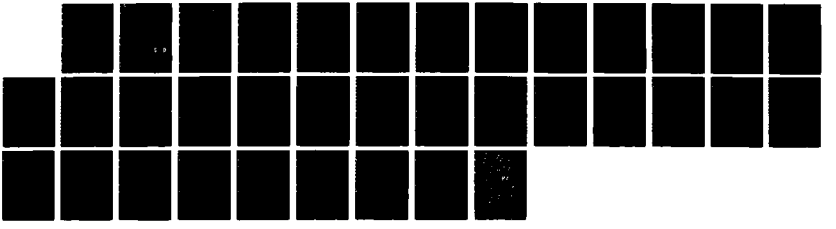
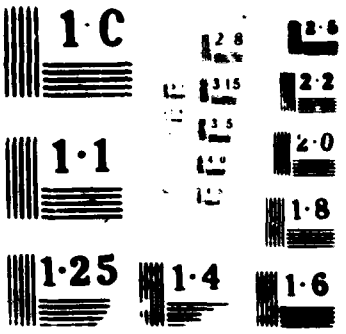
AD-A187 719 BUTTERFLY (TRADENAME) EXPERT SYSTEMS EXECUTION ENVIRONMENT: A FUNCTIONAL SPECIFICATION(U) BBN LABS INC CAMBRIDGE MA C QUAYLE APR 86 BBN-6225 MDA903-84-C-0033 1/1

UNCLASSIFIED F/G 12/5 NL




1·C
1·1
1·25
1·4
1·6
1·8
2·0
2·2

BBN Laboratories Incorporated

A Subsidiary of Bolt Beranek and Newman Inc.

Report No. 62[illegible]

Butterfly™ Expert Systems Execution Environment: A Functional Specification

Casey Quayle

April 1986

Prepared for:
Defense Advanced Research Projects Agency

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

REPORT DOCUMENTATION PAGE		READ INSTRUCTIONS BEFORE COMPLETING FORM
1. REPORT NUMBER Report No. 6225	2. GOVT ACCESSION NO. A187 712	3. RECIPIENT'S CATALOG NUMBER
4. TITLE (and Subtitle) Butterfly Expert Systems Execution Environment: A Functional Specification		5. TYPE OF REPORT & PERIOD COVERED Technical Report April 1986
		6. PERFORMING ORG. REPORT NUMBER 6225
7. AUTHOR(s) Casey Quayle		8. CONTRACT OR GRANT NUMBER(s) MDA903-84-C-0033
9. PERFORMING ORGANIZATION NAME AND ADDRESS BBN Laboratories Inc. 10 Moulton Street Cambridge, MA 02238		10. PROGRAM ELEMENT, PROJECT, TASK AREA & WORK UNIT NUMBERS
11. CONTROLLING OFFICE NAME AND ADDRESS Defense Advanced Research Projects Agency 1400 Wilson Boulevard Arlington, VA 22209		12. REPORT DATE April 1986
		13. NUMBER OF PAGES 28
14. MONITORING AGENCY NAME & ADDRESS(if different from Controlling Office)		15. SECURITY CLASS. (of this report) Unclassified
		15a. DECLASSIFICATION/DOWNGRADING SCHEDULE

16. DISTRIBUTION STATEMENT (of this Report)

17. DISTRIBUTION STATEMENT (of the abstract entered in Block 20, if different from Report)

18. SUPPLEMENTARY NOTES

19. KEY WORDS (Continue on reverse side if necessary and identify by block number)
Expert Systems, Object Oriented Programming Support, Rule System Programming

20. ABSTRACT (Continue on reverse side if necessary and identify by block number)

This document is the initial functional specification of the Butterfly(TM) Expert Systems Tool Kit currently under development at BBN. It is an evolving document which will be updated at least four times a year.

The Buuterfly Expert Systems Tool Kit will support the development of expert systems on the Butterfly multiprocessor. It is implemented entirely in Common Lisp. The tool kit will support object oriented programming, annotated slots(or "Active Values")& rule system programming.

DD FORM 1 JAN 73 1473 EDITION OF 1 NOV 65 IS OBSOLETE

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

The tool kit also provides some basic utilities for a prorammer interface, debugging, and metering features for measuring concurrency.

The Butterfly Expert System Tool Kit is distributed between a Butterfly multiprocessor and a front end lisp machine. Theeparts running on the Butterfly include run time support for the tool kit and the expert system it supports. The back end Butterfly componant of the tool kit communicates performance and debugging information to the front end for processing and display. The back end receives programs and commands from the front end.

The front end componant provides programmer and end user interfaces for the expert system. The front end lisp machine will support the expert system developer during incremental editing of programs and knowledge, debugging sessions, and program tuning. It will also handle source file control and will down load programs and knowledge to the Butterfly during a session. During performance session, it will interact with the expert system user.

The lisp machine is capable of running in stand alone mode when a Butterfly is unavailable. All of the programs written for the tool kit will execute serially in the Common Lisp environment of the front end lisp machine.

Report No. 6225

Butterfly™ Expert Systems Execution Environment: A Functional Specification

Casey Quayle

April 1986

Prepared by:

Bolt Beranek and Newman Inc.
10 Moulton Street
Cambridge, Massachusetts 02238

Prepared for:

Defense Advanced Research Projects Agency
1400 Wilson Boulevard
Arlington, Virginia 22209

Butterfly is a trademark of BBN Laboratories Inc.

TABLE OF CONTENTS

	Page
1. OVERVIEW	1
2. OBJECT ORIENTED PROGRAMMING	3
2.1 CommonLoops	3
2.2 Defining Classes	3
2.2.1 Automatically Built Messages	4
2.2.2 Accessors	4
2.2.3 Class Precedence	5
2.2.4 Constructors	6
2.3 Defining messages	6
2.3.1 Using Defmeth	6
2.3.2 Type Specifiers	7
2.3.3 Options for defmeth	8
2.3.4 Method Combination	8
2.4 Parallelism in CommonLoops	8
2.4.1 Explicit Parallelism	8
2.4.2 Implicit Parallelism	10
2.4.3 Synchronization and Locking	10
2.4.4 Caveats on Discriminating Functions	12
2.5 Summary of Parallel extensions	12
2.6 Programming Environment	13
2.6.1 Editing	13
2.6.2 Tracing and Breaking	14
2.6.3 Using the Debugger	14
2.7 Metering	14
2.7.1 Measuring Serial Bottlenecks	14
2.7.2 Measuring Concurrency	15
3. ANNOTATED SLOTS	17
3.1 Implementation of Annotated Slots	17
3.1.1 Fetch Function	18
3.1.2 Replace Function	18
3.1.3 Creating Annotated Slots	19
3.1.4 Removing Annotated Slots	19
3.2 Parallelism Using Annotated Values	19
3.3 Programming Interface	20

3.4 Metering 20

4. RULE SYSTEM ENVIRONMENT 21

4.1 Rule System Architecture 21
4.1.1 Rule System Processor 22
4.1.2 Rule System Context 22
4.1.3 Rule System Access Interface 22
4.1.4 Rule System Development Interface 23
4.2 Conventions 23
4.3 Example Rule Systems 25
4.3.1 Access Environment Rule System 25
4.3.2 Propositional Based Rule System 27
4.4 Rule system metering 28

1. OVERVIEW

This document is the initial functional specification of the Butterfly™ Expert Systems Tool Kit currently under development at BBN. It is an evolving document that will be updated at least four times a year.

The Butterfly Expert Systems Tool Kit will support the development of expert systems on the Butterfly multiprocessor. It is implemented entirely in Common Lisp. The tool kit will support object oriented programming, annotated slots[1], rule system programming. The tool kit also provides some basic utilities for a programmer interface, debugging, and metering features for measuring concurrency.

The Butterfly Expert System Tool Kit is distributed between a Butterfly multiprocessor and a front end lisp machine. The parts running on the Butterfly include run time support for the tool kit and the expert system it supports. The backend Butterfly component of the tool kit communicates performance and debugging information to the front end for processing and display. The backend receives programs and commands from the front end.

The front end component of the tool kit provides programmer and end user interfaces for the expert system. The front end lisp machine will support the expert system developer during incremental editing of programs and knowledge, debugging sessions, and program tuning. It will also handle source file control and will down load programs and knowledge to the Butterfly during a session. During performance session, it will interact with the expert system user.

The lisp machine is capable of running in stand alone mode when a Butterfly is unavailable. All of the programs written for the expert system tool kit will execute serially in the Common Lisp environment of the front end lisp machine.

The following sections describe each of the major programming methodologies that is supported. The programming environment utilities that relate to each of style are described in each of these relevant section.

---

[1] Also known as *Active Values*.

Section 2 describes the object oriented programming support. A parallel version of the CommonLoops object oriented programming system is to be provided on the butterfly. CommonLoops also provides an implementation basis, along with Common Lisp, for the other components of the tool kit.

Section 3 describes annotated slots, the Butterfly Expert System Tool Kit version of active values. Any slot on a CommonLoops object instance may be annotated with a demon function that will run whenever data is read from or placed in the slot.

Section 4 describes the support for rule system programming. Described here is the support for the development of a rule system. Two rule systems under development in the rule system environment are also described.

2. OBJECT ORIENTED PROGRAMMING

The form of Object Oriented Programming supported by in the Butterfly Expert Systems Tool Kit is provided by implementing a version of *CommonLoops*[2] . In this section we will describe the basics of Commonloops and then describe how it will be supported on the Butterfly.

2.1 CommonLoops

In CommonLoops objects are instances of *classes*. Each class is a member of a class heterarchy. New classes are defined by an extended version of the Common Lisp **defstruct** form. Classes are represented as objects, they are instances of *metaclasses*. A metaclass is also a class.

Objects, or groups of objects, respond to *messages* in a class specific manner. The behavior of a message over some class or classes is procedurally described as a method. Methods are defined using an extended version of the Common Lisp **defun** form.

2.2 Defining Classes

Classes are defined using an extension of the Common Lisp defstruct form. The name of the extended form is *ndefstruct*. The extension includes the introduction of a new defstruct option, *:class*, and the introduction of a new slot option, *allocation*. Other than these extensions a ndefstruct form appears identical to a defstruct form.

Prototype ndefstruct forms are:

```
(ndefstruct (3-d-position (:class class))
     (x-pos 0)
     y-pos
     z-pos)

(ndefstruct (celestial-body (:class class)(:include 3-d-position))
    (weight 0 :allocation :instance)
    (all-known nil :allocation :class)
    (name nil :allocation :dynamic))
```

---

[2]CommonLoops was developed by XEROX Corporation at the Xerox Palo Alto Research Center

In this example the class *3-d-position* has **class** as its metaclass. This is the usual choice for the metaclass designation. 3-d-position has three variables the variable x-pos is initialized, by default, to 0. The other two variables, y-pos and z-pos receive no initial default value.

The second example *celestial-body* has three variables, their designation illustrates the use of the **:allocation** slot option. The most common allocation is **:instance** as used in the description of the weight variable. An allocation type of :instance means that a slot for the variable is allocated for a local value for the variable in each instance, this is the same as in Common Lisp's defstruct. Since the :instance allocation type is so common it is the default, as illustrated in the 3-d-position example. These are called *instance variables*.

An :allocation type of :class means that a single slot for storage is allocated in the instance of the class object. This location is shared by all instances of the class. A :class variable may be thought of as a global variable common to all instances. These are called *class variables*.

An :allocation type of :dynamic means that no storage for the slot is preallocated. A storage location is allocated on an instance by instance basis when the first attempt is made to access the variable. These are called *dynamic variables*.

The class *celestial-body* illustrates the use of the :include defstruct option. This enables celestial-body to *inherit* slots from 3-d-object. The :include option may take a list argument if it is desirable for a class to inherit from a multitude of classes.

2.2.1 Automatically Built Messages

Like Common Lisps defstruct, ndefstruct will automatically build functions that serve as a constructor, copier, predicate, and accessors to slots. These functions are implemented by ndefstruct as messages.

2.2.2 Accessors

In the celestial-body case the constructor would be ***make-celestial-body***, the predicate would be *celestial-body-p*, and the copier would be ***copy-celestial-body***. The accessors would be *celestial-body-weight*, *celestial-body-all-known*, celestial-

body-name, *3-d-position-x-pos*, *3-d-position-y-pos*, and *3-d-position-z-pos*[3] The accessors will return the values of the slots, slots may be modified using the setf form on the accessor (e.g. **(setf (celestial-body-weight asteroid) 2000)**).

The automatically constructed accessors defined by ndefstruct are useful when it is known at definition time what slots should be accessed. Sometimes it is useful to determine the slot to access at run time.

To allow run time calculation of the slot to reference the primitive **get-slot** is provided. Given the form (get-slot Object Slot), Object should evaluate to an instance reference and Slot should evaluate to the name of a slot belonging to the object. The value stored in the slot of the object is returned. The form (setf (get-slot Object Slot) value) will modify the value stored in slot of the object[4].

For example, the form (3-d-position-y-pos point) is functionally equivalent (though more efficient) to (get-slot point 'y-pos).

2.2.3 Class Precedence

A class will inherit from all the classes on its :include list and their transitive closure. The group of classes a class would inherit from form a lattice. This lattice is ordered into a totally ordered *class precedence list* by a method associated with the meta class of the class. The class precedence list for classes whose meta class is *class* is computed by first doing a depth first tree walk of the lattice. Then duplicate entries are removed bye keeping only the last occurrence of each.

A class inherits only one slot of a given name. The slot description comes from the description on the earliest class on the class precedence list.

Since the computation of the class precedence is a method defined on the meta class, one can define different meta classes that have a distinct method for computing the class precedence.

---

[3]Note that the inherited slot accessors retain the "conc name" defined in the form that named them.

[4]A form **put-slot** is also defined, but following Common Lisp conventions using the setf alternative is preferred usage.

2.2.4 Constructors

A constructor is automatically built for class defined by ndefstruct. The constructor for 3-d-position is make-3-d-position. This constructor follows the convention of the Common Lisp constructors for determining the initial values of slots.

It is sometimes convenient to decide the name of a class to instantiate at runtime To do this the primitive **make** is defined. The form

```
(make-3-d-position :x-pos 10)
```

is equivalent to

```
(make '3-d-position :x-pos 10)
```

After building a virgin skeleton of the given class it is sent an initialize message to fill in the initial values of the slots. A specialization for this message on the class can be written, see section 2.3.4 for more details.

2.3 Defining messages

Objects, or groups of objects, respond to *messages* in a class specific manner. The behavior of a message over some class or classes is procedurally described as a method. Methods are defined using an extended version of the Common Lisp **defun** form The extended defun is called *defmeth*.

The accessors described in section 2.2.2 are in fact messages that the objects respond to For example the message 3-d-position-y-pos is a message to an instance of 3-d-position asking for the value of the y-pos slot.

2.3.1 Using Defmeth

User defined messages are defined as a method statement for the class. Method statements are defined using *defmeth*, an extended version of defun. An example of a defmeth is

```
(defmeth new-position ((point 3-d-position) new-x new-y new-z)
     (setf (3-d-position-x-pos point) new-x)
     (setf (3-d-position-y-pos point) new-y)
     (setf (3-d-position-z-pos point) new-z))
```

This method defines, procedurally, how instances of 3-d-position would respond to the new-position message. The message would be sent in the following way:

```
(new-position my-point (3-d-position-x-pos my-bed)
                       (3-d-position-y-pos my-bed)
                       (3-d-position-z-pos my-bed))
```

The general form of defmeth is.

```
(defmeth <name&options> <lambda-list>
    . <body>)
```

The **<name&options>** is a symbol or a list consisting of a symbol followed options following the same syntax as defstruct options. The name is called the *selector* of the method, it is the name of the message that the method is defining.

2.3.2 Type Specifiers

The **<lambda-list>** is an extend form of the lambda list that would be given to defun where any required argument may be for the form *(var type-specifier)*. Each type-specifier names a class.

A discriminating function is placed on the function cell of the selector. The discriminating function is entered when the message is sent, it is responsible for running the method whose type specifiers match the all supplied type-specifiers[5]. A method with no type specifiers is the default method that is run when no other is applicable. Since the discriminator checks all arguments a type specifier may appear in any required argument position, and several type specifiers may be given.

For example the new-position message can be defined to move a celestial-body to some 3-d-position in the following manner.

```
(defmeth new-position ((to-point 3-d-position) (mover celestial-body))
      (setf (3-d-position-x-pos mover) (3-d-position-x-pos to-point))
      (setf (3-d-position-y-pos mover) (3-d-position-y-pos to-point))
      (setf (3-d-position-z-pos mover) (3-d-position-z-pos to-point)))
```

This message could then be sent as:

```
(new-position my-bed my-point)
```

There may be several methods having type specification that match a set of arguments to a message. A method precedence list is defined to order methods according to their specificity. The class precedence list of each type specifier is used

---

[5]This has some implications on how parallelism is realized, see section 2.4.4 for details.

in a left to right manner to compute the method precedence list

2.3.3 Options for defmeth

The discriminator and method defined by defmeth are themselves instances of the classes "discriminator" and "method" respectively The options list given in the <name&options> portion of a defmeth form may be used to designate alternative classes for either (using discriminator or method option)

2.3.4 Method Combination

Often it desirable to extend slightly the behavior of a class of object over the behavior of its supper-class This is called *method combination*, the way to invoke method combination is with the primitive **run-super** It is defined to run the most specific method matching the arguments of the current method that is more general than the method in which the run-super occurs

For example

```
(defmeth new-pos ((item celestial-body) x y z)
    (run-super)
    (update-display item))
```

The proper way to write a specialization for the initialize method of a class (see section 2.2.4) is

```
(defmeth initialize ((object <class-of-object>) init-plist)
    (run-super)
    . <additional initialization code>)
```

2.4 Parallelism in CommonLoops

There are three areas that the parallelism of Butterfly Common Lisp are relevant to object oriented programming in CommonLoops explicit parallelism, implicit parallelism, synchronization and locking There is also a caveat related to discriminating functions

2.4.1 Explicit Parallelism

The body of a method statement is a procedural description of how an object or group of objects respond to a message, it is Common Lisp code The future primitive

may be used anywhere inside a method to introduce explicit parallelism. For example

```
(defmeth new-position ((to-point 3-d-position) (mover celestial-body))
      (future (setf (3-d-position-x-pos mover) (3-d-position-x-pos to-point)))
      (future (setf (3-d-position-y-pos mover) (3-d-position-y-pos to-point)))
      (future (setf (3-d-position-z-pos mover) (3-d-position-z-pos to-point))
    ))
```

Another example would be the specialization of the initialize method described in section 2.3.4 might be parallelized by

```
(defmeth  initialize ((object <class-of-object>) init-plist)
      (prog1 (run-super)
             (future (progn . <additional initialization code>)))))
```

This may be the preferred way to do it, see section 2.4.4.

It is anticipated a common style of explicit parallelism will be defined using a form equivalent to:

```
(defmeth <name> <lambda-list>
       (future (progn . <body>)))
```

A new specialization of the class **method**, called **parallel-method** will be defined in Butterfly Common Loops that will automatically support this style of parallelism. Using this **parallel-method** the method statement would be written as:

```
(defmeth (<name> (:method parallel-method)) <lambda-list>
       . <body>)
```

Some methods may be true functions in the formal sense, that is no matter how often they are called with the same arguments they return the same value. This kinds of lisp forms are often called *memoizable*. Another specialization of class **method** will be *memoization*, it will use a hash table to cash the answer and return a future the first time the method is called on any set of arguments and return the contents of the hash table on subsequent calls. Memoizable methods would be equivalent to the following form.

```
(defmeth <name> <lambda-list>
   (let ((result (gethash <arguments> <name-hash-table>))
      (if (null result)
          (setf (gethash <arguments> <name-hash-table>)
                (future (progn . <body>)))
          result)))
```

A **memoizable method** would be expressed as:

```
(defmeth (<name> (:method memoizable)) <lambda-list>
       . <body>)
```

2.4.2 Implicit Parallelism

The run time support for Common Loops is amazingly fast, therefore it currently appears that there is little need to introduce implicit parallelism during run time.

During definition time, i.e. while processing ndefstruct or defmeth forms, there are some delays - mostly associated with class precedence computation that may benefit from the introduction of parallelism.

2.4.3 Synchronization and Locking

When programs are written in an asynchronous environment such as a multiprocessor there is often the need to describe a critical section of code. A way to guard a critical section is to lock a lock on entry and unlock it on exit using a binary semaphore. The class **binary-semaphore** will be defined as part of the Butterfly Expert System Tool Kit. Methods will be defined to **make** a semaphore, **lock** a semaphore, and **unlock** a semaphore.

Often it will be important to allow a single writer or reader to access a slot. To support such care, two extensions of the meta class **class**, called **protectable-class** and **protected-class**, will be defined. The slot options of **ndefstruct** will be defined to include the option :**lockable** when semaphore protection is desired for a slot. The :**lockable** slot option is only allowed when the meta class, the argument of the :class defstruct option, is **protected-class**, **protectable-class**, or is a specialization of one of these.

The difference between **protectable-class** and **protected-class** is the default value for the :**lockable** option. For the meta class **protectable-class** it is **nil** and for the meta class **protected-class** it is T.

When the :**lockable** slot option is used and is T a locking protocol will be established for the slot unsing a **binary-semaphore**. When the :**lockable** option is not T no protocol will be established. When a slot has a T :lockable option a method, called the **guardian method** will be defined which will retrieve the semaphore associated with the slot. The name of the guardian method will be the string *"GUARDIAN-"* prepended to the name of the accessor method for the slot. That is, if the slot accessor is 3-D-X-POS, then the guardian method would be GUARDIAN-3-D-X-POS.

The accessor methods for :lockable slots will first lock the slot's semaphore, retrieve or set the slot value, and finally unlock the semaphore. The accessor methods of a :lockable slot will take an optional second argument, a semaphore. If the semaphore is the slot's semaphore and is locked the accessor will assume the process providing the semaphore is the process that locked it and will access the slot, the semaphore will not be unlocked on exit. It will be the responsibility of the calling process to unlock it. This capability is provided for situations where the critical section of code is larger than a simple, atomic, slot access (as in first reading the slot and then writing an altered value based on the result).

Get-slot *of* a :lockable slot will obey the same protocol as the accessor of the slot.

A new special form, **with-locked-accessor** will be defined to make using :locked slots easier. The general form of with-locked-accessor is:

```
(with-locked-accessor (<lock-forms>)
    . <body>)
```

where <lock-forms> is one or more instances of the sequence (var object slot) var is bound to the locked semaphore associated with the slot **slot** of object **object**. Body is executed, under unwind-protection, within the lexical scope of each of the bound lock vars. On exit, the locks are unlocked. The value returned by <body> is the value returned by with-locked-accessor.

For example[6]:

```
(ndefstruct (3-d-position
             (:class protected-class))  ; all slots are locked
    x-pos
    y-pos
    z-pos)

(defmeth new-position ((point 3-d-position) new-x new-y new-z)
   (with-locked-accessor  ((x-lock point 'new-x)
                           (y-lock point 'new-y)
                           (z-lock point 'new-z))
      (setf (3-d-position-x-pos point x-lock) new-x)
      (setf (3-d-position-y-pos point y-lock) new-y)
      (setf (3-d-position-z-pos point z-lock) new-z)))
```

---

[6]Note however that if in the same context of this example a second method that attempts to lock the same set of slots and both methods are concurrently invoked on the same instance then deadlock is possible.

2.4.4 Caveats on Discriminating Functions

Section 2.3.2 describes the discriminator function that is defined for each message selector. The discriminator must determine the type specifiers of the arguments to the message. To determine the type of an argument, the discriminator must touch that argument. In short, there is an implicit join on all concurrent computation that produce the arguments of the message.

The implication of touching a datum is if the datum is a future that has yet to return, then the toucher is suspended until the future value is established. This means that sending a message to a future will not induce parallel computation, e.g. the form (<message> (future (<make-object))) is wasted overhead in setting up a future since the selector for <message> must touch the future to establish the type of the resulting object.

To avoid unnecessary touching of data, the discriminator functions defined by the Butterfly Common Loops run time system will be careful to touch a datum only when it must, i.e. when it is really necessary to establish the class of a datum.

2.5 Summary of Parallel extensions

- **Explicit Parallelism**. The use of the *future* construct is allowed in the body of method code. See section 2.4.1
- **Parallel-method method class**. Use of Parallel-method as the **:method** option in defmeth will cause the method to immediately return a future when it is called. See section 2.4.1
- **Memoizable method class**. Use of Memoizable as the **:method** option in defmeth will cause the method to check a hash-table for a result. If a result is found it is returned. If a result is not found then the method body is run a-la parallel-method and the result is placed in the hash-table. See section 2.4.1.
- **Binary-semaphore class**. A class that instantiates semaphores. This class receives the **lock** and **unlock** messages. A process sending a **lock** message to a locked semaphore will suspend until the lock is unlocked. See section 2.4.3.
- **The :lockable slot option**. Available for classes whose meta class is either **protectable-class**, **protected-class**, or one of their specializations. When the :lockable option is T accessors to the slot behave as if they are surrounded by implicit **lock** and **unlock** messages to a **binary-semaphore** associated with the slot. The semaphore may be obtained via the slot guardian function built for the slot, separately locked and used as an

optional argument to the accessor. See section 2.4.3.

- **Protectable-class meta class**. This meta class or one of its specializations allows the :lockable slot option to be used in class definitions. The default value for the :lockable option is NIL. See section 2.4.3.

- **Protected-class meta class**. This meta class or one of its specializations allows the :lockable slot option to be used in class definitions. The default value for the :lockable option is T. See section 2.4.3.

- **Guardian method**. Method built to retrieve the binary semaphore associated with slots that have been defined using the :lockable option with a value T. The semaphore may be sent lock and unlock messages and may be provided as an optional argument to the slots accessor method. See section 2.4.3.

- **With-locked-accessor special form**. Special form to conveniently work with the semaphores associated with :lockable slots. With-locked-accessor will surround its body with code to lock and unlock the accessor as designated. Analogous to the Common Lisp special form *with-open-file*. See section 2.4.3.

2.6 Programming Environment

2.6.1 Editing

The front end machine will be responsible for providing editing services on Butterfly Common Loops forms. An internal data base will be maintained that indexes a *commonloops-reference* to the editable instance of the form. The ed function defined by **Common Lisp** will be extended to accept a *commonloops-reference* as an argument. The front end's resident editor will be invoked on the editable instance of the *commonloops-reference* definition.

A commonloops-reference is list of symbols. The first symbol is either the keyword :**struct** or the keyword :**meth**, these determine if the reference is to a **ndefstruct** definition or a **defmeth** definition. If the keyword is :struct then only one symbol should follow, the name of the class being edited. If the keyword is :meth then the second symbol should be the name of the method selector (the name of the message) followed by the type specifiers that denote the classes named in the lambda-list of the method definition. For arguments that are not typed in the lambda-list the symbol T should be used. Examples of commonloops-references are:

```
(:struct 3-d-position)  ;defined in section 2.2
(:meth new-position 3-d-position t t t) ;defined in section 2.3.1
(:meth new-position 3-d-position celestial-body)  ;defined in section 2.3.2
```

During an editing session the programmer may cause the new definition of a CommonLoops form to be asserted in a manner identical to the way it is done with Common Lisp functions.

2.6.2 Tracing and Breaking

Methods may be traced or broken. The tracing and breaking utilities of Butterfly Common Lisp will be extended to accept a commonloops-reference in addition to the names of functions.

When printing information about method entry and exit the debugging utilities will use commonloops-reference notation.

2.6.3 Using the Debugger

Sending a method entails two function calls. The first is the entry into the discriminator function. The discriminator function determines the proper method function and calls it. The debugger will only show frames for the method function while suppressing calls to the discriminator function when it is displaying a backtrace. If a programmer attempts to move the debuggers attention to a stack frame the debugger will not move to the discriminating function.

2.7 Metering

All function profiling utilities developed for Butterfly Common Lisp will be extended to profiling of methods.

2.7.1 Measuring Serial Bottlenecks

The implicit join described in section 2.4.4 represents a serial bottleneck. To help the programmer understand how much concurrency is lost by this bottle neck a specialization on class **discriminator**, called **metering-discriminator** will be developed that will keep track of the amount of time spent waiting for a future to return. This information will be kept on a per **selector** basis and reported as a histogram.

The function meter-selector will replace the discriminator on a selector name with a metering-discriminator and initialize the statistics it maintains. If the discriminator is already a metering-discriminator then meter-selector will simply

reinitialize the statistics.

The function report-meter-selector will print the histogram in a window on the front end machine.

2.7.2 Measuring Concurrency

We are planning to provide meters that describe various concurrency parameters. Ideas include:

- The distribution of number of concurrent messages an instance is concurrently processing.
- The distribution of concurrent messages a class of instances is concurrently processing.
- The distribution of the number of concurrent invocations of a particular message.

The complete specification of these meters is pending further experimentation with the Butterfly Common Loops implementation and will be described more completely in future versions of this specification.

3. ANNOTATED SLOTS

Annotated slots are used to invoke a function when ever the slot is accessed, i.e. either read or written. Annotated slots are called "Active Values" in some systems[7].

A slot annotation may be thought of as a demon on the slot. It causes a procedure to be run when ever the slot is accessed. The annotation may mask the operation if it so chooses, that is return a value that is not really in the slot or alter the value that is being placed on the slot.

Annotated slots may be placed on individual instances of a class, or may be placed on all instances of a class.

A slot may have several annotations at once. Here the annotations are said to be nested. Nested annotations work best when they are independent of one another.

Currently, any class whose meta class is **class** may have any slot annotated. If the overhead of this generality is too high future implementations will define a specialization of the meta class **class** that will allow active values.

3.1 Implementation of Annotated Slots

A slot annotation is implemented as an object, it is an instance of the class **slot-annotation**. The class definition of slot-annotation is effectively:

```
(ndefstruct (slot-annotation (:class class))
     name
     object
     local-state
     fetch-function
     replace-function)
```

Name is the name of the slot that has been annotated. *Object* is the instance whose slot name has been annotated. *Local-state* holds the data that would normally be in the slot. *Fetch-function* is the name of a function that gets called with standard arguments when an attempt is made to get the slot. If the fetch-function is

---

[7] e.g. Loops (XEROX) or KEE (Intellicorp). We are intensionally introducing the new terminology, it is a slot that is active - not a value, the activity stems from an annotation explicitly placed on the slot.

nil, then the value of local-state is returned. *Replace-function* is the name of a function that gets called with standard arguments when an attempt is made to modify the slot. If the replace-function is nil, then the value of the the local-state is modified with the new value.

The local-state may itself be an instance of slot-annotation, then annotated-slot processing recurs. The outer annotation will invoke the inner annotation when the outer annotation attempts to access the local-state. Thus the programmer has a wide range of control over the the effective invocation order of the demons[8].

3.1.1 Fetch Function

The *fetch-function* of an annotated-slot names a function of three arguments. *object*, *name*, and *annotation*.

- *object* is the instance that contains this annotation on one of its slots.
- *name* is the slot-name, from ndefstruct, of the slot that has been annotated.
- *annotation* is the instance of class **slot-annotation** that is being accessed, to get its local-state one sends it a **get-local-state** message (if the local state is itself an annotated slot then the process recurs).

The fetch-function should return the value it wants the caller of the slot-access to see. The message get-local-state is uses the accessor method, slot-annotation-local-state, to retrieve the local-state slot and recure if necessary.

3.1.2 Replace Function

The *replace-function* of an annotated-slot names a function of four arguments: *object*, *name*, *annotation*, and *newvalue*.

- *object* is the instance that contains this annotation on one of its slots.
- *name* is the slot-name, from ndefstruct, of the slot that has been annotated.
- *annotation* is the instance of class **slot-annotation** that is being accessed, to modify its local-state one sends it a **put-local-state** message (if the local state is itself an annotated slot then the process recurs).

---

[8] By deciding when in each demon to access, perhaps concurrently, the local-state, see section 3.2.

- *newvalue* is the new value to place in local state.

The replace-function should return the replaced value.

3.1.3 Creating Annotated Slots

An annotated slot is created and placed on an instance by the function create-annotated-slot. It takes four arguments. *object*, *slot-name*, *fetch-function*, and *replace-function*.

- *Object* is the instance that is receiving the annotated slot.
- *slot-name* is the name of the slot, from ndefstruct, that is being annotated.
- *fetch-function* is the name of the function that will be used as described in section 3.1.1.
- *replace-function* is the name of the function that will be used as described in section 3.1.2.

If the slot is annotated then the existing annotation will be nested inside the annotation being created. If a nested annotation is identical to the one being created, that is the same slot on the same object already is annotated with the same fetch-function and replace-function, then a new annotation is **not** created.

3.1.4 Removing Annotated Slots

If a slot is annotated, then the annotation may be removed and replaced with the local state of the annotation by the function *remove-annotated-slot*. The arguments to remove-annotated-slot are the same as the arguments to create-annotated-slot (see 3.1.3).

If the slot on the instance does not have an annotated slot with the same fetch-function or replace-function at any level of nesting, then no action is taken. An annotation is removed at any level of nesting, outer or inner layers are not disturbed.

3.2 Parallelism Using Annotated Values

Concurrency is introduced explicitly by using the *future* construct inside the definition of the fetch-function (section 3.1.1) or replace-function (section 3.1.2).

The effect of concurrently executing annotated slots is achieved by embedding both the call to **get-local-state** (or **put-local-state**) and the body of the rest of the demon in separate *future* forms. For example

```
(defun Trace-slot (object name annotation)
    (let ((result (future (get-local-state object))))
       (future (print-trace-info object name result))
       result)
    )
```

3.3 Programming Interface

To aid in the development of programmer defined fetch and replace functions two editor commands will be built for the editor residing on the front end machine.

Create Fetch Function Template will create a form from a template to be filled in by the programmer to define a new fetch-function. The template will appear as:

```
(defun <name> (object name annotation)
    "Fetch Function for getting the local state of ANNOTATION,
    an annotated-slot on slot named NAME of instance OBJECT"


    )
```

Create Replace Function Template will create a form from a template to be filled in by the programmer to define a new replace-function. It will be similar to the template used for Create Fetch Function.

3.4 Metering

Annotated slots are implemented as objects and all the metering tools defined for objects can be applied.

4. RULE SYSTEM ENVIRONMENT

A **Rule System Application** is a problem solver developed using an existing rule system. Example of a rule system applications are R1 and MYCIN.

A **Rule System** is a particular definition of rule structure, rule capabilities, and rule execution. Some examples of rule system are OPS5 and EMYCIN.

A **Rule System Environment** is the programming environment used to implement a rule system. In the past programming languages, with little additional support, have been used as a rule system environment; examples are Lisp and Bliss.

An objective of the Butterfly Expert System Tool Kit is to provide an environment for the rapid development of rule systems that will run on the Butterfly. This section describes the Rule System Environment.

The existence of Butterfly Common Lisp, Object Oriented Programming, and Annotated Slots constitute the beginnings of a Rule System Environment. To accelerate development and prototyping of rule systems some additional architectural foundations will be laid, some utilities will be defined, and we will establish some programming conventions.

The result will be the capability of several rule system applications, perhaps implemented in different kinds of rule systems, running concurrently on the butterfly. Following the guidelines outlined in this section, a rule system developer or experimenter, may build new rule systems by directly using or copying and customizing modules from existing rule systems.

4.1 Rule System Architecture

A Rule system, at a general level of description, is composed of four parts: a *processor*[9] (section 4.1.1), a *context* (section 4.1.2), an *access interface* (section 4.1.3), and a *rule development interface* (section 4.1.4).

---

[9]The term processor should not be confused with an individual cpu on the Butterfly as the term is used in other contexts. Here we mean the code that is responsible for the execution of the rule set, see below.

4.1.1 Rule System Processor

The **rule system processor** or simply **processor** is the set of programs that is responsible for executing the rules in the rule set. In many traditional rule systems most of the the program called the "rule interpreter" is the processor. In rule systems that translate rules directly into lisp code[10] the processor is trivial, it is simply the function call on the lisp code.

There is a wide range of trade offs available between the degree of "interpretation" as opposed to the degree of "compilation" of rules that would be executed by the processor.

4.1.2 Rule System Context

The **rule system context** or simply **context** consists of the pool of data that is accessible by the the rules and the "internal representation" of the rules employed by the processor.

In OPS5, the context is the working memory, the rule memory, and the rete network.

In the LOOPS rule system, the context is the set of variables used by the rules[11] and the function that implements the rule set.

In KEE's Rulesystem2, the context is the set of rules in a rule packet and the set of units in a knowledge base.

4.1.3 Rule System Access Interface

The **rule system access interface** or simply **access interface** is the set of routines

---

[10]This translation is handled by the rule development interface, see section 4.1.4. The resulting lisp code may call some run time support utilities, these utilities are part of the access interface, see section 4.1.3

[11]The variables used by a rule falls into several categories: a distinguished variable, self, that is a LOOPS object (an instance of the workspace class),other arguments to the rule set, the instance and class variables of self, rule variable and meta variables that are local to the rule set, and, InterLisp variables referenced in the rule set.

that are used by the processor (section 4.1.1) or the rules themselves to read or modify the context (section 4.1.2).

In OPS5 the access interface includes the *rete algorithm* and the modify and remove primitives.

In LOOPS the access interface includes the set of methods the workspace objects will respond to and the lisp code support to access local and global variables.

In KEE the access interface is the set of routines that match clauses in rules to units and modify units.

4.1.4 Rule System Development Interface

The **rule system development interface** or simply **development interface** is the "programming environment" presented to the knowledge engineer. The development interface is responsible for the acquisition and editing of rules, translating rules to a form suitable for the processor (section 4.1.1), and the initialization of the context (section 4.1.2).

In LOOPS the function DefRSM (define rule set as a method) is one of several parts of the development interface. It will initialize a rule set from a template in an editor, allow rules to be entered, compile the rules into a lisp function, and install the rule set.

In OPS5 the development interface includes a text editor (of the knowledge engineers choice) and the functions **P**, **lateralize**, and **make**.

4.2 Conventions

The Rule System Environment of the Butterfly Expert System Tool Kit establishes some conventions that would be used in the implementation of a rule system using the architectural guidelines described in section 4.1. By following the guidelines during the design and development of a rule system, a developer has the advantage of exploiting the work of previous developers.

A rule system application is a set of rules associated with a particular

processor, context, access interface, and development interface. Note that what KEE calls a rule packet, as well as what LOOPS calls a rule set, are rule system applications.

A particular rule system application will be invoked by name. It appears as a function call.[12] A rule system application may be invoked as an annotated slot by placing the function call to the application in the body of the appropriate fetch-function or replace-function.

The function associated with the rule system application name is the rule processor, it is defined by the development interface. The context consists of the arguments to the "function call" and perhaps some global structure in the lisp heap pre-allocated by the development interface. The processor will access the context via the access interface.

The following capabilities are expected to be provided by any rule system that is developed:

- The rule system application will be implemented as an object on the front end machine.
- The rule system application object handles the message down-load, a development interface message, that will do what ever processing is necessary on the text of the rule set and down loaded to the Butterfly Lisp environment. The eval service provided by the Butterfly to the Lisp machine can be used for this purpose.
- The down-load method should ensure that any initial context that is necessary be established, the access interface for the rule system is defined, and any interpreter support needed for the processor is defined.
- The entry point to the rule system application should be the name of the rule application that is being defined. The development interface should establish a unique name for the application.
- Every rule system be capable of reporting a rule firing per unit of time statistic[13]. To do this is should count the number of rules executes and divide the count by the execution time of the application. This would be reported via a rule-per-time message sent to the front end development

---

[12] Recall that in Common Loops, sending a message appears the same as a function call.

[13] To enable the necessary bookkeeping to do this calculation, it may be necessary to set a switch in the development interface and re-down-load the rule set

interface via eval service.

4.3 Example Rule Systems

To aid in the development of the Butterfly Rule System Environment, two example rule systems will be built following the conventions of section 4.1. They will display different functionality and demonstrate different degrees of implementation efforts for particular components of a rule system.

The first system is similar to LOOPS (section 4.3.1), showing the advantage of a strong development interface with particular attention paid to the careful generation of lisp code optimized for parallel execution.

The second rule system operates on a context containing a data base of propositional forms (section 4.3.2). It will show how a pattern matcher can be implemented, and show how rule scheduling and rule execution can occur concurrently.

4.3.1 Access Environment Rule System

The access environment rule system is similar in spirit to the LOOPS rule system. The major highlight of the access environment rule system is the notion of the *access environment*, a significant subset of the context. It also emphasizes a strong development interface that provides an integrated editing environment for rule acquisition and editing and the compilation of rule sets in to optimal, potentially highly parallel lisp code.

The access environment is a significant portion of the rule system context. It consists of an object, data that is accessible via the composition of accessing functions on the slots of the object, local variables defined within the packet, and lisp variables used globally.

Slots on objects visible in the access environment are called *rule variables*. The objects will be instances of classes whose meta class defines the notion of a property list on a slot as well as the usual slot value. These property lists on rule variables can be used to store certainty values and audit information.

The access interface will consists of several methods, a sample is:

- **Known-p**. Determine if a variable reference is a reference to a rule variable in the current access environment.
- **Get-rulevar**. Given the name of a rule variable, get its value.
- **set-rulevar**. Given a rule variable set its value.
- **set-certainty**. If a certainty value calculus is used set the certainty of a rule variable.
- **Access**. Given an access path on a variable, build and execute the composition of accessor functions that describe the access path.

The development interface supports a human readable version of the rules and the translation of the readable form into calls on access interface calls that can then be compiled into lisp code. This process is called *rule expansion*.

An example of a rule in human readable form is:

```
If   the number of missiles on our ship >
     the number of missiles on their ship
 Then our ship has more fire power than their ship
```

The same rule expressed in term of generated lisp code might appear as.

```
(If (> (future (length (access our-ship   missiles)))
       (length         (access their-ship missiles))
    )
  (progn (set-rulevar our-fire-power 'superior)
         (set-rulevar their-fire-power 'inferior)))
```

The rule translator will note control settings and compilation directives made by the developer and will generate lisp code that represents the rule set. The rule set control structures supported by LOOPS[14], additional control options that direct the generation of concurrent code will be supported[15].

A special case has been identified that allows for race free massive parallelization of the rules in a rule set. When the control structure of the rule set is doall and the rule variables accessed in the left hand side of a rules are disjoint from the variables modified in the right hand side of rules, then all rules in the rule

---

[14] e.g. do1, doall, while1, and while-all

[15] e.g. Parallel-rules, Parallel-lhs, parallel-rhs

set may test and conditionally act concurrently[16]. This concurrency may schematically expressed as:

```
(progn
   (future (if ...))
           .
           .
           .
   (future (if ...))
)
```

4.3.2 Propositional Based Rule System

The Propositional Based Rule System will contain a processor that represents a rule interpreter[17]. The context will consist of a rule index, a propositional data base, and some scheduling queues.

The propositional data base will be implemented using small frame system[18].

The processor will consist of several concurrent processes.

- A pattern matcher that monitors the data base and updates a table of partial rule instantiations. When a rule appears to be fully instantiated the matcher places the instantiated rule in the scheduler queue.
- The scheduler monitors the scheduler queue and selects rule instantiations from it for execution, the scheduler will allow rules to execute in parallel if it can tell they do not conflict. For conflicting rules, the scheduler plays the role of conflict resolution establishing an execution order relation across rules or chosing not to execute part of the conflict. An interesting feature of the scheduler is that *it may be implemented as a rule set.*
- The rule executer is responsible for rule execution. It can run rules either forward or backward depending on context.
- The data base maintainer receives messages to place new propositions in the data base.

---

[16]The disjointness property is an extremely strong necessary condition. Work is in progress to attempt to weaken the condition

[17]Contrast this to the Access Environment Rule System where the processor was simply an interface to call the function generated by the rule translator

[18]By small we mean just barely beyond that of a text book example, however functionality will be demonstrated so that a more industrial strength frame system can be plugged in. We are considering KANDOR as an example.

The access interface includes the queuing operations, and the Common Loops accessor methods for the frame data structures.

4.4 Rule system metering

The Common Loops meters may be applied to those parts of a rule system that are implemented in an object oriented manner.

See also section 4.2 which describes a rule firing per second meter for a rule application.

END

DATE

FILMED

3-88

DTIC