

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, 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.

```

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 1: slot access protocol

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. In the following sections we describe some applications that use this technique.

3. 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 [17]. 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 by the program due to buffering by either the operating system or the application program. The problem is that 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 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

```

(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)))

```

Figure 2: Finalization of a file handle

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 object may be garbage collected). These two extensions¹, 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 finalization somewhere safe, so that when the space occupied by the object is recovered by garbage collection, the values that were stored in the finalization 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

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

```

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

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

;; instance allocation
(defmethod allocate ((cl <finalizable-class>) lst)
  (let ((handle (make-vector (finalizable-slot-count cl)))
        (obj (call-next-method)))
    ((setter primitive-slot-ref) cl
     (finalizable-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)))
      (do (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 3: Fragment of finalization code

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 proviso that this particular instance cannot be finalized. From the code in Figure 3, you will see that a slot is added to the instance to be finalized when it is allocated. This slot is initialized with a vector in which the slot

values needed for finalization are stored. The vector 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 (`make-proxy-object`) with the slot values from the vector 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 3.

This technique is adequate in most circumstances, 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. Virtual Shared Memory

Virtual Shared Memory (VSM) is a software technique for simulating the shared memory of many parallel architectures on a network of processors with local memory only — giving objects a form of spatial persistency. It can also 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 be 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, such as pattern matching. This would provide 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.

4.1. 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. 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 (abstract) 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 if new page and address classes are defined, it must be observed. One change from the defined semantics of EULISP is that objects may not appear to be `eq` to themselves because of caching arrangements (a page may leave the local processor and return). However, it is arguable whether `eq` should be well-defined (or even used!) in these circumstances. The function `eq1`, 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 [16], although a paration [28] implementation has also been developed [2]. The underlying interprocess communications mechanism is PVM [14], 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. The integration of VSM with persistence is being investigated so that one can employ both temporal and spatial persistency in a system.

5. Persistence

Persistence has 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] aim to 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 [23] include:

- reduced complexity;
- reduced code size and time to execute;
- 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 some of the problems encountered in porting it from its original language, Allegro Common Lisp, to EULISP.

5.1. Persistence in PSE

PSE's persistent object system supports sharing, maintaining, and inspecting of objects. Sharing of persistent objects has not been pursued because it involves issues of transaction management and is not one of the primary goals of this work. Other persistent object languages, for example, GEMSTONE and Picasso, are researching this topic and their results will contribute to the success of persistent object systems.

In general, an object which is declared to be persistent is retained in secondary storage after program execution terminates. In PSE, once a class has been declared to be persistent, instances of that class will automatically be made persistent. However from the programmer's perspective, persistent objects in PSE are referenced identically to non-persistent simulation objects. Furthermore, fetching and instantiating of a persistent object from secondary storage is performed transparently by the underlying PSE kernel. The kernel implementation of PSE is based on Rowe's [27] SOH (shared object hierarchy) methodology.

PSE is composed of the following components pictured in Figure 4: 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 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 (i) a pointer to the object's memory location, which is "nil" if the object is not present in the object space (ii) the object's location in the object file (iii) whether or not the object has been modified (iv) 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.

The database consists of several files. One file stores the objects, and there are separate files for the caches and the classes. Each object is stored as a fixed-sized record. If an object is modified to increase its size such that it exceeds the fixed size allocated, then the object is moved to the end of the

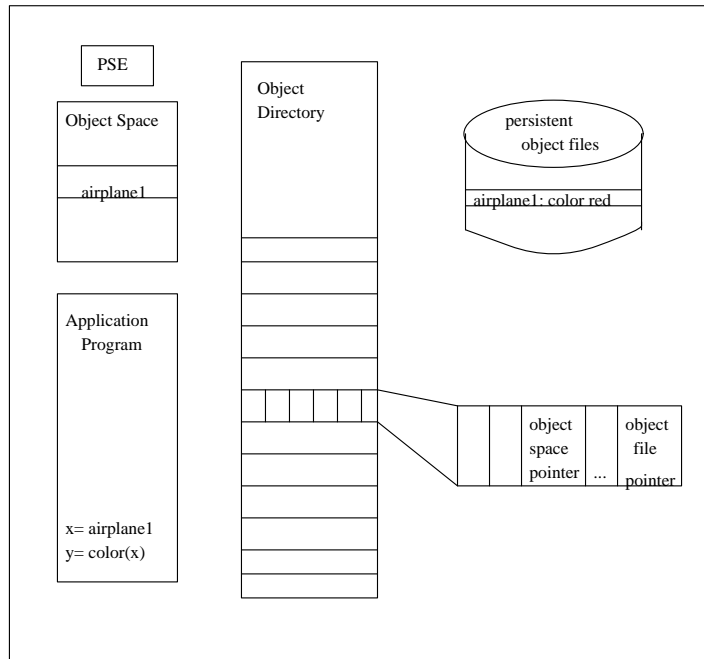


Figure 4: Components of PSE

file. Its size allocation is then increased to meet its new specification. The system also contains a routine that will garbage collect the unreferenced objects in the database.

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 mentioned earlier, the determination of the object's location, fetching, and instantiation are handled by the persistent object system and is transparent to the programmer.

The system provides a set of parameters that can be used to tune performance. For example, the total size of object space, and the size of the object directory. It is also possible to extract information on the dynamic sizes of these objects, so that a program may adjust the upper limits as necessary. The combination of these system parameters with the three choices of update modes, provides users with facilities for comparing performance under different PSE system constraints.

5.2. Porting PSE to EULISP

Concurrency provides the possibility of reducing the real execution time of a program through simultaneous execution of different code segments. PSE was ported to EULISP because it provides a thread facility on the Star-dent Titan multiprocessor which allows the implementation of concurrent programs. The components ported to EULISP include the persistent object system (described above) and the discrete-event and process-based simulation utilities. Those components had previously been implemented in Common Lisp at RAND Corporation.

The port of PSE to EULISP required a significant amount of effort, because the entire interface with the object system had to be rewritten, and EULISP modules provide a more austere environment than Common Lisp packages — the EULISP language prefers to gain run-time efficiency at the expense of development-time aids such as binding names at runtime, and the `eval` construct. In the Common Lisp version of PSE, the object system had been merged with the file system through low-level modifications. A new EULISP metaclass was created for persistent classes and objects and methods were added at the low-level to handle access and modification of these objects.

Implementing persistent classes as the hardest part. In Common Lisp, when a class was input from the database, PSE would construct a call to `defclass` and `eval` to instantiate the class. This technique was not possible in EULISP, because it does not have `eval`. Another problem is that when a class is created, the accessors for the slot values need to be defined. In Common Lisp, `defclass` defines the accessors. However, in EuLisp, the lack of `eval` makes it impossible to construct and execute a call to `defclass` as in Common Lisp. Calling `make` on the metaclass is one alternative, but `make` does not create the accessors, so that would have to be done manually. A problem further arises when creating the accessors in EuLisp at run time, because the accessor functions must be bound to the accessor names, but EULISP does not allow the creation of new bindings in modules at run time. This problem was only partially solved by having the accessor names defined at system load time. This restriction means the EULISP version of PSE does provide persistent classes seamlessly in the way of the Common Lisp version. Additionally, we note that it required more than twice as much code to implement persistent classes in EULISP than it did in Common Lisp.

Likewise, module restrictions made it difficult to have applications reside in separate modules from the PSE module. Since EULISP does not allow mutually referential modules, it was necessary to define all persistent classes in the PSE module, because local bindings can only be done in macro expansions and the functions to build the accessors are in the PSE

module. Therefore, since the accessors were defined in the PSE module, the application must be a part of the PSE module if it wants to use the slot accessors.

There are some compromises that can be made to alleviate this problem. The first compromise would be to store classes in their own module, then all of the slot accessors would have to be declared in the export list of the module. This solution is quite unsatisfactory though, because when one imports a module of classes, it will load all of the classes in that module instead of loading classes as demanded by the program. Also, it requires the application programmer to structure their code differently from how they would if the classes were non-persistent. This requirement violates the definition of persistent systems as having a *seamless* interface between the program and the database. Another solution, which was implemented, was to provide a construct called *persistent-classes*. This construct is used when loading classes from an already existing database. When persistent classes are defined, the system stores code in the database which when loaded into EuLisp and executed will create the accessors for a class and bind them to the slot-names listed in the call to *persistent-classes*. Thus, the use of the *persistent-classes* construct does create a seam of sorts between the program and database, but it is less of a seam than would be necessary if the classes were required to be in a different module.

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 caused all slot classes to be `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 work 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.

6. Applications of Persistence

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 processing, 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.

6.1. Concurrent Process-based Simulation

The motivation for concurrency is to improve the performance of simulations. However, to execute a persistent object-based simulation requires protocols to manage the concurrency to ensure that the simulation semantics are not altered from its sequential version. The choice was made to implement concurrency at the event level rather than the database transaction level, because if dependent events are synchronized on each object, database transactions will be synchronized as well. Event-level synchronization will allow independent events (eg. *move car to station X, process first car on station Y*) to execute in parallel. Dependent events which act on the same object (eg. *move car A to station X, process car A at station X*) will be executed in lock-step. Since events are the parallelizable unit, if they are synchronized based on the write sets of objects (a *write-set* is a group of objects that are dependent on each other because they modify or access the same mutable attributes), so will the database transactions they generate. If dependent events are synchronized, then all database transactions will be serialized for each object, because events are the driving apparatus of the simulation and the only agent which produces database operations. Thus, the environment demands a protocol which will control concurrency for events and the transactions will follow suit.

As is described in [9], conservative protocols [22] were chosen. The main advantage being that conservative mechanisms require less primary memory than optimistic ones, because there is no need to save the state each time an event is processed and input queues contain no antimessages. Also, in the case of persistent systems, there is no need to save modifications to the database, because due to the lock step execution of events for each object, once modifications are made there is no need for the protocol to undo them which is necessary under optimistic mechanisms [18].

6.2. 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 [10, 13, 12, 30].

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 [13] other neural network tools do not support an object-oriented design methodology. Both Neula and NSL [31] have object-oriented constructs, but differ in their

```

(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 5: Fragment of POCONS code for a Middle Earth neural net

syntax and semantics which is unlike the widely-used object-oriented languages like Smalltalk [15], C++ [29], 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.

6.2.1. *An 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

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 5.

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 initialization 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-names* 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].

6.3. Petri nets

Petri nets are widely used in the simulation of concurrent systems [25]. As a result of the popularity of Petri nets, a variety of tools have been developed [11], including ones for the 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], which 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 [21]. 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.

6.3.1. *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:

- Persistence;
- Declarative;
- 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 of PSE. 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 [26].

Several Petri net models have been implemented using Per-trans and it has been extended to produce parallel simulations [5]. The Per-trans language is explained in detail with examples in [6].

7. Conclusions and Further Work

We have discussed the use of the Telos object system in 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 Protocol. 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. Burdorf, C. and Fitch, J. Cloning Persistent Objects Under a Conservative Mechanism for Concurrency Control. In *Proceedings of the 1993 European Simulation Multiconference*, Society for Computer Simulation (1993).
10. D'Autrechy, C., Reggia, J. A., Sutton, G. G., and Goodall, S.M. A General-Purpose Simulation Environment for Developing Connectionist Models. *Simulation* (1988).
11. Feldbrugge, F. Petri Net Tool Overview 1989. In *Lecture Notes in Computer Science: Advances in Petri Nets 1989* (1989).
12. Feldman, J. A., Fanty, M. A., and Goddard, N. H. Computing with Structured Neural Networks. *IEEE Computer* (1988).
13. Floreen, P., Myllymaki, P., Orponen, P., and Tirri, H. Compiling Object Declarations into Connectionist Networks. *AICOM* (1990).
14. Geist, G. and Sunderam, V. *Network Based Concurrent Computing on the PVM System*. Oak Ridge National Laboratory (1991).
15. Goldberg, A. and Robson, D. *Smalltalk-80: The Language and its implementation*. Addison-Wesley, Reading, Massachusetts (1983).
16. Halstead, Robert H. Multilisp: A Language for Concurrent Symbolic Computation. *ACM Transactions on Programming Languages and Systems*, 7, 4 (October 1985).
17. Hayes, B. Finalization in the collector interface. In *International Workshop on Memory Management 92*, Springer-Verlag (September 1992) 277-298.
18. Jefferson, D. Virtual Time. *ACM Transactions on Programming Languages and Systems*, 7, 3 (July 1985).
19. Kiczales, G., des Rivieres, J., and Bobrow, D. *The Art of the Metaobject Protocol*. MIT Press, Cambridge, Massachusetts (1991).

20. Kind, A. and Freidrich, H. A practical approach to type inference for eulisp. Published in this Journal.
21. Marsan, M. Stochastic Petri Nets: An Elementary Introduction. In *Lecture Notes in Computer Science: Advances in Petri Nets 1989* (1989).
22. Misra, J. Distributed Discrete-Event Simulation. *Computing Surveys* (March 1986).
23. 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).
24. Padget, J.A. and Nuyens, G. (Eds.). The EULISP Definition. (1993). in preparation.
25. Peterson, J. L. *Petri Net Theory and the Modeling of Systems*. Prentice-Hall, Englewood Cliffs, N.J. (1981).
26. 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.
27. Rowe, L. A. A Shared Object Hierarchy. In *International Workshop on Object-Oriented Database Systems*, IEEE (1986).
28. Sabot, G. W. *The Paralation Model: Architecture Independent SIMD Programming*. MIT Press, Cambridge, MA (1988).
29. Stroustrup, Bjarne. *The C++ Programming Language*. Addison Wesley, second edition (1990).
30. Wang, D. and Hsu, C. SLONN: A Simulation Language for modeling of Neural Networks. *Simulation* (1990).
31. Weitzenfeld, A. *Neural Simulation Language Version 2.1*. Technical Report 91-05, Center for Neural Engineering, University of Southern California (August 1991).