(2)

September 1991

CLF MANUAL

CLF Project

USC Information Sciences Institute

4676 Admiralty Way

Marina Del Rey, California 90292

This research is supported by the Defense Advanced Research Projects Agency under Contract No. MDA903-87-C-0641 and by the Naval Ocean Systems Center under Contract No N66001-87-D-0136/#40.

Table of Contents

List of Figures

List of Tables

NOTATIONS and TERMINOLOGY

- Lisp, unless otherwise qualified, refers to the Common Lisp Language, as described in *COMMON LISP - The Language, Second Edition, Guy L. Steele Jr, Digital Press* (CLtL).

- Names of macros, functions, variables, relations, commands, keywords and arguments are shown in **boldface type**.

- Descriptions of syntax follow the conventions laid out in section 1.2.5 of the above mentioned Common Lisp language document.

1. INTRODUCTION

Common Lisp [CL] is a programming language with an expressed goal of providing a common dialect of Lisp that will be adopted by a broad segment of the Lisp programming community. Common Lisp has been implemented by several commercial hardware and software vendors and is in wide use within both academic and industrial research and development centers. A standards committee is actively preparing a proposal for an ANSI language standard.

Several sophisticated and quite diverse programming environments had grown up around various Lisp dialects since the appearance of Lisp 1.0 over twenty years ago. Since 1980 these environments have evolved to take advantage of more powerful processors and user interface hardware and modern window interface concepts. For many indivividuals, these environemnts are one of the major attractions of Lisp. In fact, many development efforts have been able to avoid, or at least postpone, significant costs by making innovative uses of the Lisp programming environment tools as part of the "run time" environment of their applications. Nevertheless, no programming environment standard has been established for Common Lisp, nor is one envisioned.

Although CL provides no window interface standards of its own, all major commercial implementations of CL are able to act as clients for X11 user interface servers via the CLX library, non-proprietary software providing functionality similar to that found in the XLIB library used by C language programmers to program clients for X11 servers. CLF relies on CLX for its window interface.

Several major activities in software development involve the manipulation of text—program source code, textual input and output of applications being developed, and the input and output of instrumentation and debugging tools. CLF relies on Epoch (an extension of Gnu EMACS) for all text manipulation. CLF's user interface is therefore comprised of windows belonging to CLF itself and windows belonging to Epoch. The content of the latter windows is provided by CLF and by user interactions.

Several hardware vendors continue to support high quality program development environments for the implementations of Common Lisp on their machines. The Common Lisp Framework [CLF] differs from these environments in three major ways:

- CLF provides an object based, rather than file based, organizational view of software. The object based view comprises not only the definitions that make up an application, but specification, documentation, development history and other non-procedural information necessary to the development, maintenance, and distribution of large software systems.

- CLF strives to provide an "open" architecture, occasionally even at the cost of considerable efficiency, to enable programmers to tailor and extend the programming environment to meet their individual needs without the necessity of reimplementing the existing environment.

- CLF has been produced and is maintained by a non-commercial organization, the University of Southern California's Information Sciences Institute (ISI). It is not targeted at any particular vendor's CL implementation or hardware, but tries to provide a highly portable programming environment that interfaces naturally to the native operating system environment on each supported CL implementation. Originally developed on special purpose "lisp machine" hardware produced by Symbolics, Inc., and Texas Instruments, CLF has now been ported to Unix-based workstation platforms. In particular, it can run under both Allegro and Lucid CL, and has been tested with these software platforms on both HP300 series workstations and SUN Microsystems SPARC workstations.

2. SPECIFICATION LANGUAGE

2.1. AP5

Software developed under CLF may be written in an extension to CL called AP5. AP5 affords several advantages to a programmer over the use of pure CL in writing applications. First, programs can be written using a relational notation for access to data. The programmer thereby avoids early commitments to particular data structures and algorithms for accessing that data. By adding *annotations* to the program, the programmer can guide the AP5 compiler's selection of data structures and algorithms. The resulting "object code" is CL with limited reliance on run-time facilities that are themselves written in CL. Because the annotations can only affect the *efficiency* of the resulting object code, not its *functionality*, we consider the use of AP5 to be a specification based programming paradigm. In addition to the use of relations for describing data access, AP5 affords the programmer the ability to define *rules* as a means of specifying some processing that would have to be specified procedurally in pure CL. Two sorts of rules are provided.

Consistency rules define, using an extended first order logic notation, invariants that must hold at all times on the data. AP5 produces code that detects any attempt by a program to modify its data in a way that would violate one of these invariants. It can then reject the modification and signal a handleable error to the program. Alternatively, the programmer can associate with a consistency rule a repair procedure. This procedure, which is given access to the invalid data, has an opportunity to incorporate additional data modifications that will restore the rule's invariant. If it is able to do so, no error need be signalled to the program. AP5 provides a number of utilities, based on consistency rules, to define class hierarchies and incorporate class restrictions into CL.

Automation rules are analogous to "whenchanged" rules in some AI languages, although they provide more power than procedural attachment in frame-based paradigms. The programmer is able to define *transitions* of interest to him. An automation rule consists of the specification of such a transition together with a response procedure. Whenever a program modifies its data in a way that instantiates a rule's transition specification, its response procedure is applied to the relevant data.

CLF itself relies on AP5. Even if CLF is used manage the development of applications that are independent of AP5, the use may wish to take advantage of AP5 to tailor and extend CLF to better meet personal or organizational needs. This document describes the most significant relations that are used in CLF's implementation -- the relations most likely to be needed in writing new rules or defining new derived concepts. However, due to the significant amount of material needed to document AP5, and because AP5 may be used in any CL programming environment, independent of the presence of CLF, reference documentation for AP5 is not included here. Consult the AP5 Reference and Training Manuals.

AP5 uses the term "type" as a synonym for "unary relation". This usage does not match the traditional concept of types in programming languages, although it is consistent with the extended notion of type offered by CL's DEFTYPE primitive. In this manual, the term "type" will be used in the AP5 sense unless otherwise noted. An object "belongs to" or "is an instance" of a type in any database state in which the one-tuple consisting of that object is a tuple of the type.

The term "attribute" in this document is synonymous with "binary relation".

3. TASKS

Although CLF runs as a single OS process, communicating with another process, Epoch, to provide a user interface to text and with an X11 server to provide a windowing interface, internally CLF may be interleaving multiple tasks for the user. Tasks may be busy building user interface displays, compiling programs, installing programs, testing an application under development, etc.

CLF implements multitasking with multiprocessing extensions to CL that have been provided by the vendors on all platforms to which CLF has been ported. In all cases, these extensions provide a preemptive scheduler; tasks may be interrupted at almost any time. Some of these platforms have a priority scheme imposed on the scheduler. Where possible, CLF tries to give its user interface manager a relatively high priority.

AP5 provides considerable synchronization at points of database access; a task that wants to modify the database may block for a noticable period of time while another task has access to the database. With this exception, users are aware of multitasking mainly because of the user interface support of interactive tasks, described in Section 4.1.1.

4. USER INTERFACE

CLF's user interface requires a bitmap display controlled by an X11 server, a pointer device, and a network server connection to an Epoch process. The CLF process, X11 server, and Epoch process may be running on any physical hosts providing suitable network connections. CLF makes limited use of color in its displays; the interface is usable on a monochrome display, but some distinctions are lost.

A typical CLF screen consists of a help window, a "System Operations button", editor windows, and object views. Figure 4-1 exemplifies a CLF screen. CLF does not provide its own interfaces for moving, deleting, resizing, restacking, or iconifying windows. Its windows obey the X11 X11 Inter-Client Communication Conventions for interacting cleanly with an X11 window manager. The help window is an output-only area; as the user moves the pointer over various object depictions in object views, text will appear in the help window summarizing the effect of the three mouse buttons.

The System Operations button is a small window from which a menu of useful operations can be exposed. Each member of the initial set of operations is described in this document in the section relevant to its functionality. Selecting an operation from this menu will cause the operation to be performed as a new task.

Object views are of three varieties:

- *Browsers* display a static view of a focus object and the relationships in which it participates. By selecting visible depictions of objects related to the focus object, the user changes the focus. A history of focussed objects is maintained; the user may display a menu of these and reselect any one as the current focus.

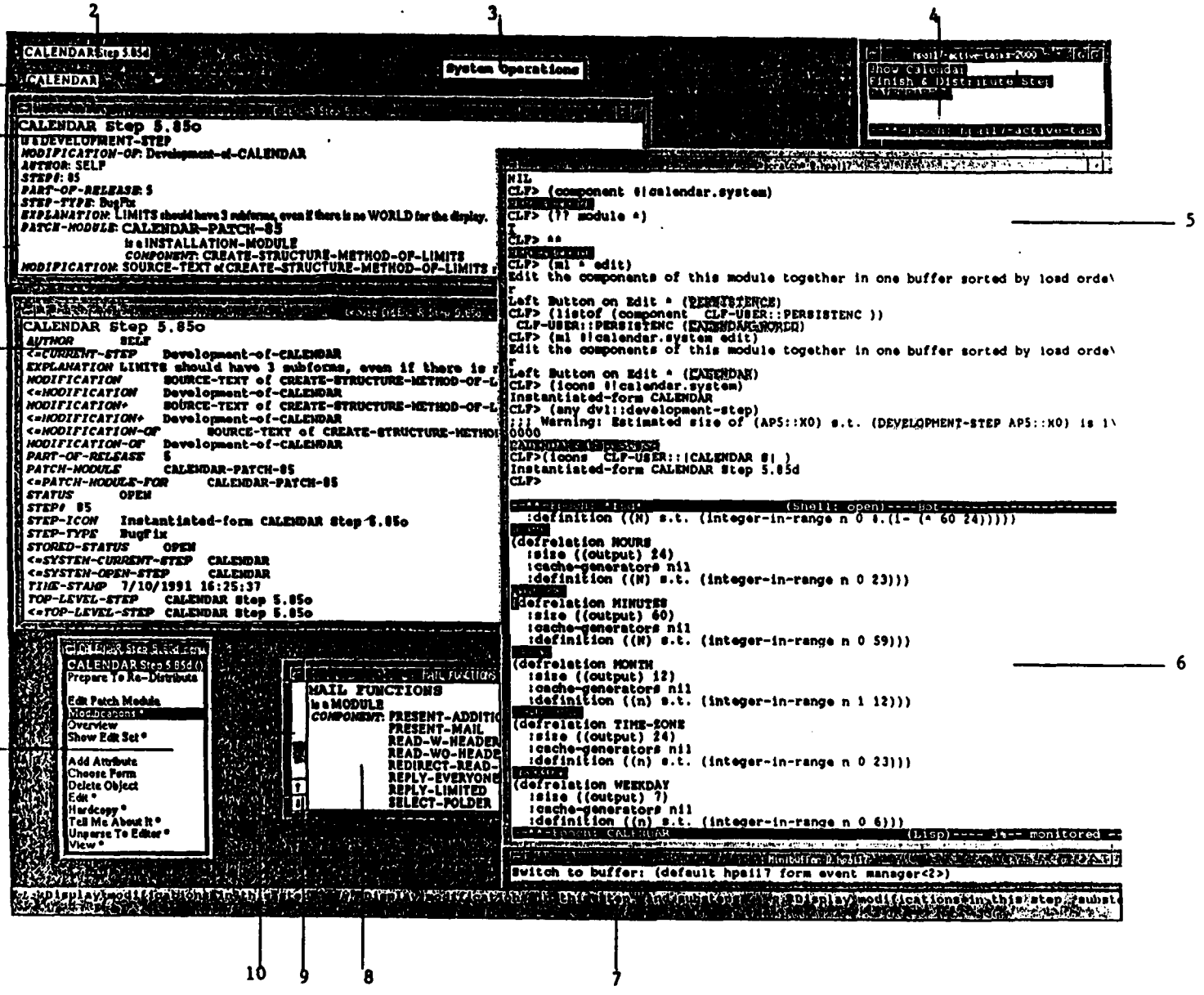
- *Dynamic Views* display a selected view of information in CLF's virtual database. Commonly this information consists of a focus object and a collection of its attributes. The information presented in a dynamic view is synchronized with the virtual database; at screen synchronization points, these displays are updated as needed to reflect the current state of the view.

- *Icons* are truncated dynamic views. Each icon displays only the name of its focal object. Icons require little screen real estate; they are useful because of the standard operations that can be invoked from them using the mouse.

Whenever an object view contains one or more vertically or horizontally scrollable subviews, an appropriate scroller will be attached to the window. A scroller consists of three elements:

- The **analog** control displays the percentage of the overall view that is currently visible and its relative position within the overall view. By clicking the mouse in the analog control the user can select a different portion of the overall view to be visible.[1]

[1] Eventually the analog control will support continuous drag.



1,2 Iconic views
3 System Operations
4 Task Status
5 Commmand Shell
6 Module being Edited
7 Help Window
8 Dynamic View (Module)
9 Forward Scroll
10 Analog Scroll
11 Menu
12 Browser (Step)
13 Dynamic View (Step)

Figure 4-1: CLF User Interface

- The **backward** control allows the user to discretely scroll the view to make portions nearer the start visible. The three buttons scroll by one unit, and entire viewport's worth, and "all the way to the start".

- The **forward** control is just like the backward control, but makes portions nearer the end of the view visible.

4.1. The Epoch Interface

4.1.1. The Task Interface

Each task initiated by CLF is assigned its own Epoch buffer for textual interactions with the user[2] The buffer and task are assigned a mnemonic name. For tasks generated by menu selections, this name is the text that appeared in the menu item selected[3]. The text in these buffers may be edited with the full complement of Epoch editing commands.

CL defines several variables that should always be bound to input, output, or input-output streams. When CLF starts a task, it binds these varibles local to that task to the task's buffer.

CLF provides a display of active tasks that are in certain distinguished states. It is based on a simple finite-state model, shown in figure 4-2. Each state has an associated color. CLF maintains a single buffer in Epoch, named <CLF-HOST>-active-tasks-<nnnn>. <CLF-HOST> will be the name of the host on which the CLF process is executing; <nnnn> is a four-digit number that distinguishes that CLF process from any other CLF processes that may be running on the same host[4]. This buffer, hereinafter referred to as the task status buffer, will display one line for each task that is currently in a distinguished state. The text is the name of the task's buffer. It is displayed over a background whose color designates the state.

The distinguished states are:

- Output Available (green): The task has produced some output in its buffer that the user may not have seen.

- Output Available/Task Terminated (black): This indicates that unseen output is available, and further that the task has terminated.

- Input Block (red): The task is blocked waiting for the user to provide input to its buffer.

---

[2] A buffer is Epoch's largest contiguous textual unit. An Epoch process may manage any number of buffers.

[3] Epoch may append a few characters to the buffer name to guarantee uniqueness.

[4] A single Epoch process may provide edit services to multiple CLF processes, and a single CLF process may utilize services from multiple Epoch processes. A similar many-many relationship exists in CLF's window interface. This makes it possible for a user at one workstation to tie into another user's CLF process and both view and modify data.

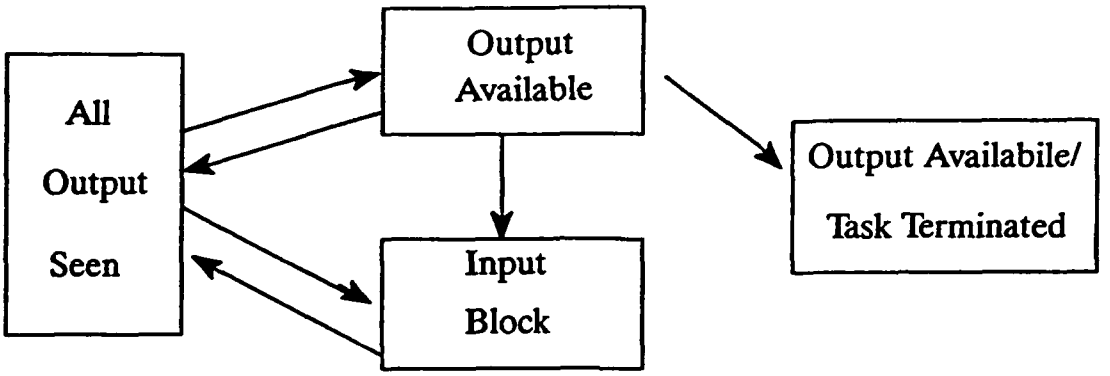


Figure 4-2: Task state model

The user may select a task's buffer through the usual Epoch buffer selection mechanisms at any time, regardless of whether the task is in a distinguished state. When a task is in a distinguished state, the user may expose and select[5] its buffer by clicking a mouse button on the line displaying it in the active tasks buffer. This will remove its entry from the active tasks buffer. In the case of the Output Available state, it will also consider the user as having "seen" the available output.

When a process is in the Output Available/Task Terminated state, a line containing "Click here to free this buffer" is placed at the end of the buffer. Clicking any mouse button over this text will free the buffer and remove any entry for the task from the active tasks display. Clicking the middle button over the entry for a terminated task in the active tasks display will remove the entry and free the buffer immediately, without exposing it.

When a terminated task's buffer is eliminated, the content of that buffer is appended to a "transcript buffer" that CLF establishes when it creates its connection to Epoch. The transcript buffer will be named "transcript buffer <CLF-HOST>-<nnnn>", where the host and four-digit number are the same as for the task status buffer.

[5] Exposure refers to making the buffer content visible in an Epoch window. Selection means that interactive editing commands will be interpreted relative to that buffer.

4.1.2. Command Shell Tasks

A CLF user typically runs one or more command shell tasks. A command shell task runs a lisp read-eval-print loop, in which the user enters a lisp expression (program) into the command shell's task buffer. The expression is then parsed (read), executed (eval), and its value(s) output (print) to the command shell's task buffer. The active tasks buffer does not display the state of command shell tasks, because they are typically already the focus of the user's attention[6].

4.1.3. Editing Modes

Each task buffer has an associated *editing mode* that may change depending on the use being made of that buffer by the task. The primary function of the mode is to determine:

- *when* user input is to be transmitted to the task. Most modes provide some buffering so that the user can prepare, inspect, and modify input before transmission.

- *what* text to transmit. A task buffer gives the user the full range of editing power of Epoch. Text in the buffer may be arbitrarily modified, and cut-and-paste may be used to prepare input.

When a task buffer is visible, its editing mode will be displayed in a status line, which also shows the buffer name, below the visible portion of the buffer's text. The modes employed by CLF are:

- Explicit — Each task buffer keeps track of the position of the last character tranmitted to it by the task. In explicit mode, a transmission always consists of all text in the buffer following this location or the end of the last transmission, whichever is later. The keyboard command c-z z (the "control" key depressed concurrently with a "z", followed by another "z"), initiates transmission. The keyboard command m-<return> (the "meta" key depressed concurrently with the <return> key) inserts a new line at the end of the buffer and then transmits.

- Response — Response mode is almost identical to explicit mode. The only difference is that the starting point of text to be transmitted is the end of the buffer at the time it entered response mode. CLF tasks use this mode to provide the user with text to modify and send back. A task typically sends some instructions for the user into its buffer, then places the buffer in response mode, and then sends the text to be modified into the buffer. The user makes the desired modifications, then types c-z z to transmit the text back to the task.

- Line — This mode is used by interfaces that want a single line of text from the user. Transmission occurs only when a <return> is entered with the text cursor positioned at the end of the buffer. The final line of text in the buffer is transmitted, even if that line is empty.

- Character — This mode is used by "completing" interfaces. In character mode keystrokes that normally cause insertion of single characters (including the <return> key) are transmitted immediately to the task. The character is not inserted in the buffer by the editor; any echoing is performed by the task upon receipt of the character.

- Shell — This mode is used in command shell task buffers. In this mode, a <return> typed at the end of the buffer initiates trensmission if and only if the text between the command

[6] Soon it will be possible to have command shell tasks included in the active tasks display at the user's option.

shell prompt and the return constitutes a well-formed lisp command[7].

4.2. Editing Objects

In addition to buffers that provide an interface to command shells and other interactive tasks, CLF provides buffers in which users edit textual representations of objects. When the user is editing text in task buffers, a task is explicitly waiting for the user's transmission. In the case of buffers that are editing objects, a new task is created to deal with any changes transmitted by the user.

This is a generic facility, supported for any type of object that provides "parsing" and "unparsing" methods. The only such type documented here is software definitions.

A buffer may present an entire module, a single definition, or a collection of definitions and/or modules that satisfy some condition. In any case, what the user sees is a sequence of individual objects. For each, there will be a single read-only line with the object's name, followed by the object's source text. If the object has a documentation attribute, the documentation text is presented, labeled as such, above the line with the object's name. These labels initially have a green background.

The user may change the text at his own initiative, other than in the labeling lines. When the text of a definition or documentation unit has been changed in the buffer, but not yet transmitted back to CLF as the new version, its label's background changes from green to red.

The user may transmit the new text of all modified units in a buffer by typing the keyboard command c-z z. If the new text for a unit is acceptable[8], the unit's label will revert to a green background. The labels of unaccepted units will remain red; in addition, the task rejecting the new text will print a notification in its task buffer.

If a unit has been modified, but the change has not yet been committed back to CLF, the user may restore the text to the version currently held by CLF. To do this, the user enters the command m-x Revert-Object while the text cursor is within the object to be reverted.

The user may also selectively dispose of the objects that have been changed but not yet been committed back to CLF. The command m-x Dispose-of-Changed-Objects presents the user with a check-off menu. For each modified unit, the user may choose to commit the changes, revert to CLF's current version, of simply leave the unit in its modifed state.

---

[7] Well-formedness is heuristically determined. Basically, any text with no parentheses or with balanced parentheses is deemed well-formed.

[8] In the case of a documentation unit, any text is acceptable. In the case of a software definition, syntactic well-formedness is required.

The text cursor may be move forward and backward over entire units. c-z f moves the cursor to the start of the next unit; c-z b moves to the start of the previous unit. Both commands accept numeric arguments.

4.2.1. Hypertext

When CLF prints objects to task buffers, it prints them as a form of hypertext[9]. The hypertext objects appear over a background color that contrasts with the buffer's overall background. The user can perform operations on hypertext objects with the mouse, as described in section 4.4.

4.3. Interface-Database Synchronization

Dynamic object views are synchronized with changes to the database in two phases. The database undergoes *atomic* updates that lead from one consistent state to the next. On each such state transition, sufficient data is recorded to synchronize affected object views to the new state. The synchronization currently only happens automatically in three ways. First, at the start of each top-level iteration of a command shell the interface is synchronized to the current database state. Second, each task created from a menu selection requests synchronization when it completes. Finally, a small set of operations that can be initiated by commands given from Epoch (see section 4.2) request synchronization when they complete.

4.4. Using the Mouse

Wherever CLF displays a depiction of an object, whether in an object view or as hypertext[10], three standard operations are available:

- Left button: The object will be entered into the current Epoch buffer at the position of the text cursor. This is useful when entering commands interactively to a read-eval-print interace, such a a listener or a lisp debugger. If the object has a readable print representation – e.g., it is an integer or a string – that print representation is used. Otherwise, a lisp symbol will be created and globally bound to the object, and the symbol will be printed in the buffer.

- Middle button: A browser will be prented with the selected object as its focus. If the selection is made from a browser display, the focus of that browser is switched to the selected object.

- Right Button: A menu of operations applicable to the selected object is presented to the user. The content of the menu is sensitive not only to the object's type, but to the current state of the database as well. In the menu operations appear in groups, separated by vertical space. A group corresponds to operations defined at the same layer of type hierarchy. Selecting an operation from one of these menus will cause the operation to be performed as a new task.

---

[9] Currently, only some objects are printed in this way

[10] With respect to mouse inititated operations, the label of an object in a buffer editing the object behaves exactly like a hypertext depiction of the object.

Each command shell is initialized to have package CLF-USER and readtable CLF. This is a CL readtable with the addition of a read macro, #!. #!<string> reads in as an expression whose value is any object whose name is <string>. If <string> contains no whitespace characters, the string quote delimiters may be omitted. In this context, an object may have many names. Any string or symbol that is the value of one of the attributes in the list bound to the variable AP5::*NAME-ATTRIBUTES* will be considered a match. The possibility of ambiguity can be reduced by specifying a required type for the object. This is done by following the name with a "." and a typename -- e.g. #!email.system.

Clicking on the System Operations button with any mouse button will present a menu of operations that are not obtainable from available objects' menus.

4.5. The Keyboard

Although CL has no "standard Keyboard," all workstations having Cl implementations are likely to contain several (non-shift) keys in addition to the standard QWERTY set. If the CL implementation makes these keys transmit new CL characters, they can be useful in the interface for that workstation.

We have identified several "generic uses" of these extra keys, described below. For each, there are two special global variables in the implementation. One is bound to a string, the label found on the appropriate key for that implementation. The other is bound to a character object, the character transmitted by the key for that implementation.

For the sake of portability, applications programs, as well as CLF kernel programs, should use these variable rather than their literal values.[11] For example:

```
(format *query-10* "Type your answer. ~
   <press the ~a key to quit the operation>"
   *QUIT-KEY-LABEL *)
(LET ((response (Get-Users-response)))
(when (EQL response *QUIT-KEY*) (throw :operation-quit NIL)[12]
```

4.5.1. Quit

Sometimes it is desirable to let a user drop out of a dialog if he decides he does not wish to perform an operation, or has made an error earlier in the dialog. *QUIT-KEY* and *QUIT-KEY-LABEL* are in intended to be used for this purpose.

---

[11] The character variable should always be compared with EQL, of course.

[12] The character variable are declared with DEFCONSTANT. The label variables are declared with DEFParameter, to account for the possibility that a given implementation may run with several keyboard variants.

4.5.2. Menu

In some input contexts, a user is given the choice of entering a response from the keyboard or mouse, or of asking for a menu of possible responses. *MENU-KEY* and *MENU-KEY-LABEL* are used for this purpose.

4.5.3. Activate

In some input situationd, the user needs a special gesture to indicate that he has finished entering his response. *ACTIVATE-KEY* and *ACTIVATE-KEY-LABEL* are used for this purpose.

4.5.4. Complete

In some input contexts, the user can ask the system to supply some of his input automatically, usually echoing it as if it had been typed. *COMPLETE-KEY* and *COMPLETE-KEY-LABEL* are provided for this purpose.

4.5.5. Help

Whenever an application is awaiting user input, it is desirable to provide the user a means for obtaining an explanation of what is expected from him (e.g., an address for a message), how it may be provided (type an address or pointer to a person or mailbox), and how it will be used (as a CC recipient of the message you are composing). *HELP-KEY* and *HELP-KEY-LABEL* are provided for this purpose.

4.6. Browsing

4.6.1. Browsing by Pointing

The *dynamic browser* is a mechanism for finding information of interest in the database when you are not certain of the relational path you need to reach information. It is a blend of the TELL-ME-ABOUT facility and the standard form windows.

The browser is invoked by applying the function SELECT-AND-SHOW-ATTRIBUTE-VALUES to an object, or by clicking the middle mouse button on an object in a form window or command shell. The browser appears (near the mouse) in a window displaying the object, its most specific classification(s), and all attributes and their values.[13] From this display, all the operations that can be initiated from a standard CLF form window can be performed, using the same gestures. Moving the mouse outside the window causes it to disappear, just like a temporary menu. Clicking the middle button on one of the

[13]Someday this may extend to show information about occurrences of the object in relations of arity greater than 2, as TELL-ME-ABOUT does.

values displayed in the window also causes the window to be replaced by a dynamic browser viewing the selected object. A small area labeled "CHAIN" can be moused to provide a menu of all objects viewed in a session with the dyanamic browser (excluding the one currently being viewed). Selecting one of those objects will replace the current window with one viewing the selected object.

4.6.2. APROPOS

APROPOS *function*

APROPOS-LIST *function*

APROPOS-LIST *variable*

APROPOS-LIST* *variable*

APROPOS-LIST** *variable*

APROPOS-WILDCARD *variable*

The common lisp functions APROPOS and APROPOS-LIST provide a way to find symbols whose name contains a given substring. A package argument determines a subspace of all known symbols in which to search. CLF extends these two functions to allow search spaces other than packages. The first argument is still a string. If the second argument is a package or package name the CLF functions execute their common lisp counterparts.

The CLF functions extend their counterparts primarily by allowing search spaces other than subsets of symbols. The second argument is the search space designator. If it is the name of a type, then the search space is all objects of that type. If the designator it is an instance I of some type for which a Default-Structuring-Attribute A has been declared, then the search space consists of I and all objects reachable from I via A.[14] For example, the Default-Structuring-Attribute of the type Module is Component. So if the search space designator is a module, the search space includes all of that module's direct or indirect components. The name of each object in the search space is matched against the string pattern. Matching objects are either printed (APROPOS) or gathered into a list that becomes the value (APROPOS-LIST).

In the standard CLF release, the Default-Structuring-Attribute of the type TYPE is Isubrel, which relates a type to its immediate subtypes. Therefore, if an instance of the type TYPE (as opposed to the name of a type) is used as the search space designator, the space searched consists of that Class and its

[14] In other words, all objects in the transitive closure of I under A.

subclasses, not instances of those classes.[15]

Uses of CLF's apropos (other than the package searches provided by CL) permit a more general string pattern. The string may contain any number of occurrences of the character bound to the special variable *APROPOS-WILDCARD* (initially #\$). Appearances of this character may match arbitrary (including empty) subsequences of an object's name – e.g., the pattern "hassup$" matches both the names "has-any-supervisor" and "hassupervisor". The matching uses CHAR-EQUAL, so it is case insensitive. If the string pattern provided as the first argument to CLF's APROPOS contains no occurrence of *APROPOS-WILDCARD*, a wildcard is automatically appended to the front and end of the given string. Finally, a pattern may be a list of the form (and . patterns) or (or . patterns), with the "obvious" interpretation.

The search space for the CLF extensions to APROPOS may be augmented by passing further arguments to the function. All arguments after the search space designator may be (the names of) attributes. In this case, the string pattern is matched not only against the name of each object in the search space, but also against the name of each value of each designated attribute of that object. A match with the object's name or the name of any of the values of any of the attributes is sufficient for including the object in the set of successes. For example, (APROPOS "(reply" M 'Source-Text) would find every component of a module M that either had the string "(reply" as part of its name or as part of its source text.[16]

Finally, the symbol T may be included in the list of attributes. This serves as a shorthand for including every attribute declared to be a Standard-Apropos-Attribute for the search space class or any of its superclasses.[17]

4.6.3. Tell-Me-About

The function Tell-Me-About (documented in the AP5 reference manual) provides a means of mapping over all true tuples in which a given object appears.[18] The default functional parameter causes the tuple to be printed on the *standard-output* stream, which is the typical use of Tell-Me-About as a browsing tool. When the stream is a command shell, the related objects printed in the tuples can then be selected

---

[15]Retracting the Default-Structuring-Attribute of the class Class will result in use of a class and a class name being equivalent as search space desigations.

[16]The fact that some components have no value for the Source-Text attribute, and could not even logically have a value for the attribute, causes no error. For those components, only the name of the component is matched agains the string.

[17]The search space class is determined by the search space designator. When the search space is determined by an instance and a Default-Structuring-Attribute, the search space class is the declared range restriction for that attribute. When the search space is all instances of a class, that class is the search space class.

[18]Subject to generability considerations.

with the mouse.

4.7. Menus

There are two kinds of menus which appear generically in CLF's user interface. One is the menu of operations on a particular object, which typically appears when the user selects a depiction of the object with the right mouse button. The other is the menu of "system operations", exposed by selection the System Operations button. The contents of these menus may be modified with the macros described in this section.

DEFINTERFACE-COMMAND [NAME CLASS &key (TEST T) (AUX NIL) (MENU-HELP-STRING NIL) (LEFT-HELP-STRING NIL) (MIDDLE-HELP-STRING NIL) (RIGHT-HELP-STRING NIL) (INTRO-HELP-STRING NIL) MENU-LABEL (PROCESS-IN-LISTENER T) (BUTTON-SENSITIVE NIL) (BODY NIL) (BODY-LEFT NIL) (BODY-MIDDLE NIL) (BODY-RIGHT NIL) ...] *macro*

Defines an interface command (menu-item and associated action) to be offered for instances of CLASS which satisfy TEST. TEST is an arbitrary form, which may use the variable mouse-object freely. Mouse-object will be bound to the object selected with the mouse. If TEST evaluates to NIL, the menu item will not be offered in the menu for the object that failed the test.

MENU-LABEL is a form evaluated at menu generation time that evaluates to a string to display in an operation menu.

BODY-LEFT, BODY-MIDDLE, and BODY-RIGHT should be forms to execute when the respective mouse buttons are clicked on this menu item. If all three buttons have the same response, a single code body may be provided with the BODY keyword rather than being duplicated for each button. These forms may also use mouse-object freely. If BODY is given as well as one or more of BODY-LEFT, BODY-MIDDLE, and BODY-RIGHT, then the BODY form applies to all mouse clicks not explicitly given a form to execute. That is, if BODY-LEFT and BODY are given, then both middle and right clicks will cause the BODY code to execute.

INTRO-HELP-STRING is a string or a form that evaluates at menu generation time to a string that is the first element of the string that appears in the help window when the mouse is over this item in the menu. It should be a string applicable to the menu item regardless of which button is pressed.

LEFT, MIDDLE, and RIGHT-HELP-STRING also appear in the help window. If given, MENU-HELP-STRING is used for any of the LEFT, MIDDLE, or RIGHT-HELP-STRINGs that are not given. Heuristics are used to prevent repeatedly printing the same string in the help window; for example, if only a MIDDLE-HELP-STRING and a MENU-HELP-STRING are given, the second line of the help window will appear as:

```
"L: menu help string here  M: middle-help-string   R: same as L"
```

The string-valued forms may also refer to mouse-object. The INTRO-HELP-STRING is printed on the first line of the help window, with the rest of the help strings printed on the second line.

The AUX keyword has as its value a list of aux variables. Each element of the list is either a symbol, in which case the variable is initialized to nil, or a pair, the car being a symbol and the cadr being an initialization form. The initialization forms may refer to mouse-object.

The TEST returns a value treated as a boolean. But it may use the aux variables (reading their initial values) and as a side effect may set them as well. If the test returns non-nil, the final values of the aux variables are retained. They are then made available to the MENU-LABEL, HELP-STRING, and BODY code. These may read the aux variables, and will see the values they had at the end of the TEST. But they cannot change the values of the aux variables in order to communicate with each other.

> **Rationale:** Although frequently the label and help-strings are constants, and occasionally the test is simply T, all of them as well as the body, may be arbitrary computations that have access to the OBJECT that was buttoned to create the menu and the qform/window as well. In order to make the menus come up fast, it is important that the test, label, and help-string computations be efficient.
>
> The content of the label and, more importantly, the help string, influence the users choice of whether to select an item, and which button to use for button-sensitive items. Thus the results of computations that go into computing this text MAY be needed in the BODY. There are two reasons why REcomputing them may be UNSAFE, regardless of their efficiency.
>
> o The computations may be non-applicative -- e.g., rely on the database. The database is not locked while the menu is up, and, for that matter, all that happens when an item is selected is to QUEUE up the body action to happen when the listener process next checks its queue. So redoing the computation in the body may not yield the same result as at menu creation time.
>
> o The computation may be non-deterministic. For instance, there is no guarantee of retrieval order consistency from the database for multiple valued attributes, so it might be necessary to introduce some arbitrary SORTing to ensure that a recomputation got the same results.

Instances of a subclass will inherit the interface commands for all superclasses. However, NAME is used to provide a shadowing mechanism. If an interface command for a subclass is written with the same name as one for a superclass, the one for the subclass will "shadow" the one for the superclass. In particular, defining such a command with a test of NIL will block the command from appearing as a choice for

instances of the subclass (and is often preferred to adding a condition to the test in the interface command for the superclass.)

DEFSYSTEM-OPERATIONS-COMMAND [NAME &key ...] *macro*
Defines a menu item for the menu associated with the "System Operations" window in a CLF frame. See DEFINTERFACE-COMMAND for an explanation of the keyword parameters. Any command already defined with the name NAME is replaced. The response functions take no parameters. Because these menu items are not defined for particular types, there is no issue of shadowing.

UNDEFINTERFACE-COMMAND [NAME CLASS] *macro*
Removes the interface command named NAME for CLASS.

UNDEFSYSTEM-OPERATIONS-COMMAND [NAME] *macro*
Removes the system operations command named NAME.

Menu selection can be simulated by entering calls on the macros described next in the command shell. These macros take a "pattern" argument. Depending on the specificity of the pattern provided in a given call, an operation may be selected without even exposing a menu, or a reduced menu may be exposed. These macros make it possible to display a menu of operation for an object for which no depiction is available for selection.

ML [X &optional (Pattern "")] *macro*
Simulates selecting the item whose label matches Pattern from the menu for object X. X is evaluated. Pattern is quoted and may be a string or symbol. For a symbol pattern, the symbol's print name is used. If exactly one item in the menu for X would have a label matching Pattern, the action that would be performed by selecting that item with the LEFT button is performed. If more than one item's label would match pattern, a menu of the matching items is presented and the user may select from it. (If any of the items in this menu are button sensitive, the button sensitivity applies. In other words, even though the menu was generated with ML, a non-left-button operation may be chosen.)
Matching is defined by case-insensitive string comparison. A $ appearing in Pattern acts as a segment wildcard. If no $ is explicitly present in Pattern, one is appended to the end. (Thus there is no way to force an exact match.)

MM [X &optional (Pattern "")] *macro*
Like ML, but simulates pressing the middle button.

MR [X &optional (Pattern "")] *macro*
Like ML, but simulates pressing the right button.

M [X] *macro* Generates a menu for object X exactly as if it had been obtained by right clicking on a presentation of the object.

CLFL [&optional (Pattern "")] *macro*
Like ML, but for the System Operations menu.

CLFM [&optional (Pattern "")] *macro*
Like MM, but for the System Operations menu.

CLFR [&optional (Pattern "")] *macro*
Like MR, but for the System Operations menu.

CLF [] *macro* Like M, but for the System Operations menu. Generates a System Operations menu just as if the System Operations button had been selected.

4.8. Miscellaneous

Each command shell runs a fairly standard lisp read-eval-print loop. Some tailoring of this interaction is possible.

TOP-LEVEL-READ-EVAL-PRINT [] *function*
A recursive Read-Eval-Print loop. Uses the current value of *terminal-io* as the stream for reading/printing. This is an infinite loop. The only exit is via a THROW or some non CL mechanism. An anonymous restart is provided whose select will restart the loop. TOP-LEVEL-READ-EVAL-PRINT establishes new bindings for and maintains the Lisp variables *, **, ***, +, ++, +++, /, //, and ///.

TL-PROMPT *variable*
A list consisting of a format control string and its arguments. This variable determines the prompt displayed on each iteration of TOP-LEVEL-READ-EVAL-PRINT. Its global value is ("~&> "). CLF binds it to ("~&CLF> ") for its top level loops.

TL-PRINT *variable*
To print each result value, TOP-LEVEL-READ-EVAL-PRINT applies the function bound to *tl-print* to the value and *terminal-io*. This function, together with *tl-prompt*, are responsible for transmitting newline characters.

TL-READ *variable*
The binding of this variable determines the reading function for TOP-LEVEL-READ-EVAL-PRINT. Its only parameter is a stream. It is globally set to Lisp's READ function.

TL-EVAL *variable*
The binding of this variable determines the evaling function for TOP-LEVEL-READ-EVAL-PRINT. Its only parameter is a form to evaluate. It is globally set to Lisp's EVAL function.

5. CLF SOFTWARE MODEL

CLF represents software in terms of **MODULEs** whose **Components** are either other modules or **INDIVIDUAL SOFTWARE OBJECTs**. The individual software objects include all the standard Common Lisp categories, such as functions and variables. They also include a set of extended software components which utilize the facilities of the objectbase allowing programs to be written that define and manipulate objects in the objectbase. Much of CLF itself was written with these extended software components. They include class definitions, attribute definitions, interface commands, and views. The combined class of modules and individual software objects is called **SOFTWARE OBJECTS**.

5.1. Attributes of Software Objects

The individual software objects have an attribute **Source-Text** whose value is a string. The string should be a legitimate Common Lisp expression. For example, a function definition would have "(defun...)" as its Source-Text attribute's value. The module hierarchy is defined through the **Component** attribute. Its transitive closure, **Component***, is also defined. **Component-of** is the inverse of Component.

A module can have a **Maintainer**. The value for this attribute should be a PERSON. **Maintainer*** is an "inherited" version of Maintainer -- that is, its value is the value of the Maintainer attribute of the "closest" parent module having a value for the attribute.

The **Module-Directory** of a module expects to have a pathname as a value. In general, only root modules have a value for this attribute. The pathname determines the directory in which CLF saves source, binary, and development files for the module. **Module-Directory*** is the inherited version of Module-Directory.

5.1.1. Reader Attributes

Because CLF stores the individual definitions as text, a programmer must provide the environment with enough additional information to allow Lisp's parser (the function READ) to correctly interpret that text.

Lisp's parser is controlled by three parameters:

- ***read-base*** --- this variable, which must be bound to an integer, controls the mapping from the textual representation of numbers to a Lisp numeric datatype.
- ***package*** --- this variable, which must be bound to a package, controls the mapping from character sequences in text to Lisp symbols.
- ***readtable*** --- this variable, which must be bound to a readtable, provides a finite state "lexical scanner" for the Lisp parser, as well as some characteristics that go beyond the power

of a finite state machine.

Details of the effects of these variables on the Lisp parser may be found in CLtL.[19] The CLF environment manages the proper binding of these variables whenever it needs to invoke the Lisp parser. It does so by associating values for each of them with each individual definition.

Corresponding to the three variables are three CLF attributes -- **Read-Base** , **Read-Package**, and **Read-Syntax**. Values for these attributes may be specified directly for any individual definition, or may be "inherited" from some module of which the individual is a direct or indirect component. If no value is explicit or inherited, a default is used. The attribute names **read-base***, **read-package***, and **read-syntax*** are derived from these three attributes so as to take into account inheritance from parent modules and defaulting.

The value of the Read-Base attribute must be a Lisp integer. The default value for the Read-Base attribute is the value of the variable *default-code-read-base*, which is initialized to 10.

The value for the Read-Package may be a Lisp package, or a package name (a string or symbol).[20]

Two packages are defined in addition to the ones already in the workstation environment.

- **CLF** -- The home package of most of the CLF code extensions documented for CLF users.
- **CLF-USER** -- A package that uses both CLF and Common-Lisp.

The default value for the Read-Package attribute is the value of the variable ***default-code-read-package***, which is initialized to **CLF-USER**.

The value for the Read-Syntax may be a Lisp readtable. This is sufficient for operating within the Lisp environment, but is not sufficient for saving and restoring data outside Lisp's virtual address space. For this reason, we require the value of the attribute to be either a *named* readtable, or the name of a readtable. Common Lisp provides no standard for associating readtables with names. We use CLF's objectbase to store the association, maintaining the name as the value of the readtable's Proper-Name attribute. Four named readtables are predefined:

- **Common-Lisp** --- a copy of the initial common Lisp readtable.
- **CLF** --- Common-Lisp, augmented with the read macro #!.

---

[19] COMMON LISP - The Language, Guy L. Steele Jr., Digital Press.

[20] In ZetaLisp, because package names are not necessarily globally known, names must be interpreted relative to some package. All names used as the value of the Read-Package attribute are interpreted relative to the Common Lisp package. Users can avoid concerning themselves with this detail either by using real packages as the value for the attribute, or by avoiding use of names assigned with the ZetaLisp's Relative-Names option.

The default value for the Read-Syntax attribute is the value of the variable *default-code-read-syntax*, which is initialized to CLF.

It is prohibited for any software object, whether module or individual definition, to have no explicit value for any of the three attributes but be an immediate component of two (or more) distinct modules that *differ* in their explicit (or inherited) values for the attribute. For example, if a function definition *F* is an immediate component of both modules *M1* and *M2*, and the explicit or inherited read-base attributes of *M1* and *M2* are 8 and 10 respectively, then it is required that *F* have an explicit read-base attribute.[21] The situations in which CLF uses these attributes include:

- Installing and compiling software objects.
- Static analysis of software objects.
- Assigning fonts to source text.
- Producing loadable/compileable source text files.
- Zmacs commands, such as C-S-E or C-S-C, that necessitate parsing text from buffers displaying software object definitions.

5.1.2. Module Component Orderings

Although the Component attribute serves only to define a ***set*** of immediate components for a module, some activities in software development require placing a meaningful ***ordering*** (or more usually, partial ordering) on a module's components.

In most programming environments, a single totally enumerated ordering, such as that implicit in the positioning of a sequence of definitions in a source file, is used for multiple purposes. CLF will be moving towards the specification of different partial orderings for different purposes. Currently, however, only two orderings are defined.

- The **Load-Order** attribute of a module determines the order in which that module's components are processed when installing or compiling the module, as well as the order in which the components appear on source text files written by SAVE-MODULE. If a module has no Load-Order specified for it, all its components are treated as equal and the implementation may process them in any order.
- The **View-Order** attribute is used to control the order in which the components appear when editing the module, in listings produced by Hardcopy, and in the display of components in an object viewer. If no View-Order is specified for a module, alphabetical ordering based on the components' Proper-Names is used.

---

[21] There are consistency rules in the environment that attempt to enforce this condition, but until the AP5 based release of CLF, it is possible to violate the restriction in some circumstances without being informed. But this can *only* happen if software objects are components of multiple modules, and only in circumstances where modules (not individuals) below the root are assigned explicit values for the attributes.

The orderings that can currently be specified for these slots are total orderings, with "ties" allowed. The value for an ordering attribute can be in either of two forms:

- a function of two arguments, returning non nil if the first is to be treated as less or equal to the second.
- a list of immediate components of the module. For any two components A and B, if both are in the list, the ordering is induced by their positions. If only one is in the list, that one is treated as strictly less than the other. If neither is in the list, they are treated as equal in the ordering.[22]

An ordering associated with a module is treated as local to that module. It is not "inherited" to submodules in any way. For example, if a time-of-creation comparison were specified as the *View-Order* for module *Utilities*, and *Utilities* contained as a component another module *String-Utilities* with no *View-Order* specified for it, then the components of *String-Utilities* would be sorted alphabetically (the default for *View-Order*) in a hardcopy.

IMPLEMENTATION NOTE: CLF uses LOAD-ORDER as an ordering for both compilation and installation (just as common lisp compiles forms in a file in the same order in which it loads them). The LOAD-ORDER must account for both load and compile dependencies. Typically both are partial orderings whose union is also a partial ordering. For example, the compiler must process macros before it compiles functions that use them, but, when installing those components from a compiled file, the ordering is irrelevant. On the other hand, initialization forms must often be installed in a particular order when one relies on the global state established by another. However, the compiler could usually process them in any order. We have not encountered situations in which the dependencies between compile and installation orders conflict with one another.

The initial setting of *DEFAULT-MODULE-LOAD-ORDER-GETTER* enforces a partial ordering on components induced by the classes mentioned in the list *SOFTWARE-CLASS-LOAD-ORDER-PRECEDENCE*. The initial setting of this list is:

```
(PROCLAMATION STRUCTURE SOFTWARE-CLASS TYPE-DEFINITION
 RELATION-DEFINITION EVENT-DEFINITION GENERALIZED-VARIABLE-DEFINITION
 UPDATE-MACRO-DEFINITION  MACRO-DEFINITION  FUNCTION-DEFINITION
 GLOBALVAR CONSTANT  INITIALIZATION-FORM LISP-FORM  MODULE)
```

A component D1 precedes another component D2 if and only if there is a class C in this list such that D1 is an instance of C but not of any class preceding C in the list, and D2 is neither an instance of C nor of any class preceding C.

FOCAL-COMPONENT *attribute*

[22] There is not currently total enforcement of a restriction on the use of lists to specify component orderings that the list contain *only* components of the module it orders. There is likely to be such a restriction at some future time. If for no other reason, it is thus unwise to build a single list of software objects from multiple modules and use it as the specification of the ordering for all of them.

Focal-Component is an optional attribute of MODULE. It is constrained to be a subrelation of COMPONENT, i.e., the Focal-Component of a module must be one of its immediate components. The only semantics behind this relation (so far) has to do with COMPUTED Load-Orders and View-Orders.

When the effective load ordering for a module is specified with an ordering function (such as the default ordering based on component class described recently), the function will ordinarily be asked to compare objects that are IMMEDIATE components of the module. However, when an immediate component is a module with a Focal-Component, that component is passed to the ordering function, rather than the module. More specifically, when comparing two immediate components with the ordering function, the function is passed to the "ordering surrogate" for each component. If X is not a module, or is a module with no Focal-Component, X is its own ordering surrogate. Otherwise, the ordering surrogate for X is defined to be the ordering surrogate for the focal component of X.[23]

5.2. Software Classes

At the leaves of the component hierarchy in the CLF environment are individual software objects. These objects are subcategorized into a number of distinct classes. These classes correspond primarily to the various "name spaces" of Common Lisp -- functions, variables, types, structures, etc. Most operations in CLF are common to all the classes.

Each software class has a number of attributes that enable CLF to behave in ways specific to that class. These include:

- **Installer**, **Compiler**, and **Uninstaller** {optional}. These should be functions obeying the protocols for these three methods [*** to be documented ***]. The values for these attributes are inherited through the class hierarchy. The root class, Individual-Software-Object, has an Installer attribute that EVALuates the lisp form(s) that result from READing the value of the Source-Text attribute. It has a Compiler attribute that Compiles the form(s). Individual-Software-Object has no value for the Uninstaller attribute.

- **Defining-Function**. A symbol that is defined as a function, macro, or special form. When parsing lisp text into software objects, forms having the defining function of a class as their CAR will yield objects classified in that class.

- **Defining-Template** {optional} should be either a symbol or a string. The defining template is used as a format control string , applied to the proper-name of a software object of the defined class, for generating an initial source-text for the object if it is edited prior to being given any explicit source-text. A symbol S is equivalent to the string "(S ~2~~%0)".[24]

---

[23] The Focal-Component of module M plays no role in ordering the components of M. It is only used in ordering the components of modules CONTAINING M.

[24] The appearance of ~n in these strings is currently the means of specifying FONT information. Think of it as a format directive "switch to font n". In CLF software object buffers, font 2 defaults to a boldface font, Font 0, the default, is CPTFONT.

5.2.1. Common Lisp Software Classes

FUNCTION-DEFINITION *class*

Function-Definitions are software objects that hold the definitions of functions. DEFUN is recognized as a Defining-Function.

MACRO-DEFINITION *class*

Macro-Definitions embody the definitions of macros. DEFMACRO is recognized as a Defining-Function. MACRO-DEFINITION is a subclass of FUNCTION-DEFINITION.

TYPE-DEFINITION *class*

Type-definitions embody the definitions of new types. DEFTYPE is recognized as a Defining-Function.

STRUCTURE-DEFINITION *class*

Structure-Definitions embody the definitions of new structures. DEFSTRUCT is recognized as a Defining-Function. STRUCTURE-DEFINITION is a subclass of TYPE-DEFINITION.

UPDATE-MACRO-DEFINITION *class*

Update-Macro-Definitions embody the definitions of read-modify-write macros. DEFINE-MODIFY-MACRO is recognized as a Defining-Function. UPDATE-MACRO-DEFINITION is a subclass of FUNCTION-DEFINITION.

GENERALIZED-VARIABLE-DEFINITION *class*

Generalized-Variable-Definitions embody the definitions of generalized variables. DEFSETF is recognized as a Defining-Function.

CONSTANT-DEFINITION *class*

Constant-Definitions embody the definitions of named constants. DEFCONSTANT is recognized as a Defining-Function.

GLOBAL-VARIABLE-DEFINITION *class*

Global-Variable-Definitions embody the definitions of global variables. DEFVAR and DEFPARAMETER are recognized as Defining-Functions.

LISP-FORM *class*

Lisp-Forms provide a catch-all for code that needs to be executed in the process of creating a software system, but does not fit readily into any other software class. The Source-Text of a LISP-FORM is expected to be a sequence of one or more evaluatable lisp forms.[25] Typical uses of LISP-FORMs include:

- Initialization of global data structures.
- Tailoring the environment in which the application resides.
- Providing traces or logs of the system creation compilation process.

[25] More precisely, it should be possible to READ successive forms from the text, evaluating each one as it is read.

5.2.2. Defining New Software Classes

SOFTWARE-CLASS-DEFINITION *class*

Software-Class-Definitions embody the definitions of new software classes. DEFSOFTWARE-CLASS is recognized as a Defining-Function.

DEFSOFTWARE-CLASS *macro*

```
(name &key (superclasses '(individual-software-object))
 installer compiler uninstaller
 defining-functions defining-template        )
```

This macro permits the definition of new subclasses of individual-software-object. *SUPERCLASSES*, if provided, should contain at least one type compatible with INDIVIDUAL-SOFTWARE-OBJECT. *SUPERCLASSES* may be a single class name or a list of class names. *DEFINING-FUNCTIONS*, if provided, should be a single symbol, or a list of symbols, to be the value(s) for the Defining-Function attribute of the new class. *DEFINING-TEMPLATE* should be a string or symbol to use as the value for the Defining-Template attribute of the new class.

5.3. Software Interface Commands

Install

software-object *interface command*

Install an interpreted version of the current source text definitions of the object.

Compile

software-object *interface command*

Install a compiled version of the current source text definitions of the object.

Save

module *interface command*

This invokes the function SAVE-MODULE on the module, saving a permanent version of all the module's components, and their attributes, in a file in the module's directory. Clicking left saves source only; clicking right saves source and binary form.

Break

function *interface command*

Modifies the definition of the function so that the debugger is entered when the function is invoked.

Unbreak

function *interface command*

Removes the "break" mechanism from the definition of the function.

5.4. Editing CLF Objects

CLF provides the user with an editing environment similar to that of Epoch for editing the textual representation objects. There are, however, a few important differences. The buffers used are created by CLF and are unknown to the normal Epoch buffer commands and operations. When an object is selected

for editing, normally by clicking on the edit option from the menu of operations obtained by selecting an object with the right button, a buffer is created if needed and added to CLF's buffer queue. If a function-definition with the name **TRY-ME** were selected for editing the Epoch status line would appear as:

```
Zmacs (Lisp)  TRY-ME
```

Here Lisp is the mode, and TRY-ME is the buffer's name. As soon as the buffer is altered an asterisk will also appear in the status line.

The buffer itself will contain

```
TRY-ME
on the top line, with a green background, and be followed by the function's
source text

(defun TRY-ME ()
    ......
    ......
  )
```

Should a collection of objects, such as a module, be selected for editing, a buffer will be created for the whole collection with the collection's name appearing as the buffer's name in the Epoch "mode line" and each of its members appearing separated by the (initially green) label lines. It is not possible to delete members of a collection by deleting the corresponding section of text from the buffer, nor to add a new member by inserting text between existing units. Except for the label lines, which may not be altered, the user may perform editing functions in the usual way using all the standard Epoch facilities as they apply to the text being manipulated.

5.4.1. Buffer Coordination

Coordination is maintained between a software object and all textual views of it so that when the source-text of a software object is changed (by saving a buffer, by loading updates to a system, or direct assertion in the objectbase) all buffers containing the old text are updated. If any of these buffers had been modified the system interacts with the user to determine their disposition. Similarly, if the same object has been differently modified in distinct buffers, if you attempt to save any of the buffers, you will be informed of a possible anomolous situation, and the system will interact with you in an attempt to resolve the anomaly.

Coordination extends to both deleted and added components of a module.

6. Software Evolution Monitor

Develop is the portion of the CLF system which structures the evolution of a system into meaningful units of work called development steps. It records changes to software definitions in those steps, and uses this recorded history to manage the installation of those changes, and distributing the (accepted) updates to users of the changed system. It also offers version and release control, and provides maintenance documentation.

Specifically, *Develop* provides the following capabilities:

1. Automated code installation: the execution environment is updated whenever a meaningful unit of work has been completed.

2. Automatic distribution of software revisions: Accepted revisions are automatically distributed to the user community.

3. An agenda of pending work: Pending development steps can be created which represent future commitments.

4. Ability for non-maintainers to make suggestions about the handling of either bugs or enhancements: The maintainer can choose to accept the suggestion for general users, or permit the originator of the suggestion to continue having her own suggestions installed into her own environment.

5. Semi-automatic production of documentation: The type of documentation that can be produced is a development history, possibly enhanced with analyzed program listings using other tools in CLF[26] .

6. Support for the development of true program alternatives and multiple-version systems[27].

6.1. Model for Software Evolution

A system is a specialization of a module. CLF automatically records the evolutionary changes made to a system by keeping a **development** as a sequence of structured **development steps**. Each development may have a **current step**. While any step is the current step, any changes made to the system are considered to be part of that step[28] . The step which represents the current goal being worked on is the value of the **current-step** attribute of the development of the system.

A development step can have zero or more **modifications**, where each modification can be either another development step (i.e., a sub-step) or a **development modification**. Each development step must have a **step type** and an **explanation**, both entered by the human developer, indicating the

[26] This has not been completed.

[27] This feature is in the design phase.

[28] If some aspect of a system is altered when there is no current step, a new one is created.

purpose of the step. As the developer changes the system to achieve this goal, these changes are automatically recorded as modifications to the current development step. When the developer has achieved the goal of a step, she can **finish**(close) it. All of the changes are then automatically **installed** in her environment. After testing, the developer can make these changes available to other users of the system by **distributing** (accepting) the step.

A development step represents a particular goal to be achieved by the alterations which are part of it. Development steps may have sub-steps, just as goals may have sub-goals. The idea is that when one wishes to make some change to a system, a step is opened, representing the main goal of the alterations. This "root step" is called the **top-level step**. Any step created when a particular step is the current step is automatically made a sub-step of the current step, and itself becomes the new current step of the development. This affords the ability to structure a series of changes in a comprehensible manner.

The leaves of the step sub-step hierarchy are **development modifications**. These are the primitive alterations carried out as part of the implementation of the higher level goal represented by a development step. Each development modification records a single change, addition, or deletion to a system at the level of individual attributes of the objects in the system. For example, if the **source-text** attribute of an individual definition is changed, then *Develop* appropriately represents that change as a development modification, part of whatever the current development step might be.

6.1.1. Atomicity of a Development Step

Develop views each **top-level** development step as an **atomic change to the database**. In other words, while restoring the objects in a system, *Develop* performs the development modifications in each top-level development step **atomically**. This is primarily to ensure consistent replay of the development history and ensures that no program or individual is permitted to view the state of the database after only a few of the modifications in the development step have been carried out. The state of the database may be viewed either before one has started to carry out the modifications, or after all the modifications have been carried out. The development history of a system is viewed as a partially ordered sequence of such atomic changes .

Naturally, when one actually alters a system, one might have a particular development step open indefinitely while completing and testing one's changes, and those changes may, in fact, occur in some chronological order. However, for replay purposes, the changes represented by the modifications in the step are carried out **atomically**. Because of the uncertain duration for which a step could be open and current and because it is possible that one might make a mistake with modifications, *Develop* automatically fixes up a step should it be able to detect that atomic replay of the step will not succeed. For example, if one alters the source-text attribute of the same object many times in the same top-level step, *Develop* recognizes that this would result in contradictory updates when the step is replayed

atomically, and fixes up the step appropriately, representing only a single change to the attribute from its original value to its final value. Similarly, if one adds a component to the system and then deletes it from the system in the same top-level development step, *Develop* fixes up the step so that neither the component addition nor deletion are part of the step .

6.1.2. Ordering of Development Steps

Top-level development steps are ordered by the time that they are **first** finished. Steps which have never been finished are incomparable to one another. Ordering by the **time-of-first-finish** is used to determine in what order steps should be restored. A step which was finished at some time is deemed to occur **before** a step that has never been finished. Given this ordering scheme, and remembering that in *Develop* finished steps can be resumed, we must enforce certain consistency requirements to ensure proper replay:

- A given (single-valued) attribute of an object cannot be modified in two incomparable (i.e., never finished), top-level steps. Since incomparable steps are not ordered amongst themselves, depending on which one of the steps is restored first, one could get different results. Thus, this situation is prohibited. Never finished top-level steps must either modify disjoint sets of objects or different attributes of the same object.

- A component cannot be removed in an earlier step if it has already been modified in a later step. This would render the later step meaningless, and must be prohibited.

- A particular (single-valued) attribute of an object cannot be modified in an earlier step after the same attribute of the object has already been changed in a later step.

These consistency requirements are enforced strictly by *Develop* through the use of consistency rules. Without these rules, it is impossible to guarantee correct replay of the changes made to a system.

6.2. Bringing a System Up-To-Date

The systems that exist in an environment can be updated by using the **update-systems** and **update-individual-system** functions (similar to ZetaLisp's **load-patches**).[29] Precisely what information gets incorporated into one's environment when one updates a system depends on the update mode attributes of a system. The following update mode attributes are used[30] :

- **execution-mode**: What updates to the **execution environment** must be incorporated?
  - Possible values are :never-load, :frozen-steps-only, :all.

- **definitions-mode**: What updates to the **object definitions** must be incorporated?
  - Possible values are :never-load, :frozen-steps-only, :all.

---

[29] These operations may also be invoked through the menu interface.

[30] For maintainers, there are two additional modes, one to indicate if they want suggestion objects restored, and a second to specify whether to restore pending steps.

- **skeleton-mode**: For what kind of steps should the **skeleton of the step** object alone be restored?
  - Possible values are :never-load, :frozen-steps-only.
- **details-mode**: For what kind of steps should the **entire step object** be restored?
  - Possible values are :never-load, :frozen-steps-only, :volatile-steps-only, :all.

Briefly, by a "frozen" step we mean a step that cannot be altered. In *Develop*, this means a step which has already been distributed, suggested, or terminated, since only such steps may never be altered. Volatile steps are steps which have not yet been frozen. A volatile step can be altered by making it the current step, and making more modifications to the objects in the system while that step is current.

Typical users do not care about the details of a development step or the definitions of the objects involved. They merely want the binary versions of the updates installed into their environments. System stubs for such users usually have the following update mode attribute values:

- **execution-mode** == :frozen-steps-only
- **definitions-mode** == :never-load
- **skeleton-mode** == :never-load
- **details-mode** == :never-load

The values indicate that only execution updates for newly distributed steps must be restored. No other aspects of the system are visible to the typical user.

A maintainer of a system, in contrast to a typical user of the system, requires much more information -- including the definitions of all the objects in the system, details of at least the volatile development steps, etc. Thus, update mode attribute values for a system stub in its maintainer's environment might be as follows:

- **execution-mode** == :all
- **definitions-mode** == :all
- **skeleton-mode** == :frozen-steps-only
- **details-mode** == :volatile-steps-only

The values indicate that the execution environment must be updated using all the steps that were distributed or finished at some time[31] . The definitions of the objects must be updated for all steps, whatever their status. *Develop* restores steps at two levels:

- Skeletal view: suppresses the intricate details of a step, providing just the step substep

[31] *Develop* assumes that open or suspended steps are too unstable to be installed. Thus, mnemonic ":all" does not include such steps for installation purposes.

hierarchy, and the modified objects of the step.

- Detailed view: the entire step in all its detail, including all the modifications of the step, and their attributes, representing the changes carried out in the step

Typically, one would like the skeletons of frozen steps to be restored rather than the details. Frozen steps cannot be altered, and it could be wasteful of one's virtual address space to have the details of these steps in the database at all times just in case one needs to review the development history. The details can be restored on demand through the menu interface, by buttoning on the step whose skeleton alone was restored, and selecting the "Load Details" command. There is not much choice but to restore the details of volatile steps because they can still be changed.

The above are the typical values for the update mode attributes of a system in a given environment. If any update mode attribute is not specified, *Develop* behaves as if the attribute had the value :never-load. However, if the update mode attributes of a system are not set to one's satisfaction, they may be changed conveniently using the "Change Update Modes" menu item from the menu for the system.

There are rules in *Develop* which automatically incorporates new information into one's environment (as necessary) when one changes the update mode values of a system. At all times, therefore, a system's update mode values will be an accurate reflection of what aspects of that system have been restored into that environment. For example, if the Definitions-Mode of a system is changed to :all from :never-load, *Develop* will proceed to restore the initial definitions for the system from disk, and then the series of incremental definitional updates to the system from each step.

6.2.1. Ordering the Systems for Update

It is usually required that the systems in an environment be brought to up-to-date in a specific order. The order is determined by some notion of which systems depend on others, i.e., which systems use resources provided by other systems. Systems which provide certain resources must be initially installed and updated before systems which use those resources. While *Develop* has no idea *why* a system might depend on another, it offers a way for users to specify the dependencies between the systems. Thereafter, when *Develop* needs to update the systems, it will find an order consistent with the specified dependencies, and update the systems in that order.

In order to specify system-to-system dependencies, one uses the global interface action "Add Remove Dependency". The **middle** button on this menu item lets one specify a dependency between a pair of systems, one of which must necessarily be before the other. In this manner, one can specify so-called "global" dependencies among systems. These dependencies determine the order in which *Develop* will update systems.

Over a large evolutionary period, however, the above ordering scheme for updating systems usually breaks down. It may be necessary to update systems out of order, at least in specific cases. For example, the MAIL-SERVICE system uses facilities from the system EMPLOYEE-SERVICE. Thus one would specify a dependency to the effect that the EMPLOYEE-SERVICE should be updated before the MAIL-SERVICE. Suppose, that Step 50 of the MAIL-SERVICE introduces a message sending feature that the maintainer of the EMPLOYEE-SERVICE would like to use in Step 68 of that system's development. Clearly, what we need is a way to ensure that the MAIL-SERVICE is up-to-date until at least Step 50, before we install Step 68 of the EMPLOYEE-SERVICE. In this situation, we cannot specify a global dependency.

Develop offers a way to specify requirements or pre-conditions for steps to be incorporated. These requirements are of the form: *In order to install Step N_i of system S_1, first ensure that Step N_j of system S_2 has already been installed*. Such a specification would cause the following behavior: *Develop* first finds an order to update systems. If S_2 preceded S_1 in this order, then *Develop*'s usual default behavior will ensure that Step N_j of S_2 gets installed before Step N_i of S_1. However, if S_1 preceded S_2 in this order, then, just before installing step N_i, *Develop* will note the requirement placed on installing that step, and switch to updating S_2. It would bring S_2 up-to-date up to Step N_j, and then switch back to updating S_1, installing Step N_i, and steps which follow[32] .

Dependencies between specific steps of two systems can be specified using the "Add Remove Dependency" from the CLF global menu. The left button lets the user specify a new dependency between a particular step of one system and a particular step of another. Similarly, the right button permits the user to remove a previously specified dependency between the steps of systems. The instructions to be followed are fairly simple after using either the left or right buttons on this menu option.

In summary, use global or system-to-system dependencies between systems when the usual pattern of dependencies between systems is well-established and understood. This will establish a default order of update for systems. However, if it is required that before installing a specific step of a system, one needs to ensure that a particular step of another system be installed first, and one cannot be sure that the default order will guarantee this, one can actually specify a step-to-step dependency as discussed in this manual. While updating systems, *Develop* will then use the default order, changing it as necessary when the step-to-step dependencies require it to update other systems first.

---

[32] Yes. It is possible that there are cycles in the specified dependencies. While *Develop* does not detect the cycles when the dependencies are specified, it does detect them at update time.

6.3. Development Steps

As noted before, a development step is the basic unit of evolutionary change in *Develop*. In this section, we see more of how development steps are used to structure the evolution of a program.

Initially, there are two possible states that a development step may be in -- **pending** and **open**. A pending step represents a commitment to a future software revision, that when **handled** will become the current step. An open step, on the other hand, is a step that is currently being worked on. Once opened, modifications may be made to the system or its components. These modifications are then recorded as part of the current step of the corresponding development.

When all the modifications have been made the maintainer must **finish** the step which causes the step to be saved to disk (so that its persistence is ensured.) If the step is a top-level step, the revisions in the step are (optionally) automatically installed into the maintainer's LISP environment. Afterwards, the maintainer of the system may **distribute** these changes to the user community, making the step visible to the users at large. It is also possible to **terminate** a finished step. Termination of a step has the same effect as distribution for modified definitions in the step, but there is no executional update associated with a terminated step at all. In effect, it is a step of some historical and definitional interest, and does not affect one's LISP environment.

In addition, steps may also be **aborted**, **suspended**, or **resumed**, as shown in Figure 6-1. Suspending a step causes it to be saved to disk, but not installed. In addition, the parent of the suspended step, if any, becomes the current step. Resuming a step makes it the current step. As a side effect, the step that was current and its ancestors are suspended.

Aborting a step has the effect of **undoing** all its constituent modifications. Under certain circumstances, it may not be possible to abort a finished development step. For instance, one cannot abort a development step if there are later steps that modify a (single-valued) attribute of a particular object which is modified in the same attribute within that step itself. The later steps "depend" in some sense on the earlier steps in our ordering, and one cannot abort steps arbitrarily. Thus, "dependent" steps of a development step must be aborted before a development step can be aborted. Aborting a step may become difficult to do if multiple top-level steps modify the same attribute of the same object. Finally, an aborted step may not be further modified, distributed, or terminated.

Similarly, distribution of a finished step will not be permitted if there are earlier (undistributed) development steps which modify the same objects and attributes. For example, if step 5 was finished before step 6, step 5 occurs before step 6 in the ordering. Say there is a modification to the **source-text** attribute of an object **foo** in both steps 5 and 6. In this situation, the earlier step (5) must be distributed before step 6, since both modify the same attribute (**source-text**) of the object **foo**.

From the above, it must be kept in mind that if a maintainer intends to work with several active top-level steps, she does need a broad plan about what she is going to modify in each step so that she can avoid having a confusing number of top-level steps; otherwise care must be taken to ensure that the top-level steps modify either disjoint sets of objects or different attributes of the common objects. If *Develop* does not let a user do something in a specific step or carry out some operations that the user thinks she should be able to do, it is almost certainly because permitting that operation would make it impossible to have a consistent replay of the development history

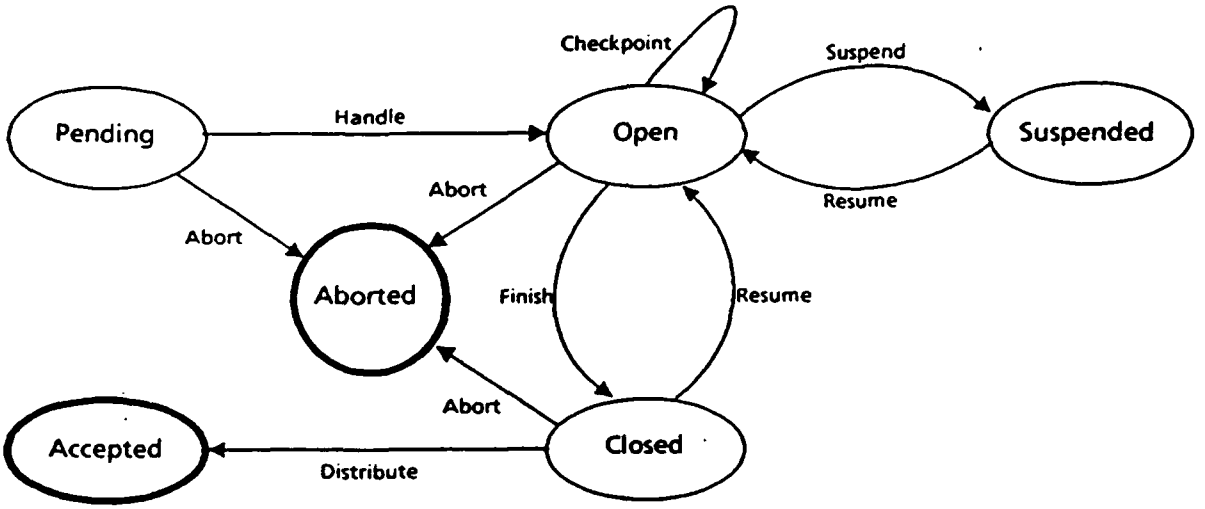


Figure 6-1: A transition network of the states of development steps and the actions that can change them.

6.3.1. Finishing an Open Step

As we have already pointed out, finishing an open (top-level) step is one of the important occurences in the life-time of the step. For one, it determines the ordering of the step relative to other steps. Further, it does indicate that the unit of work represented by that step has come to a meaningful state of completion (at least temporarily). For this reason, *Develop* chooses this time to remind the maintainer of a series of book-keeping details having to do with that step. For example, *Develop* permits one to install the modified definitions into one's LISP environment.

Often, one forgets to include certain changes in the step. For example, a common scenario is that the user makes the textual changes to the objects in the object edit buffers, but forgets to commit the changes to the database. When a step is finished, *Develop* looks for around for modified buffers containing objects in the relevant system to see if the modifications in those buffers are also to be committed in that step, *before* actually finishing the step. This provides a valuable reminding service to the user.

Another activity supported by *Develop* at finish time is the task of supplying a load order for newly added modules, or a new load order for modules to which new components have been added in that step. *Develop* offers the capability to construct a load order interactively before finishing a step. Users often forget to specify necessary dependencies between the objects they add to a system. This invariably results in an unexpected order of installation of the objects, causing anomalous behavior and errors.

Develop also deduces a load order for an installation module[33] at the time the corresponding development step is finished. This load-order can be tailored appropriately by the user. The deduced load order for the installation module is based on the load orders for different modules in the system that the user has already provided. Based on those specifications, *Develop* deduces a load order for installation modules. If the deduced load order is incorrect, it can be changed as needed at the time of step finishing through a series of interactions with *Develop*.

6.3.2. Undoing Selected Modifications

Aborting a development step has the effect of undoing **all** its modifications. The attributes of objects modified are all reverted to values they had when the step was first created. In essence, the state of the system components and their properties are as if the step were never created. Often, one does not want to abort a complete step, but merely wants the effects of selected modifications to be reversed. Perhaps, some changes were not needed or some were simply mistakes which one needs to retract. *Develop* offers one the capability to undo selected modifications. Button on the development step which contains the modifications, and select the "Undo Modifications" menu item. A menu of all the modifications will be displayed. The user selects the changes which she wants undone, and exits the menu by buttoning the mouse in the area labelled "Do It". If she wants to abort the undoing operation, the mouse should be buttoned in the area labelled "Abort".

A pre-requisite for one to be able to undo modifications of a step is that the step be the current step of the development, i.e., it is the step being worked on. Also, in keeping with *Develop*'s consistency requirements, one cannot undo modifications if there are later steps (i.e., steps finished later than the current step) which assumed those modifications. The selected modifications will be removed from the development step, and their effects on the database will be reversed, i.e., the changes which they represent will be undone.

Since the most frequently occuring changing attribute is the *source-text* of an individual-software-object, there is another way to revert the definition of an individual-software-object. When a particular step is current, and the user wants a particular object whose definition was altered in that step to be reverted to its previous definition, button on the object, and select the "Revert Definition" menu item.

---

[33] Installation modules are discussed later in this section of the manual

This will revert the source definition of the object, and remove the corresponding development modification from the step.

6.3.3. Background Step Saving Activities

The persistence of everything that one does using *Develop* is assured by periodically writing out the contents of development steps to permanent storage. Typically, this saving operation is carried out when the status of a step is altered by the user. *Develop* tries to execute all activities related to saving steps or compiling step files in the background. This frees up the user from having to wait until the saving or compiling is finished, and she can resume other activities. Thus, there could be an appreciable lag (typically a few minutes) between the time one changes the status of a step, and the effects of that change are actually recorded on disk.

While *Develop* does not actively prohibit further changes to the step, it has sufficient information during the background activities that it can restore the step and the files corresponding to it to their last consistent state. Each step has its own dumping process and queue. The dumping of multiple processes dumping different steps is synchronized wherever necessary.

In most cases, the process dumping the step is active for just a few minutes, unless the step contains a lot of modifications. If the process runs into an error while dumping, the changes are relatively easy to undo. *Develop* protects the user againt dumper process errors as much as possible, (e.g., the remote host machine to which one is writing is down) by restoring the database to the state before the step status change was made. Thus, one is free to keep working on the system while something has already committed is being saved to disk.

6.3.4. Type of Steps

Each development or pending step must categorize the kind(s) of change to be effected and give an explanation of it. The explanation is simply text that describes the change. The step types used to categorize the change are predefined but can be extended by the user via the **ADD** operation. The primary ones are:

Augment *menu item*
Add a new capability.

BugFix
Fix a bug.

CleanUp *menu item*
Simplify existing functionality.

Fortify

A modification that makes the associated software more robust.

Generalize *menu item*
A change that allows software to accept more cases.

Maintain *menu item*
A modification that is made to retain functionality when other software (i.e., the environment) has changed.

Reorganize *menu item*
Move components between modules or code between components.

Revise

A change that is not upwardly compatible.

Tune

Improve the efficiency of the underlying algorithms or data representations.

6.4. Installation Order for Modifications

We have seen how the modifications in a top-level development step are viewed as occuring atomically from the point of view of **the database**. Thus, the ordering of modifications has no effect on the resultant **definitional state of the system** after the step is replayed. However, this is clearly not true for updates to the **LISP environment**. Order could be vital in creating a compilable form of a step for distribution as a patch to users, and different orders of installing the revisions could give different results. An example of a case where order is important for compilation is when a macro has to appear before any function which calls it. Clearly, therefore, some ordering mechanism has to be set up for the purpose of compiling and installing a step into the LISP environment.

Develop allows one to have an **installation-module** corresponding to each development step, created automatically when the step is created. The installation module is intended to contain all the objects whose new definitions will be distributed as updates to the LISP environment of each user of the system when the corresponding step is distributed. When the maintainer modifies an object, it is automatically inserted as a component of the installation module for the step under which the modification was carried out. Thus, *Develop* fleshes out the installation-module for each step.

The maintainer is free to alter an installation module in any way. Components may be added or removed as needed by the maintainer. If the order of installation is crucial, she can then assert a **load-order** for the module[34] , as she could, indeed, for any other module. This load order is used as the

[34] Note that *Develop* assists a user in finding a load order when a step is finished

order in which the modified objects will get written out and compiled. The installation module for a step is part of the default view for a step. It is the value of the **patch-module** attribute of a development step.

Develop offers another convenient mechanism of inserting components into the installation module of a step. This is the editor command "HYPER-i", similar to the Hyper-z command in the Programming Service. This command can be issued while editing software objects in a system. The meaning of the command is as follows: Unlike the hyper-z editor command which saves new definitions of changed objects in the edit buffer, the hyper-i command **creates copies of the changed objects** in the buffer, making the copies components of the installation module for the current step[35] . The original objects, if they were in the installation module, are removed from it. Also, the definitions of the originals are unchanged. The command provides a convenient way to handle cases where what is installed is not the permanent definition of an object, but some other piece of LISP which has a close relationship to the permanent definition. *Develop* will notify the user if the original changes subsequently IN THE SAME STEP, reminding her to change the copy if needed.

Installation modules permit a very important conceptual separation in *Develop*: updates to the definitions of the objects in the database are handled through atomic updates using the (unordered) modifications in a step, and updates to the LISP environment are handled through an installation module corresponding to the step whose contents are determined jointly by *Develop* and the maintainer, and for which a **load-order** can be asserted to ensure installation in a specified order.

Installation modules are particularly handy in some fairly common boot-strapping situations in software maintenance. For example, say there are some forms which must be evaluated before a particular feature is distributed. The easiest way is to handle this by creating a step, and inserting all the boot-strapping forms as components of the **installation module**. This is a step with no actual modifications to the objects in the system, but will serve the purpose of distributing the boot-strapping forms when it is distributed to other users. Similarly, it is possible to have definitional updates alone in a step by removing all the components of its installation module. Such a step serves as a definitional update with no effect whatsoever on the LISP environment.

6.4.1. Fixing Frozen Steps Which Cause Errors

It often happens that **after** a step is frozen, i.e., it is distributed, one discovers that users report errors when it is installed into their environments. This could happen for a variety of reasons, usually because of an oversight on the maintainer's part while creating the step. *Develop* provides certain facilities to deal with this common situation.

---

[35] It is an error to issue the command when there is no current step.

Once a step is distributed, nothing about its **definitional** aspects can be altered since this would have an adverse impact on the integrity of the database state when the steps are replayed. However, one can alter the execution aspects (i.e., installation module) of a distributed step, removing and adding things to it to fix up the problems when the step is installed. In most cases, the maintainer must create a later step to fix up the definitional environment if needed.

One can "thaw" out a frozen (distributed) step by buttoning on the distributed step and selecting the "Prepare to Redistribute" option. If the details of the step need to be restored, *Develop* will automatically do that, and mark the step as being ready for redistribution[36] . Now, the user can alter the installation module as needed with one important caveat: **the objects in the system itself cannot be altered** unless those modifications are to be recorded as new modifications to the system.

The procedure just discussed assumes that one is in a environment where the distributed step already exists. In case the step is not present (in either detailed or skeletal form) in the environment, one can button on the development object for the system (part of the default view of a system), and select the "Load Specific Step" menu item. Provide the appropriate step number when prompted, and *Develop* will restore the step, setting the stage to use the above procedure. To get the appropriate step number (i.e., to remember which step needs to be re-distributed) one could use the *Develop* facilities to examine the off-line development history of a system, discussed later in this manual.

After making the changes necessary, the user buttons the step , and selects the "Re-Distribute" menu item. *Develop* will then use the altered installation module to create new executional updates which incorporates the changes made to the installation module.

This feature of *Develop* is intended to provide a convenient method to correct the inevitable gaffes that one makes in patching a system. There are limitations:

- People who have already installed the step in its wrong form cannot be helped without a later step which corrects things. Of course, there are cases where once the wrong version of a patch is installed, there is no way to recover, period.
- The definitions of the objects may need to be fixed up depending on how the problem caused while installing the step was resolved.

The points above should emphasize that the redistribution feature is not to be construed as an invitation to be careless about what gets distributed the first time around.

---

[36] If somebody tries to update the system at a time when a distributed step is in the "thawed" state, this step and all others occuring after it are NOT installed

6.5. Non-Maintainer Suggestions

Develop has features to help **non-maintainers** communicate their recommendations to the maintainer in a reasonably clear manner. Especially in environments where the users of a system are also expert programmers, users do not just report problems and wish lists to the maintainer, they may also wish to communicate **how** the problem they experienced can be solved or an enhancement ought to be implemented. *Develop*'s suggestion facility is tailored to suit these needs.

The fundamental idea is that if a potential suggestor can get hold of the definitions in a system that she is using, she can recommend ways to change those definitions or adding new definitions by creating a "suggested" step with those changes. The maintainer reviews the recommended changes, and if they look useful to the user community at large, a regular development step containing the changes is distributed, making the suggestions visible to every one. This permits every one to share the development burden somewhat by letting them think of ways to implement the things on their minds. The maintainer is left with the task of synchronizing possibly conflicting suggestions with the ongoing agenda of development for the system.

There are basically three aspects to supporting suggestions. First, how suggestions are made by a non-maintainer; second, giving the suggestor the capability to have his or her own suggestions installed during updates to the system if the maintainer has not yet acted on them or has acted but not accepted the changes; and, finally, how the maintainer can act on a suggestion, examining and either incorporating or rejecting the suggestion.

6.5.1. Making a Suggestion

A non-maintainer can make suggestions to the maintainer. The suggestion could involve new definitions for objects which are already in the system or the addition of new objects to the system. For example, if one discovers a bug in a function, and knows how to change its definition to fix the problem, one would create a suggestion proposing a new definition for the function. Later, this suggestion would be examined and perhaps incorporated into the system by the maintainer.

The manner in which suggestions are created is not substantially different from the way in which steps are created. A non-maintainer merely opens a step, puts her suggested modifications into that step, finishes the step, and, finally, "distributes" it. The step is not actually distributed, it merely gets written out as a suggested step which the maintainer will eventually examine. No users other than the suggestor and the maintainers of the system will see the suggested update until the maintainer incorporates it.

A key aspect of making suggestions is to get the "current" definitions of the objects to modify. The current definition for non-maintainers is the definition as of the last frozen step. In order to get the definitions to modify, a non-maintainer has two choices. First, the definitions can be restored via the

"Load System" global CLF interface action. This is likely to be much more than is needed for the suggestion because it would restore **all** the definitions in the system, whereas the contemplated suggestion may involve just one or two functions. A second possibility is to use the "Hyper-." editor command to edit the definitions of specific objects. The Hyper-. command brings in the last distributed definition of an object from disk by reading a master file containing these definitions.

The universe of alterations which a suggestor can make is restricted. For example, a suggestor cannot alter the release or maintainer attributes of a system. She can only change definitions or add components. In fact, the components added by a suggestor to a system are not represented as true components of the system. Rather, they are represented as **suggested-components** of the system. *Develop*, nevertheless, monitors alterations to these objects in its usual manner. It is important that any components added to a system have the proper package and syntax designations. The suggestor should ensure this for the components that she might explicitly add to the system.

After a suggestion has been made, if the suggestor wishes to retract it as a suggestion, and treat it like an aborted step, she can button on the suggestion, and select the "Cancel" option[37] . *Develop* simply updates the status of the suggestion to be aborted in the patch directory. During subsequent updates by a maintainer or by the suggestor, the step will be treated like any other aborted step--it will be ignored.

6.5.2. Updating a System with Suggestions

In general, a suggestor's patches do **not** affect the patches installed for the typical user, who still gets only the patches distributed by a maintainer. However, *Develop* will install the suggested patches when the author of the suggestions does an update to a system.

It is conceivable that a person is neither a maintainer nor the author of a suggestion to a system, but wishes to install it while updating the system . *Develop* offers a mechanism to do this. One can select the "Accept Others' Suggestions" menu item by buttoning on the appropriate system for which one want to receive the suggested patches. *Develop* will prompt for the person whose suggestions are to be received. During subsequent updates, any applicable suggestions authored by that person will be included in the set of patches to be considered for installing. One can receive suggestions authored by more than one person by repeating the procedure above. If one should choose to remove one of these persons from the list of people whose suggested patches one wants to receive, the "Stop Accepting Suggestions" menu item when one buttons on the appropriate system is used.

Whether a user is the suggestor (in which case *Develop* automatically assumes the user wants her

[37] There is no difference between the effects of this operation and an "Abort" operation issued while the suggested step was still volatile.

suggestions to be considered for installing if they have not been handled by a maintainer) or a general user who has opted to receive suggestions authored by one or more persons. *Develop* will permit the installing of suggested patches under certain circumstances.

It is not easy to characterize just when it is safe to install the suggestions because the suggestions contain arbitrary LISP code. *Develop* supports the suggestor in figuring out if it is safe to install a suggestion. When a suggestor does an update to the system, *Develop* collects all the relevant suggestions, and displays them in a menu along with diagnostic information. The diagnostic information includes the following:

- What was the maintainer's action on the suggestion? There are four possibilities:
  - :pending-action -- indicating the suggestion has not been acted upon.
  - :totally-subsumed -- the suggestion has been acted upon and is entirely incorporated into a later step.
  - :partially-subsumed -- the suggestion was partly incorporated into a later step.
  - :rejected -- the suggestion was not received well, and was not deemed appropriate for the public at large.
- An explanation by the maintainer of her action in the cases where an action has been taken. This enhances the above information.
- Are there later steps (i.e., steps to be installed after the suggestion) which modify the same objects? *Develop* categorizes each suggestion as being:
  - :safe -- no later patches at all.
  - :probably-safe -- there are later patches, but they modify other objects than the ones modified in the suggestion.
  - :probably-unsafe -- there are later patches which modify some of the objects modified in the suggestion.

Based on the above diagnostic information printed for each suggestion, the suggestor selects which suggestions are to be installed through the menu. The selected suggestions are installed, and the others are ignored.

6.5.3. Maintainer's Handling of a Suggestion

At the outset, the maintainer can choose not to deal with suggestions at all by simply setting the **load-suggest-info?** attribute of the system in question to nil[38] . Suggestions can be accepted by setting the same attribute to t. This can be done by using the menu interface, buttoning on a system, and selecting the option to either "Load Suggested Steps" or "Don't Load Suggestions" when the **load-suggest-info?** is set to nil and t respectively.

When a system's update mode attributes indicate that the steps are to be restored, and the system's **load-suggest-info?** attribute is set to t, suggestions are restored when a system update is done

[38] *Develop* treats a system with no value for the **load-suggest-info?** attribute as if it had a value of nil

Restoring a suggestion has the effect of restoring the step **without affecting in any way the objects in the system**. The suggestions and their constituent modifications are merely stored as part of the development history.

The maintainer can examine a suggestion in a "read-only" manner. This can be done by selecting the "Examine" option from the menu one gets by buttoning on a suggestion. Examining has the effect of displaying the suggested modifications in a read-only buffer, edits the installation module for the suggestion, and displays the current definitions of any modified objects if possible. This gives the user a chance to assess the value of the suggested change. *Develop* also prints information about which changes are possible to incorporate exactly as they were suggested (meaning, the suggestor modified what is the current definition for the object in the maintainer's environment) and which changes may require a merge between the current definition in the maintainer's environment and the suggested definition.

A suggestion can be rejected by choosing the "Reject" menu item after buttoning on the suggestion. This has the effect of making the step invisible to general users (except the suggestor himself!) and the maintainer for subsequent updates.

The suggestion can be incorporated by selecting the "Incorporate" menu item after buttoning on the suggestion. *Develop* now puts up a menu of modifications from the suggestion. The maintainer selects those modifications she feels should be made a permanent part of the system. If new components are proposed in the suggestion, and the maintainer likes them, the modifications corresponding to their addition must be selected as well. *Develop* opens a new step, and the selected modifications are viewed as having occurred in the new step. If a step is already open, one has the option of either using that step itself to incorporate the suggestion or to suspend that step and create a new step.

Incorporation happens automatically except for **suggested-components**. Recall that these are not true components of the system, but, rather, are components suggested by the suggestor. At incorporation time, *Develop* prompts the maintainer to decide which module the suggested component should become a part of. An object is not permitted to be both a **suggested-component** of a system and an actual **component** of the system. If there are objects in the installation module of the suggestion which are not part of the system, *Develop* queries the user about whether to add each to the installation module for the step into which the suggestion is being incorporated. After this, as part of the incorporation, *Develop* prompts the maintainer to describe her action regarding the suggestion. This information is used in helping suggestors to figure out if their suggestions are still safe to install.

After the selected modifications are put into the currently open step, and the database reflects the appropriate definitions, the maintainer merely finishes and distributes the step as she would any other. This makes the suggested changes available to all users of the system.

6.6. Querying the History

Develop maintains a detailed history of how the objects in a system change over time. The intention is that such a history will provide the maintainer important information about when and how an object was changed, relieving the maintainer of much of the tedious book-keeping burden which she might otherwise assume. However, merely having the information is not sufficient. There must be some way for users to access the information as well.

Develop can present several paraphrases about the development steps in one's environment at any time. For example, when one buttons on a system, the "Overview" and "Active Steps" menu items provide a summary of all the development steps **in the user's virtual address space**, and all the volatile steps **in the user's virtual address space** respectively. However, these kinds of reports presuppose that the development step objects are already in the environment. What should one use to find out about steps that do not exist in that environment?

Develop permits one to view summaries of the off-line history it maintains. The facility can be accessed by simply buttoning on a system and selecting the "Off-line History" menu item. "Off-line", in this context, means the history is not in the user's virtual address space, but is out on persistent storage. This facility comes in handy if one wants to find out about the development of the system without actually restoring the steps into one's environment.

The facility permits one to get an overview of all the steps in the system, or just the volatile steps, just the suggestions, just the pending steps, or steps which modify a specific object etc. A rich variety of information is made available through this facility. The limitations are that there is no general query mechanism to interface to the information on persistent storage. *Develop* provides some "canned" queries to access the off-line information. However, if one needs to navigate the information in some arbitrary manner, there is no way to do this unless the relevant steps, in all their gory detail, are restored into the user's virtual address space. The user is then free to use AP5 queries to navigate the history as desired.

6.7. Making New Release of a System

The steps in the development history of a system apply to a specific release of that system. Whenever one has to start a new major generation of the system, for example, when one needs to make a major change in the surrounding kernel software, one has to make a new release of the system. By making a new release of a system, one is essentially starting a **new** development history for the system, a new sequence of development steps which will apply only to the new release of the system. Obviously, creating new releases is not a routine or commonly occuring maintainer activity. Nevertheless, it does happen, and *Develop* supports creation of new releases to make the transition to the new release smooth.

Only a maintainer can make a new release of a system, and even she cannot do so if there are volatile

steps in the development of the system at that time. Volatile steps indicate an initiated, but unfinished, agenda. Therefore, creation of a new release of a system is not permitted when the system has volatile steps. One can create a new release by buttoning on a system, and selecting the "New Release" menu item.

Creating a new release of a system has the effect of saving the system completely -- all its definitions and a complete recompile. *Develop* increases the *release* attribute of the system by 1, and then sets up a new patch directory to hold its steps. *Develop* permits the transfer of certain kinds of steps interactively from the old history the new history:

- suggestions by non-maintainers which have not yet been handled.
- pending steps which have not yet been handled.

Thus, the ongoing agenda of things to do can be transferred to the new release at the maintainer's discretion.

An important decision needs to be made by the maintainer. Is it safe for users of the old release of the system to continue to receive upgrades from steps made to the new release of the system? There is no way for *Develop* to determine this automatically. It merely finds out from the user. If the two releases are incompatible, as is often the case, the development steps corresponding to the old release should be maintained disjointly from the steps for the new release. If the maintainer wants to permit users of the old release to continue receiving upgrades, she can specify that she needs a **bridge step** from the old release to the new release. A bridge step automatically increments the release value of a system when it is installed, thereby enabling users of the old release to receive the upgrades to the new release on subsequent updates.

6.8. Operations

CLF enables the user to edit, install, compile, add, or delete software in CLF. *Develop* records those modifications, encourages the user to structure them into meaningful units of work, and uses this recorded history to manage the installation of changes, for revision and release control, for distributing updates, and for providing maintenance documentation. Most operations can be chosen by selecting items from a menu. This menu is generally obtained by selecting a CLF object with the right mouse button which pops up an appropriate menu. The operations are:

Abort

development step *interface command*

Abort a development step. A dump file is written. This step, however, can never be resumed, finished, or distributed. This interface command is only offered when a step is pending, open, or finished.

Accept Others' Suggestions

system *interface command*

Prompts for a person who will be added to the list of people whose suggestions to that system will be installed into one's environment at the next update.

Active Steps system *interface command*

Displays the open steps for a development (i.e., those that have been created but not yet finished), or the previously opened step if none are currently open.

Add Attribute software object *interface command*

Adds an attribute to an object. For software objects, this is the standard way to add **maintainer**, **documentation**, **local-declaration**, **module-directory**, **read-syntax**, **read-base**, and **read-package** attributes.

Add New Component(s)* module *interface command*

Creates a new software object and adds it to the module. Clicking right on this item will select the editor as the current process and ask you to enter the defining form(s) for the new component(s) using the editor. The appropriate type of component is deduced by the defining form. Clicking left, on the other hand, would prompt you for the type of object and its definition. During prompting for the object type, typing COMPLETE will pop up a menu of valid types, one of which must be selected. Finally, clicking middle alows an existing component to be added to a module. attribute) command when applied to modules.

Add/Remove Dependency *global interface action*

Command to specify new dependencies between a particular system and step and another system and step. Can also be used to remove such dependencies, and to add global dependencies.

Add Type/Explanation development step *interface command*

Add a step type and/or explanation to a development step.

Bring Up To Date individual program object *interface command*

Installs the most recent definition of the object (and correspondingly updates its source-text definition). Only offered on objects whose source definition is not current (because a previous version of it had been installed by the **(Re)Install** command).

Cancel suggestion *interface command*

Abort a suggested step after it has already been suggested to the maintainer. If no-one has already handled the suggestion, this will let a user retract her suggestion.

Change Update Modes system *interface command*

Puts up a menu of the update mode attributes and their possible values. Current values are marked. Newly specified values are used to reset the update mode attributes.

Checkpoint development step *interface command*

Dump a development step but leave it open.

Coerce To System module *interface command*

Coerce a module to a system. If a **module-directory** has not already been given for the module then the user will be queried for one. One should type in something that can be coerced into a valid pathname specification. A development is then created with an initial step that records the initial definitions of the components of the system.

Create Step system *interface command*

Create a new development step as a sub-goal of the currently active step. If no step is current, Develop will pop up a menu containing **open** and **pending**: one of these must be selected as the initial status. Otherwise, the status of the step is :open. Then, another menu will pop up which asks if you want to "set the type and explanation": clicking on a mouse button is an affirmative reply, moving the mouse away means that you will provide a type and explanation at a later time. If you decide to add a type and explanation, a menu of valid step types will be displayed. You must choose one, and must input an explanation for the change.

Create/Load System*
global interface command

Clicking left causes a new system with no components to be created. one must specify the **module-directory** attribute, which specifies the directory where the software will reside. A development is also created. Clicking right loads an existing system.

Definition development modification *interface command*

Display the recorded object definition for a modified software object and/or the incremental change that was made to it. Module modifications are always incremental changes, whereas modifications to individual software objects are generally newly recorded object definitions.

Describe Copies individual-software-object *interface command*

Shows information about the different copies of a given individual software object in different installation modules.

Distribute
development step *interface command*

Distribute a patch to the user community. This causes the modifications in a step to be compiled, which will then be installed into a users environment when she updates the system which corresponds to the step. When distributing a step the compiled text corresponds to that which has been installed in the maintainers environment. The maintainer is warned if a modification made within the step has been superceded by another from a step not yet distributed.

Dont Load Suggestions system *interface command*

Do not restore suggested steps for this system at future updates. Issued by the maintainer when she does not want to review suggestions.

Edit
program object *interface command*

Modify an object by editing its textual representation. The textual representation of an CLF object will be placed in an CLF object buffer. After editing has been completed (as indicated by **hyper-z** or **hyper-x control-s**) the modified textual representation will be reparsed to define the structured definition of the object. For individual software objects, the new definition will be saved and a development modification will be created corresponding to this change. If a development step is not open then *Develop* will automatically prompt the user to create one.

Edit Patch Module
generalized-step *interface command*

Edits the installation module of the buttoned development step or suggestion. The order of editing can be selected by using different the left or right click.

Examine
suggestion *interface command*

Examine a suggestion -- edits the current definitions of the modified objects, puts the

suggested definitions of the objects in a buffer, and edits the installation module if it contains objects not already in the system.

Finish
development step *interface command*
Only the author of a step can finish it. The step and its modifications will be dumped to ensure its persistence. If it is a top level step, all modifications made within it that have not yet been installed will be installed.

Finish Active Steps* system *interface command*
Finish the open step(s) of the system and dump them. Only the current step will be finished if the left button was depressed. Otherwise, all open steps will be finished. If a top-level step is finished, all of its modifications which have not yet been installed are automatically installed.

Handle
development step *interface command*
Handle a pending development step now. The status of the step is changed from pending to open, and the step becomes the current step.

Incorporate
suggestion *interface command*
Incorporate a suggestion. If a step is already open, it may be used. Otherwise, a new step is created. A menu of suggested modifications is put up. Selected modifications are inserted into the step in which the suggestion is incorporated.

Install/Compile development step *interface command*
Install any uninstalled modifications of the development step into the environment.

Restore Details development step *interface command*
Restore the most recent details about the development step, restoring it completely. Used often when one has only the skeleton of the step restored.

Restore Specific Step
development *interface command*
Prompts for specific step number, and restores the step, with all its details. The objects in the object base are not affected.

Restore Suggested Steps system *interface command*
Puts up a prompt to determine whether to restore the suggested steps to the system, either when the next update to the system is done or right away in the background. Carries out the selected option.

Modifications* applies to several types *interface command*
Print the modifications made to a individual program object in reverse chronological order if the object clicked on was either a software object or development modification. However, if it was a development step then print the modifications made within it. If the middle button is used then also print the modifications to embedded steps.

New Release system *interface command*
Creates a new release of the system, a fresh dump of the system, and a new patch directory. The system must be definitionally up-to-date, and there must be NO volatile steps in the development for one to be able to use this command.

Off-Line History system *interface command*

Provides facilities to view the history of the development of the system using canned queries.

Overview*

system or development step *interface command*

Print an overview of the changes to a system or development step. In the case of a system, the user has the option of seeing all development steps, all pending steps, or all suspended steps by clicking left, middle, or right on the menu item, respectively. In the case of a development step only an overview of that step is provided.

Prepare to Redistribute

development step *interface command*

Prepares a distributed step for redistribution by restoring the details of the step if necessary, and marking it appropriately.

Query History

individual-software-object *interface command*

Summary of changes to a specified object from the development steps in virtual address space or out on persistent storage.

Re-Distribute

development step *interface command*

Re-distributes the designated step by creating a new lisp file for its installation module and re-compiling it. Will be offered only if the step is prepared for redistribution (See above.)

(Re)Install

development modification *interface command*

Install a development modification. This updates both the definition of the modified object in the database and also, possibly, the LISP environment.

Reject

suggestion *interface command*

Reject a suggestion, issued by a maintainer. The suggestion will become invisible to everybody except the suggestor herself.

Resume

development step *interface command*

A suspended step may be resumed, which causes it to become the currently active step. Any open steps which are not ancestors of the resumed step are themselves suspended.

Revert Definition

individual-software-object *interface command*

While a step is current, this is used to revert the source definition of an object modified in that step to its previous definition, and remove the corresponding modification from the step.

Stop Accepting Suggestions

system *interface command*

Prompts for a person who should be deleted from the list of people from whom suggestions are being accepted. Suggestions from the person will not be installed at the next update to the system

Suspend

development step *interface command*
Suspending a step causes the step to get dumped and its parent (if one exists) to be selected as the current step.

Terminate
development step *interface command*
Terminate a finished step. For the maintainer, it like a distributed (i.e., frozen step), but no non-maintainers will ever see it.

Undo Modification
development step *interface command*
When a step is current, this command lets one selectively undo modifications in that step.

Update
system *interface command*
Update the system as directed by the update mode attributes.

Update Systems *global interface command*
Causes all systems to be updated according to the specified update mode attributes. These attributes are usually pre-set to either a user profile or a maintainer profile. Depending on need, one can alter them directly.

View Dependencies
system *interface command*
View the dependencies specified between the steps of the buttoned system and the steps of other systems.

6.8.1. Additional Editor Commands

HYPER-i *editor command*
Create copies of the changed objects in the buffer, making the copies components of the installation module of the current step of the development. The original objects are unchanged, and will themselves be removed from the installation module if they are already components of the installation module. Error to do hyper-i without a current step in the development.

HYPER-. *editor command*
edit the definition of an object. The last frozen definition of the named object is obtained from a master file. The object is created in the user's environment as a suggested-component of the system of which it is a part.

6.9. Examples

To create a step without specifying a type or explanation.

```
user input:
Selected menu item: Create Step (FOO)
Current step:
  0-Step 3 of FOO
FOO
```

To create a new system.

```
user input:
Selected menu item: Create System
Input the proper-name of the system: foo
Specify the default system directory: local:>
No step active.  No previous step.
Development FOO created for system FOO.

)
```

Appendix I
Lisp Universal Kode Elaborator

I.1. Overview

LUKE (Lisp Universal Kode Elaborator) is a code walking "shell". Code walkers are used for a variety of program analysis tools. Luke is a shell in the sense that it performs no useful task in its own right: it must be tailored to a particular application by providing certain functional parameters, and, perhaps, designating specific control for selected macros, special forms, or functions.[39] LUKE can be used either to compute a transformation (some "image") of a piece of code, or solely for side effect -- e.g., to gather statistics about the code.

LUKE imposes a limited sort of "grammar" on Common Lisp that suffices to factor code into roughly a dozen subcategories. The code walk is driven by a walker function for each category. The default walker for each category may be overridden. The actual walker to use for each category is passed through the code walk in keyword parameters.

LUKE is highly extensible. The collection of "grammatical" categories can be extended by adding a new keyword and a default walker for the category. A LUKE code-walk is guided by an "application" name. Each application may specify dispatches to special purpose code (designated walkers) on forms having designated symbols in their CAR position. Designated walkers may also be designated as application-independent "defaults". This is primarily used to tell LUKE how to treat macro uses without expanding them. It is essential for extending LUKE to cover embedded languages. A designated walker may *override* the default behavior or simply *augment* it.

LUKE provides a template language to ease the specification of designated walkers. Templates are *compiled* into designated walkers, not interpreted during the code walk.

---

[39] It is not possible to designate specific control for uses of lexically scoped macros or functions.

Appendix II
Source Code Importer

To support the use of CLF in maintaining software developed in other environments, a facility is provided for importing source code files. The facility works by making a single pass over the file to produce a module containing a single software object for each top level lisp form in the file.

Each software object is given a Source-Text attribute. The value of this attribute is the string representation of the text in the file from the beginnning to the end of the form on the file. In particular, case distinctions, read macros, and comments are preserved as they appear on the file.

Top-level comments on the file, whether in ";" or "#|" form, are turned into strings and made the value of the Documentation attribute. Since it is not possible to be sure just what software object a comment would be best attached to, the following heuristics are used. Any comments appearing before the first software object form on the file, or after the last one, are attached to the module. All others are attached to the first form following the comment.

Two top level forms are given special treatment. When an IN-PACKAGE form is encountered, no software object is created.[40] Instead, *package* is changed so that following forms will be read in this package. If no components have yet been created for the module, this package is made the Read-Package of the module. As each component is created, if the current package differs from the module's Read-Package, then the component is given an explicit Read-Package attribute. Otherwise it is allowed to inherit from the parent.

A top level EVAL-WHEN form does not lead to the creation of a new component. Instead, each form within the eval-when is treated as a top-level form of the file. The components created for these forms are given explicit values for their Eval-When attribute corresponding to the times list of the eval-when form.[41]

MAKE-LISP-FILE-INTO-MODULE [Pathname &key given-module module-name]

> This is the function for importing a file. Pathname should be either a pathname, string, or file stream.[42] If the :given-module keyword is provided, its value should be an existing module. In this case, the module created is made a component of that module, and the initial reading

---

[40] Therfore, when an IN-PACKAGE form is the first form in a file, comments immediately following it are regarded as pertaining to the module, not to the next form

[41] Nested eval-when clauses result in the components being given Eval-When attribute values for the intersection of the times lists scoped over them.

[42] Any character stream would do, but the Importer relies on the "random access" capabilities provided by the FILE-POSITION function.

environment is determined by the given module. Otherwise the reading environment from the calling environment is used, values for the Read-Package, Read-Base, and Read-Syntax attributes are assigned to reflect that environment, and a Maintainer attribute is assigned to be the logged-in user. The Proper-Name of the new module is that specified by the :module-name keyword, if provided. Otherwise, the name component of the file is used.

IMPORT FILE *system operation*

This system operation conducts a dialog to obtain a pathname and module name from the user. It then invokes MAKE-LISP-FILE-INTO-MODULE.

Appendix III
CORONER -- An AP5 Debugging Facility

The "CORONER", provides a means to debug failed AP5 atomic transactions. There are numerous reasons why an ATOMIC transaction may fail to complete successfully in AP5. Two of the most common are UNFIXED-CONSISTENCY-VIOLATION (finite termination of consistency cycles with a rule still violated) and CONTRADICTORY-UPDATES (attempting to both ++ and -- the same tuple in the atomic transition, whether from the originating program or a consistency rule repair clause).

For these two classes of failure, if the abort is not caught by the originating progam, a PROCEEDABLE error is signalled. If you choose the corresponding proceed option, you will be given a chance to inspect and modify information about the transaction in the editor. When you indicate you are done editing (hyper-Z), you will be offered the opportunity to retry the (possibly modified) transaction.

While you are editing the transaction, you are OUTSIDE of any atomic, so you may modify the database (including its rules).

The text you edit looks like the following:

```
;; introductory explanation

(atomic
  Tuples asserted by the originating program
)

#|
 "analysis" of each consistency cycle
|#
```

In the tuples you see in the text, objects that have a readable print representation are printed in their readable form. Other objects are bound to generated variables (which have names like the defaultname of the object) and the variables are printed. If you answer YES to the "retry?" query after concluding your editing, the program you have created is executed in place of the atomic transaction from the originating program.

Appendix IV
User Interface Resources

The X11 window protocol provides a concept of "resources" which allows an end-user to state general or application specific preferences and a means for an applications (called a "client" in X11 jargon) to view and, if it chooses, honor those preferences. The preferences may be used for tailoring color and font choice, among other things.

IV.1. Color Resources

There are a number of different logical "colors" that CLF uses to highlight text in epoch. The binding to server colors is up to the user, and is now controlled by entries in the server resource database.

The resource entries take the form epochserver.*logical color name* : *server color name* Table IV-0 enumerates the logical colors used for displaying task status in the task status buffer.

Logical Color	Default	Interpretation
outputavailable	green	The background used in the task status buffer for displaying running tasks with available output.
inputblock	red	The background used in the task status buffer for displaying tasks blocked for user input.
terminated	black	The background used in the task status buffer for displaying terminated tasks with available output. Also used for the notice placed at the end of a terminated task's buffer.

Table IV-1: CLF Task Status Colors

Table IV-1 enumerates the remaining logical colors and their uses.

If neither foreground nor background is explicitly specified for hypertext, the Epoch defaults will apply, and thus the text will not be highlighted at all, although it will still respond to mouse clicks. If only a background is specified, the foreground will use textcolor. A background must be specified if a

Logical Color	Default	Interpretation
textcolor	black	The foreground color for highlighted text of all kinds. This is the color of the letters.
uptodatecolor	green	The background color of an object label (in buffers editing collections of objects) when the buffer text reflects the current database state.
outofdatecolor	red	The background color of an object label when the buffer text does not reflect the current database state.
remotecolor	yellow	The background color of an object label (in buffers editing collections of objects) when the buffer text reflects the current database state, but the object is being modified on some other machine on the network.
promptcolor	red	The background color used for prompts for user input.
hypertextbackground	*	The background color for hypertext objects printed to text streams.
hypertextforeground	*	The foreground color for hypertext objects printed to text streams.

Table IV-2: CLF Logical Colors

foreground is also to be specified.

If no value is specified for terminated, white text on a black background is used. If a value is specified, the Textcolor on the specified background is used.

IV.2. Font Resources

CLF's interface windows (other than the text editor) are generated by a tool kit called FormsKit. FormsKit applications are free to specify any fonts they choose for various forms. However, FormsKit provides X-server-dependent bindings for a number of "logical" fonts, so application writers can simplify font selection if they choose. CLF's interface forms, such as browsers and dynamic views, use these fonts.

The logical fonts comprise three *sizes* with four *faces* each, as depicted in table IV-2. The resource entries take the form Forms-kit. logical font name : server font name

size\face	Plain	Italic	Bold	Bold-Italic
small	smallFont	smallItalicFont	smallBoldFont	smallBoldItalicFont
medium	font	italicFont	boldFont	boldItalicFont
big	bigFont	bigItalicFont	bigBoldFont	bigBoldItalicFont

Table IV-3: Logical Font Resource Names

FormsKit applications expect that all fonts of the same logical size will be bound to server fonts of the same height.[43] There is also one other "logical" font, specified with the resource name "workstationStandardFontname". This determines the font that forms use by default if no font is specified at all, either in the form definition, by the program which creates the form instance, or in the X11 server's resource database.

[43] A second property that enhances many applications is for character widths to correspond across all faces of a given size. This does not mean that the fonts need to be fixed-width, but only that, e.g., an "a" in the font selected for italicFont is the same width as an "a" in the font selected for boldFont.

6x

Appendix V
Site Configuration

Although CLF is primarily a virtual memory application, it does require a file system for specific purposes, most notably, to make software objects and evolution history persistent. Because each software system designates a directory for its persistent data, no special site configuration is necessary. The user simply must be running an OS configuration that makes the strings used as module-directory attribute values resolve to legal and accessible directories.

At a given site, CLF maintains a master directory of all the software systems it manages. Each site should dedicate a file system directory for this purpose. The variable pgm::*SITE-MODULE-REGISTRY* should be globally set to a value (list of strings), suitable as a directory value for CL's make-pathname function, that will designate the chosen directory.

To interface with a sites hardcopy capabilities, CLF requires a directory where it can create temporary text files. The variable ap5::*HARDCOPY-DIRECTORY* should be be globally set to a value (list of strings), suitable as a directory value for CL's make-pathname function, that will designate the chosen directory. The initial setting of ap5::*HARDCOPY-DIRECTORY* is ("tmp"), which is suitable for most Unix platforms.

On some lisp platforms, CLF must create temporary files in order to compile source code. The variable pgm::*TEMP-FILES-REPOSITORY* should be be globally set to a value (list of strings), suitable as a directory value for CL's make-pathname function, that will designate the chosen directory. The initial setting of pgm::*TEMP-FILES-REPOSITORY* is ("tmp"), which is suitable for most Unix platforms.

Index