LISP AND SYMBOLIC COMPUTATION: An International Journal, ?, ??–??, 1993

Applications of Telos

PETER BROADBERY (*pab@maths.bath.ac.uk*)
CHRISTOPHER BURDORF (*cb@maths.bath.ac.uk*)
School of Mathematical Sciences, University of Bath, Bath BA2 7AY, United Kingdom

Keywords: Eulisp, Telos, persistence, finalization

Abstract. EuLisp has an integrated object system with reflective capabilities. We discuss some example applications which use these facilities starting with relatively simple extensions, such as the addition of new arithmetic classes, and moving to more advanced meta-object programming to support 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 [21] provides an object system (called Telos [4]) which is fully integrated with the rest of the language, and includes a meta-object protocol (MOP) [17] 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 four parts. First we introduce the Telos MOP from a user's point of view. This is followed by some simple examples of the use of generic functions and methods to change part of a EuLisp environment transparently (section 3). The reader experienced in the use of this kind of object-oriented language may wish to skip this section. After this we move up a level to discuss the use of the slot creation protocol to add finalization in a Telos based system (section 4.1), and how to implement a virtual shared memory model (section 4.2). The last part 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 a transparent 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 then 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 specialising the appropriate parts of the MOP for the new classes.

3. Extending EuLisp arithmetic

In common with many other languages (for example, C++ [25]), EuLisp provides the ability to define new types of number, and allows the programmer to add new methods on the standard numeric operators.

Any extensions to the number system should take into account the concerns of transparency, efficiency and locality. The first implies that new numeric types should be consistent with the existing classes (integer and float), so that existing generic operations, such as `+` are well-defined, as well as of polymorphic operators, such as `double`. It is unacceptable to

```
(defmethod binary+ ((x <number>) (y <number>))
   (let ((class (or (lift x (class-of x))
                    (lift y (class-of y))
                    (error "Can't lift numbers"
                           <arithmetic-condition>
                           'error-value (cons x y)))))
      (binary+ (convert x class)
               (convert y class))))
```

Figure 1: Number lifting in Feel

be forced to rename operations, or to rule out operations which rely only on the semantics of the standard operations (i.e., not on the properties of the individual numeric types). The second concern is that new classes should run as efficiently as the standard classes—overhead for calling the operations should be no more than for calling generic functions. Finally, the use of new arithmetic types should not prohibit standard optimisations on the normal types, and should not affect the semantics of existing code. The first of these can, in general, only be solved with a compiler [18], while the second implies that new numeric methods should not be excessively general. That is to say, methods should not be defined on binary functions which do not specialize *both* arguments, so that the new method does not affect the applicable-method list of other calling domains.

3.1. Numeric conversion

When a binary operation is called with arguments of different types, and no special-purpose method has been defined, standard method dispatch rules imply that the method on `<number>`×`<number>` is called (assuming that all methods are specialized by both arguments having the same class). In the Bath implementation of EuLisp, Feel [23], the most general method attempts to lift both numbers so that one of the other methods can cope with the operation. A function called `lift` is defined for numbers, and works in a similar way to the standard function `convert`, but returns a class as its result, or `nil` if the operand is of a more general type. See figure 1.

3.2. Modular Arithmetic

This example (see figure 2) shows how classes can be instantiated at runtime in order to allow types to be created "on the fly". When a new mod n number is created, and no mod n number has been created before,

```
(defmodule zmodn
  (import (eulisp-level-0) syntax (eulisp-level-0))

  ;;Metaclass (somewhere to hold values of n)
  (defclass <zmodn-class> (class)
    ((n initarg n reader zmodn-n))
    class <class>)

  (defclass <zmodn-object> (number)
    ((z initarg z accessor zmodn-z))
    class <zmodn-class>)

  ;;Constructing new zmodn class
  (defun make-zmodn-class (n)
    (make <zmodn-class> 'n n
          'direct-superclasses (list <zmodn-object>)
          'name (make-symbol (format nil "zmod-~a" n))))

  (defun find-zmodn-class (n)
    ;;implements the memoization of class creation
    ...)

  ;;i mod n
  (defun make-modular-number (i n)
    (make (find-zmodn-class n) 'z i))

  ;;NB: Not efficient, but means that remainder
  ;;only appears once in this module.
  (defmethod initialize ((proto <zmodn-object>) lst)
    (let ((z (call-next-method)))
      ((setter zmodn-z) z
       (remainder (find-key 'z lst) (zmodn-n (class-of z))))
      z))

  ;;example method
  (defmethod binary+ ((i <zmodn-object>) (j <zmodn-object>))
    (if (eq (class-of i) (class-of j))
        (make (class-of i) 'z (+ (zmodn-z i) (zmodn-z j)))
      (error "incompatible moduli" zmodn-error)))

  ...
)
```

Figure 2: modular numbers in EuLisp

```
(defmethod (lifter <uni-poly>) ((x <integer>))
  <uni-poly>)

(defmethod (converter <uni-poly>) ((x <integer>))
  (make <uni-poly> 'coeff x 'degree 0))
```

Figure 3: Lift methods for polynomials

a new class is instantiated, so that it is easy to check the compatibility of objects. This implementation reduces the space required for instances by storing the value of n in the class.

3.3. Polynomial Arithmetic

A univariate polynomial class is relatively simple to write, but in addition to the arithmetic involving only polynomials it is also meaningful to consider mixed mode arithmetic on integers and polynomials. We therefore need to define a `lift` method for integers to polynomials, and a corresponding `convert` method (see figure 3). The lift method ensures that we do not need to define a method for each combination of arithmetic class.

The `<number>`×`<number>` method of figure 1 is not very efficient. However, it can be optimized as follows: whenever a arithmetic operation is called with a new signature, and the `<number>`×`<number>` method is therefore called, carry out the target class lookup as in figure 1, but this method should also create a new method on the generic function which itself does the appropriate conversion. This avoids the lookup of the target class each time the function is called with a particular signature. Clearly this trades time for space, where the latter will depend on the number of subclasses of number in the system.

A problem with this approach is that it does not automatically convert through intermediate types; for instance, if we added a multivariate polynomial class, we would need to define lift methods for both integers and standard polynomials, even though one can convert from integers to multivariate polynomials via univariate polynomials. Additional protocol would need to be added to the lifting functions so that possible routes could be searched if this functionality was desired.

4. Redefining slot access

The Telos slot creation and access protocol [4] differs from the CLOS [17] protocol in a number of important ways, but primarily, the balance of work

```
compute-inherited-slot-descriptions
compute-slot-descriptions
compute-and-ensure-slot-accessors
    ensure-slot-reader
    compute-primitive-reader-using-class
        compute-primitive-reader-using-slot-description
    ensure-slot-writer
    compute-primitive-writer-using-class
        compute-primitive-writer-using-slot-description
```

Figure 4: slot access protocol

is shifted from the access protocol to the creation protocol.

The slot accessor creation routine has four phases (figure 4):

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

Each phase allows the programmer to specialize the slot in different ways: add extra slots to the class in the first and move slots and allocate space for hidden slots in the second. However, the last two are probably the most commonly used: one can change the function used to dispatch slot access in the third phase, whilst the last enables arbitrary functions to be called at slot access time.

Where the slot description is an instance of `<local-slot-description>` (the default case), accessing a slot is just an indexed reference or update operation. However, this can be specialized at will. The flexibility of this approach can be used to build complex systems from the primitive functions provided by EuLisp.

4.1. Finalization

In many applications it is required to do some post-processing when it can be established that an object will no longer be needed. This process is known as finalization [16]. For example, many systems require that `close` should be called on a file object before a program is exited, otherwise the stored version of the file may be inconsistent with the version held in a buffer by the operating system or application program. The problem is that

```
(defclass <file> ()
   ;;create a class with a slot to be used in the finalization method
   ((file initarg file
          accessor file-internal
          slot-class <finalizable-sd>))
   class <finalizable-class>
   initargs (open-args)
   constructor (open-safe-file open-args))

  ;;open a file and set the actual handle
(defmethod initialize ((x <file>) lst)
   (let ((new (call-next-method)))
     ((setter file-internal) new
      (apply open (find-key 'open-args lst)))
     new))

;;tidy up the file
(defmethod finalize ((x <file>))
   (close (file-internal x)))

;;various methods to allow writing to this
;;class of object, plus print methods, ...
...
```

Figure 5: Finalization of a file handle

a program may lose a pointer to such an object, and never be able to run the finalization code on it before the object is recycled by garbage collection, after which finalization is impossible. To circumvent this situation, we need to be able to to note when an object becomes inaccessible, recover its slots from some "hidden" storage and invoke a tidying operation on the object.

The Bath implementation of EuLisp provides two extensions to the language that simplified the implementation: the system allows the user to install a function which is called directly after each garbage collection (a post-GC hook) and secondly a new class of object—weak wrappers. It is guaranteed that the post garbage collection function is never called during the execution of a previous finalize, and always runs on the thread which invoked the garbage collection process. The purpose of these rules is to avoid problems with infinite loops and concurrency, respectively. Weak wrappers are objects with a single slot which initially contains some object, but is set to `nil` if the object is garbage collected (during a garbage collection, references from weak pointers are not followed, therefore the referenced ob-

```
(defclass <finalisable-class> (class)
  ((count accessor finalisable-slot-count)
   (proxy accessor proxy-class))
  initargs (proxy)
  )

;;class initialization
(defmethod initialize ((cl <finalisable-class>) lst)
  (let ((cl (call-next-method)))
    (let ((slot-posn (class-instance-size cl)))
      ((setter class-instance-size) cl (+ slot-posn 1))
      ((setter finalisable-handle-posn) cl slot-posn)
      ...)
    cl))

;;instance allocation
(defmethod allocate ((cl <finalisable-class>) lst)
  (let ((handle (make-vector (finalisable-slot-count cl)))
    (obj (call-next-method)))
    ((setter primitive-slot-ref) cl
     (finalisable-handle-posn cl)
     handle)
    obj))

;;Constructing new proxy objects
(defun make-proxy-object (class values)
  (let ((new-cl (proxy-class class)))
    (let ((obj (allocate new-cl 'proxy t)))
      (mapc (lambda (sd)
              ((slot-description-slot-writer
                (find-slot-description new-cl
                  (slot-description-name sd)))
                obj
              (vector-ref values
                          (slot-description-position sd))))
        (class-slot-descriptions class))
  obj)))
```

Figure 6: Fragment of finalisation code

ject may be garbage collected). These two extensions[1], and the facilities of Telos allow the implementation of a simple finalization scheme.

The idea is to store the slot values of an object that are needed for final-

[1]One does not absolutely need the post GC callback as one could use the `wait` primitive, but the GC callback is more efficient.

ization somewhere safe, so that when the space occupied by the object is recovered by garbage collection, the values that were stored in the slots are still available. Under the protocol for this implementation of finalization, each class which needs this facility is required to nominate a proxy-class with similarly named slots for the values needed for the finalization method. The proxy class defaults to the class itself. When the original instance is garbage collected, the change of status can be detected in the weak wrapper and the slot values can be recovered. The finalization method is then executed on an instance of the proxy-class, with the instance not subject to the finalization scheme. The slot values are stored as a vector, on which the original object maintains a handle, and is also reachable via an association-list indexed by a weak pointer referencing the object to be finalized.

When an object becomes unreachable, the post-GC function instantiates the proxy-class with the remembered slot values and calls the finalization routine with the 'resurrected' object. If the proxy class is (by coincidence) subject to the finalization scheme, then an argument is passed to the `allocate` method to ensure that the resurrected object is not added to the finalization list. See figure 6.

This technique works well, but cannot finalize a cyclic structure as the values of the slots are accessible via non-weak pointers. It should be noted that it is hard to define an algorithm for finalizing circular structures which picks the 'correct' point in the cycle to break—more likely, this indicates that the structure of the objects needs to be re-thought, although an extra level of indirection can generally be used to achieve a similar effect! To handle cyclic structures properly, more information must be given to the processor about how the objects interact. Networks without cycles take n cycles to be completely finalized, where n is the diameter of the network. To do better than this the garbage collector would have to be modified to make another pass over the weak pointers after the finalization phase is complete, which could seriously affect the performance of the system when no finalization is required. Including such extra routines would complicate the garbage collection sufficiently that stock algorithms would not be able to handle it, whereas it was desired to make the system reasonably portable—every implementation of Telos so far has been on a system with some kind of weak pointer (generally provided as an extension). The other advantage of this approach is its relative simplicity—the code itself is quite short (< 150 lines), and quite readily comprehensible.

4.2. Virtual Shared Memory

Virtual Shared Memory is a software technique for simulating the shared memory of many parallel architectures on a network of processors with local

memory only. It can be viewed as an abstraction for passing data between multiple physically disjoint processes, without message passing. It involves inventing a virtual 'arena' in which objects are stored, and some interface to deal with an object's allocation and slot access. The idea is to add a mechanism whereby objects can be passed between processes without the expense of copying the objects at each communication. Such a system is useful for a variety of reasons:

distributed data structures: The processors can co-operate to form a large data structure which is accessible equally from all processors;

hides message passing: Client programs do not need to know that other processes will be asking for data in their space—so message handling code does not need to be written explicitly;

more familiar programming model: The concept of a number of objects interacting via shared memory is more familiar to the programmer than disjoint memory spaces.

On the other hand, such a system does have its drawbacks—the relative sloth of a network is hidden by the abstraction, so it is easy to write slow code suffering from the delusion of uniform access cost. Consistency models may add even more inefficiency to this protocol.

The abstraction of VSM should be capable of expansion in several ways:

memory consistency: For some applications updates to objects never happen (for example if the program is totally functional), or, at the other extreme, updates must be atomic and no object must copied without an invalidation protocol. Both should be accommodated.

garbage collection: It should be possible to write a garbage collector on top of the abstraction so that deallocation is handled by the system, rather than by some ad-hoc method.

object naming: One should be able to retrieve objects by indirect mechanisms. This allows Linda-like functionality.

efficiency: The default mechanism should not have an excessive overhead for the most common cases. For example, multiple reads of the same slot, and additional classes can override parts of the protocol so that they may be yet more efficient.

The current version of VSM consists of a number of classes of object which interact with Telos to provide a simple virtual memory system and support for protocols such the system could be extended to support more of the other mechanisms listed.

4.2.1. Layout of memory

The allocation of memory closely mirrors that of the underlying Lisp system—objects are allocated from pages (a fixed-size group of slots or objects) which are in turn allocated by a distributed allocator which tracks memory usage to ensure that it does not swamp the rest of the system. The pages store either atomic data, such as instances of strings, symbols and numbers, or addresses of instances of other objects stored in the VSM system.

4.2.2. Implementation

The classes used in the implementation are designed to be subclassed, and provide an extension to Telos to encompass allocation strategy, garbage collection, and distribution of data. The implementation is made up of four classes of object:

page: Actually holds the information. These are sent atomically through the distribution layer.

address: Contains a page pointer and an offset. Used to reference objects.

handle: The part of an object used to store its address.

object: Seen by users of the VSM code.

The VSM system consists of a protocol which new page and address classes can specialize, plus a number of implementations of these abstract classes. Normal (application) code does not need to be aware of this protocol, although one can access it if new classes are added. One change from standard semantics is that objects may not appear to be `eq` to themselves because of caching arrangements (a page may leave the local processor and return). The function `eql`, which has an appropriate method to compare VSM addresses must be used instead.

Pages are held in caches on local processors with a reference to the page's owning processor, and when a page fault occurs the system then queries the page's owner about its location. Once found, the page is copied to the processor.

Other page lookup mechanisms are perfectly possible—a message can be sent to the owner of the page on every request, and the owner replies with the appropriate object. Other mechanisms can be supported by the protocol and work is in progress exploring the advantages of some of these.

Without a means of starting remote threads, VSM is not especially useful. Currently the system uses a version of futures [15], although a paralation [2] [24] implementation has also been developed. The underlying interprocess

communications mechanism is PVM [13], although this is transparent to the rest of the system.

The system is strictly experimental and has been designed to permit experiments to be made on the efficiency and interaction of various consistency protocols, page caching and replacement algorithms, and garbage collection. A persistent storage module will also be added so that one can employ both temporal and spatial persistency in a system.

5. Persistence

Persistence has also been explored as a topic in its own right as a means to support large-scale object-oriented simulation in EuLisp. This section discusses that experience.

Persistent object systems (POS) [1] provide a seamless integration between a programming language and a database. The POS requires a cache to hold objects which have been loaded into primary memory to avoid the need for reloading an object each time it is accessed. The persistent object cache can be viewed as similar to the working set in a virtual memory system. The size of the cache has to be limited so as not to swamp the runtime system with more objects than can exist without exceeding the size of swap space. Also, since objects may be shared with other users, it is not desirable for any one user to have control over too many objects at a given time, and therefore, caches can also be useful to limit the number of objects owned by a user.

Some of the advantages of persistent systems listed by Morrison and Atkinson [20] include:

1. reduced complexity;
2. reduced code size and time to execute;
3. data outlives the program.

Firstly, complexity is reduced for the application builders, because with persistent systems, there is no distraction for the programmer in dealing with the complexity of managing the database. He or she need only consider the complexities involved in the mapping between the programming language and the problem to be solved. Secondly, persistent systems reduce code size, because the application program need not contain code concerned with the explicit movement of data between primary and secondary memory. Also, the time to execute is reduced, because only objects required by the system get loaded into primary memory. Finally, the data outlives the program, because it resides in a database.

In this section we discuss the implementation of a such a persistent object system designed for use in simulation applications—The Persistent Simulation Environment (PSE) developed at UC Berkeley, and the problems in porting it from its original language, Allegro Common Lisp, to EuLisp.

5.1. Persistence in PSE

Persistent object systems support four major functions: sharing, maintaining, inspecting, and reusing of objects. Sharing allows the concurrent use of persistent objects by more than one application program, similar to a database management system which supports access by multiple programs. Object maintenance (insertion, deletion, and updating of simulation objects) can be performed in virtual memory during simulation processing, or through maintenance routines applied directly to objects in the persistent object repository, external to any simulation program. Objects modified during simulation processing will be transparently updated in the persistent repository so that consistency is maintained between virtual objects in the simulation and secondary storage persistent objects. Likewise, objects can be retrieved and inspected during simulation processing and at any time before or after the simulation. Finally, with a persistent object repository, simulation objects can be reused without recreating and initializing objects for each simulation trial. For simulations with thousands of objects, reusability contributes significantly to performance improvement. PSE supports three of the four functions described above; sharing of persistent objects has not been addressed because it involves issues of transaction management and is not one of our primary goals. Nevertheless, other persistent object languages are pursuing this topic and their results will contribute to the success of persistent object systems.

5.2. PSE architecture

An object which is declared to be a persistent object is retained in secondary storage after program execution terminates. In PSE, once a class has been declared to be persistent, those persistent objects are referenced identically to non-persistent simulation objects. Furthermore, fetching and instantiating a persistent object from secondary storage is performed transparently by the underlying PSE kernel. We based the kernel implementation of PSE on the Rowe's SOH (shared object hierarchy) methodology.

PSE is composed of the following components pictured in figure 7: persistent object files, object space, and an object directory. The object files store an ASCII representation of the objects in secondary storage. Object space denotes the area in main memory where the object structures reside, and the object directory contains one handle per object which maps

Figure 7: Components of PSE

an object identifier into the object handle. The object handle contains meta-information about the object and always remains in main memory. A handle includes information such as a pointer to the object's memory location (which is `nil` if the object is not in the object space), the object's location in the object file, whether or not the object has been modified, and the object's update mode. The update mode indicates how the object will be modified on disk. If the mode is *direct-update* the object will be updated immediately upon modification. If it is *deferred-update*, the object will be updated when the number of objects in the object space reaches capacity thereby triggering garbage collection of the object directory and updating of necessary objects. *Local-copy* objects only exist in main memory and therefore are not updated on disk.

During program execution, object handles are used as parameters to represent simulation objects. When a slot in an object is referenced, one of two actions is taken: if it is determined that the object is not in main memory, then it is fetched and instantiated before the slot value is returned. Alternatively, if the object is already in main memory, the value of the slot is simply returned. As discussed earlier, the determination of the object's location, fetching, and instantiation are handled by the persistent object system and is transparent to the programmer.

5.3. Port of PSE to EuLisp

PSE was ported to EuLisp, because of the availability of threads and the means for true concurrent execution on a Stardent Titan multiprocessor. Concurrency provides the possibility of reducing the real execution time of a program through simultaneous execution of different code segments. The port of PSE to EuLisp required a significant amount of time, because the entire interface with the object system had to be rewritten and EuLisp modules provide a more austere environment than Common Lisp's packages. In the Common Lisp version of PSE, the object system had been merged with the file system through low-level modifications. A new metaclass was created for persistent classes and objects and methods were added at the low-level to handle access and modification of these objects.

The primary problem involved in porting PSE to EuLisp is the lack of runtime binding and exportation of names. In Common Lisp this is entirely possible, while EuLisp prefers to gain run-time efficiency by eliminating these development-time programming aids. Feel, the Bath implementation of EuLisp, partially supports runtime binding, but because of the semantics of module importation, cannot handle exporting new names at runtime. This is a problem because classes stored in the persistent database also include some code for the class's methods. Several techniques were tried, and it was finally decided to use a single module to hold the application, and explicitly mark the classes that are intended to be persistent, and predefine their accessors. This forces a distinction between persistent, and non-persistent classes which was not present in the Common Lisp version. Such a distinction makes the POS more visible to the programmer, whereas the programmer should not be aware of the persistency of objects.

On a more positive note, EuLisp slot descriptions provided an elegant solution to the problem of access and modification of persistent slots. A macro called *defdbclass* was defined which created a new class with all slot-descriptions of class *persistent-slot-class*. Then, a slot access method was defined on *persistent-slot-class* to handle the specific mechanics of access and modification of a persistent slot as described previously.

The remaining elements of the port was spent dealing with technical differences between EuLisp and Common Lisp of which there are many but are not of great interest, so they will not be discussed further.

5.4. Persistence built into higher-level constructs

One of the advantages of Lisp is that it can be used as an assembly language to build higher-level constructs using macros that are tailored for specific domains such as rule-based systems, natural language process-

ing, and even simulation. It has been found to be advantageous to incorporate the previously described persistent-object facility into higher-level constructs for Petri net and connectionist simulations, because in the case of Petri nets, they can store the simulation history for later reference, and in the case of connectionism, they can store information, gained through the building and training of the network, for reuse.

5.4.1. Connectionism

Connectionist models provide a mechanism for representing knowledge through connections between neurons. Those connections are weighted to represent the certainty factors between semantic relationships. Due to the recent increase in interest in the use of connectionist and neural systems, there has been active development in tools that support their development [9, 12, 11, 26].

POCONS [8] [7] (Persistent Object CONnectionist Simulator) is a new component added to the EuLisp version of PSE which supports object-oriented connectionist simulation. With the exception of Neula [12] other neural network tools do not support an object-oriented design methodology. Both Neula and NSL [27] have object-oriented constructs, but differ in their syntax and semantics which is unlike the widely-used object-oriented languages like Smalltalk [14], C++ [25], or CLOS [3]. In addition, POCONS is close to CLOS and Telos in syntax and semantics (thus, there should be a shortened learning curve for programmers familiar with either systems) POCONS can be used to develop hybrid symbolic/connectionist systems, since it is embedded in Lisp which has been used extensively for symbolic inference. It is also extensible, because it allows a user to create new neurons interactively and rebuild the neural network: a feature not available in the other object-oriented connectionist simulators like Neula and NSL. Also, unlike Neula and NSL, POCONS supports persistence, and it uses objects to represent relationships between different elements of the network.

POCONS is a declarative language in that the programmer simply specifies the structure of the network, enters a command to make the system build the network's internal structure, and initiates execution of a simulation.

5.4.2. Object-Oriented Connectionist Model

POCONS is based on the object-oriented connectionist model where the user does not specify any procedural information about the network's execution. The model only requires that the user specify the neurons which represent the components of the network, their attributes, and relationships between them. POCONS can then be instructed to generate a neural network. Queries can be made on the network which initiate connectionist

```
(defdbneuron hobbit (middle-earth-inhabitant)
  ((nature initform 'good)
   (height initform 'short)
   (is-fond-of initform '((birthday-parties . 1.0)
                          (swimming . -0.7)
                          (fighting . -1.0)))
   (has-enemy initform '((dragon . -1.0)))))

(defdbneuron bilbo (hobbit)
  ((is-fond-of initform '((pipeweed . 1.0) (light . 1.0)))))

(defdbneuron dragon (middle-earth-inhabitant)
  ((has-enemy initform 'dwarf)
   (nature initform 'evil)))

(defdbopposites 'nature 'good 'evil)
```

Figure 8: Fragment of POCONS code for a Middle Earth neural net

simulations.

The underlying POCONS system translates connectionist objects into sets of neurons that represent the class hierarchy and attributes. Each class has a neuron associated with it, and likewise the class neuron has weighted *is-a* links to the neuron which represents its superclass. Also, a neuron is created for each class and slot-value pair (e.g., (`nature`, `good`)) which has links to its class and the class has links to it.

Defdbneuron is the defining component for the creation of a persistent neuron and an example is given in figure 8.

The *neuron-name* will be used as a symbol that identifies the neuron. The *superclasses* specify the class or classes from which the neuron inherits. The *slots* describe the explicit relationships that the neuron will have. Slots are specified as an initialisation list containing *slot-names* and slot options.

Defdbopposites indicates a relationship between two neuron types and is defined as follows:

```
(defdbopposites slot-name neuron-name neuron-name)
```

Defdbopposites can only take as arguments *neuron-name*s used in calls to *defdbneuron*. The result of the use of *defdbopposite* will be a negative link between the two specified neurons in the network. Also, the use of *defdbopposites* generates a persistent object containing the specified information. Thus to reuse a specific network after the first time it was executed, one

need only open the database.

The algorithm which converts this representation examines each object and a neuron is created for each neuron name and for each slot attribute and value. It then creates forward links from each subclass neuron to each superclass neuron. Links are also created from class neurons to their slot neurons.

For a more extensive description of POCONS including some examples see [8], and for experiments in mapping POCONS onto SIMD and MIMD machines see [7].

5.5. Petri nets

Petri nets are widely used in the simulation of concurrent systems [22]. As a result of the popularity of Petri nets, there have been a variety of tools developed [10]. These tools include graphical editing and creation of Petri net systems. This section describes a tool for the development of Petri nets: a language called Per-trans [6]. It features the fusion of persistent object technology with Petri net development. Per-trans is a component added to the EuLisp version of PSE.

Per-trans has features that simplify the task of developing stochastic Petri net models [19]. It contains constructs that specify the places, transitions, and token locations in the Petri net. The underlying system handles all the procedural execution of the simulation. Per-trans allows Petri net components to be represented as persistent objects.

5.5.1. Per-trans components

Per-trans provides an application programmer with primitives to represent and execute simulations using the stochastic Petri net model. Per-trans has the following general features:

1. Persistence;
2. Declarative;
3. Allows embedded Lisp code.

It supports persistence, because all the various elements of the Petri net model (places, transitions, and tokens) can be represented as persistent objects. The application programmer can decide whether he or she wants some or all of the net to be persistent. It is declarative in that the programmer need not specify any of the control information used to determine when a transition will fire and send tokens throughout the net. The Per-trans defining forms generate an event-based simulation that is executed by the

underlying scheduler and simulator. The programmer need only specify the places and transitions, where they are connected, and any time delays that might exist on transitions. The internal scheduler examines places and transitions to determine whether a transition is enabled and when it should fire. It also passes tokens to places that are enabled once a transition fires. The underlying scheduler sends messages to objects that contain a time stamp for when they should execute. The underlying simulator then executes those messages at the appropriate simulation time. Finally, Per-trans allows the application programmer to embed Lisp code in the definition of specific nodes (places and transitions). The embedded code will be executed when a token moves to the node's location in the network. Such embedded code can be used to process information or produce graphical output illustrating the net's behaviour. Graphical output can be produced in the X Window System as supported under Feel [23].

Several Petri net models have been implemented using Per-trans. Per-trans has also been modified so that it produces parallel simulations [5]. The Per-trans language is explained in detail with examples in [6].

6. Conclusions and Further Work

We have discussed the applications of the Telos object system to various programming problems, and shown how it can be used to construct applications. The use of an object-oriented language encourages a toolbox of useful routines to be written which can then be combined to form a complete application. The addition of the metaobject protocol allows this idea to extend into the representations of classes as well as the interface they provide.

One of the strengths of Telos is that it is integrated into its host language, EuLisp—to a greater degree than CLOS—and can be used to change parts of the system which are commonly not part of the object system, for instance arithmetic operations and threads. This power, in combination with EuLisp's module system assists with both language extension and language embedding. However, experience with supporting persistence suggests it makes dynamic demands that are hard to reconcile with (uncharacteristically) static tendencies of this Lisp, which have been motivated by a desire to be able to deliver more efficient applications. Clearly this is an area for further work.

References

1. Atkinson, M. and Morrison, R. Persistent System Architectures. In

Proceedings of the Third Annual Conference on Persistent Object Systems, Springer-Verlag (1989).

2. Batey, D.J. *DPL - A Distributed Implementation of Paralation Lisp.* Bath Mathematics and Computer Science Technical Report, 92-60 (June 1992).

3. Bobrow, D. *et al.* Common Lisp Object System Specification. (1988). X3J13 Document 88-002R.

4. Bretthauer, H., Davis, H., Kopp, J., and Playford, K. Balancing the EuLisp Metaobject Prototcol. In *Reflection and Metalevel Architecture*, Proc. of the International Workshop on New Models for Software architecture (November 1992) 113–118.

5. Burdorf, C. Parallel Simulation of Stochastic and Colored Petri Nets. Submitted for Publication.

6. Burdorf, C. Per-Trans: A Persistent Stochastic Petri Net Representation Language. In *Proceedings of the 22nd Annual Pittsburgh Conference on Modeling and Simulation* (1991).

7. Burdorf, C. Compiling Connectionist Simulations for SIMD and MIMD Architectures. In *Proceedings of the 1992 European Simulation Multiconference*, Society for Computer Simulation (1992).

8. Burdorf, C. POCONS: A Persistent Object-based Connectionist Simulator. In *Proceedings of the 1992 SCS Western Multiconference: Object-Oriented Simulation*, Society for Computer Simulation (1992).

9. D'Autrechy, C., Reggia, J. A., Sutton, G. G., and Goodall, S.M. A General-Purpose Simulation Environment for Developing Connectionist Models. *Simulation* (1988).

10. Feldbrugge, F. Petri Net Tool Overview 1989. In *Lecture Notes in Computer Science: Advances in Petri Nets 1989* (1989).

11. Feldman, J. A., Fanty, M. A., and Goddard, N. H. Computing with Structured Neural Networks. *IEEE Computer* (1988).

12. Floreen, P., Myllymaki, P., Orponen, P., and Tirri, H. Compiling Object Declarations into Connectionist Networks. *AICOM* (1990).

13. Geist, G. and Sunderam, V. *Network Based Concurrent Computing on the PVM System.* Oak Ridge National Laboratory (1991).

14. Goldberg, A. and Robson, D. *Smalltalk-80: The Language and its implementation*. Addison-Wesley, Reading, Massachusetts (1983).

15. Halstead, Robert H. Multilisp: A Language for Concurrent Symbolic Computation. *ACM Transactions on Programming Languages and Systems*, 7, 4 (October 1985).

16. Hayes, B. Finalization in the collector interface. In *International Workshop on Memory Management 92*, Springer-Verlag (September 1992) 277–298.

17. Kiczales, G., des Rivieres, J., and Bobrow, D. *The Art of the Metaobject Protocol*. MIT Press, Cambridge, Massachusetts (1991).

18. Kind, A. and Freidrich, H. A practical approach to type inference for eulisp. Published in this Journal.

19. Marsan, M. Stochastic Petri Nets: An Elementary Introduction. In *Lecture Notes in Computer Science: Advances in Petri Nets 1989* (1989).

20. Morrison, R. and Atkinson, M. P. Persistent Languages and Architectures. In *International Workshop on Computer Architectures to Support Security and Persistence of Information*, Springer-Verlag (1990).

21. Padget, J.A. and Nuyens, G. (Eds.). The EuLisp Definition. (1993). in preparation.

22. Peterson, J. L. *Petri Net Theory and the Modeling of Systems*. Prentice-Hall, Englewood Cliffs, N.J. (1981).

23. Playford, K. J. and Broadbery, P. A. *Feel: An Implementation of EuLisp*. Concurrent Processing Research Group, School of Mathematical Sciences, University of Bath (June 1991). May be obtained by anonymous ftp from ftp.bath.ac.uk.

24. Sabot, G. W. *The Paralation Model: Architecture Independent SIMD Programming*. MIT Press, Cambridge, MA (1988).

25. Stroustrup, Bjarne. *The C++ Programming Language*. Addison Wesley, second edition (1990).

26. Wang, D. and Hsu, C. SLONN: A Simulation Language for modeling of Neural Networks. *Simulation* (1990).

27. Weitzenfeld, A. *Neural Simulation Language Version 2.1*. Technical Report 91-05, Center for Neural Engineering, University of Southern California (August 1991).