use (also known as "don't use, don't lose"), (ii) as large a proportion as possible of the meta-object level operations should be done when classes are created rather than when they are used—in effect, a form of compile-time versus run-time tradeoff. A consequence of this second property is that the creation and access routines can be extended without imposing overheads on other programs and with minimal overheads on the client program.

There are four components to the Telos MOP:

- Class definition and inheritance;
- Slot accessor creation and invocation;
- Generic function dispatch;
- Object allocation and initialization.

Each of these consists of a number of generic functions which have defined semantics, and are guaranteed to be called at specific points in the protocol. Of course, these protocols are not entirely separate—each relies on the existence of the other three to function, but the actual details of each protocol are largely independent of one another. New behaviour is obtained by subclassing the classes specified in the definition, and specializing the appropriate parts of the MOP for the new classes.

2.1. Specializing slot access

The Telos slot creation and access protocol [4] differs from the CLOS [19] protocol in a number of important ways, but primarily, the balance of work is shifted from the access protocol to the creation protocol.

The slot accessor creation routine has four phases (Figure 1):

- Create slot-description objects;
- Finalize the details of the object representation;
- Create the slot accessor functions;
- Create specialized slot accessors.