Appendix A

Selected Systems Supporting History Tools

Six systems were selected for assessment: ALOE, INTERLISP-D, MINIT, MPW, ROOMS, and SEED. In order to provide a fair assessment of the state of current support for all uses of history, only general-purpose development environments were considered rather than special-purpose systems (e.g., text editors, information retrieval systems). Such environments support computing activities by providing access to different applications (e.g., text editing, graphical drawing, electronic communication, programming, and managing file systems).

ALOE is an interactive structure-editor-based generator for programming environments [Linxi & Habermann, 1986]. ALOE, and the environments generated by ALOE, support a history tool which records program development history. The history tool provides facilities that allow users to reuse command lines and recover from errors. Command lines may be reused individually or as a group, facilitated by a history macro facility. User errors may be corrected using an editor or using an UNDO/REDO facility.

INTERLISP-D is a display-oriented programming environment supporting the LISP programming language and a set of user facilities [Teitelman & Masinter, 1981; Teitelman, 1977]. DWIM and PROGRAMMER'S ASSISTANT are two notable user facilities that support history tools. In certain spelling error situations (e.g., function name), DWIM's spelling corrector scans related items the user recently worked with (e.g., functions) to make corrections. PROGRAMMER'S ASSISTANT maintains a history list containing user operations, a description of their side effects, and their results. As well, several commands for manipulating a history list are provided: REDO one or more operations, UNDO effects of specified operations, FIX operations before re-executing them, and USE a substitution before re-executing the operation.

MINIT allows users to issue a single textual command line combining command submission and window management operations [Barnes & Bovey, 1986]. It exists on a multi-window, single-user, graphical workstation running the UNIX operating system. As part of its support for command submission, MINIT maintains a history menu containing entered commands. The history tool allows users to reuse previous commands with possible modifications.

MPW – MACINTOSH PROGRAMMER'S WORKBENCH – is the MACINTOSH development environment [Farr, 1989]. The MPW environment provides an array of application

creation tools along with a command language called MPW SHELL . MPW SHELL combines the features of a command shell and a multi-window text editor with mouse-based command capabilities. It supports operations typically found in most UNIX command interpreters (e.g., file management and input and output redirection). Users may use the MPW ''Worksheet'' to execute and store often-used commands; the current activity is noted at the bottom of the ''Worksheet''. Furthermore, users may re-execute commands in any window. For example, users may re-execute commands embedded in source code windows, in program comments, in program output, or in the ''Worksheet''. Commands are executed at the location where they are invoked and outputs appear in the area following each command. If commands are re-invoked, previous outputs are replaced with current outputs. This manner of command execution makes it difficult to reconstruct the order in which commands are executed because there is no semblance of a linear history.

ROOMS is an enhancement of INTERLISP-D that supports multiple virtual workspaces. Each room represents a virtual workspace consisting of all programs used to perform a single user task and no two rooms are displayed simultaneously. In addition to the history capabilities available in INTERLISP-D , ROOMS provides history capabilities for navigating around multiple workspaces. Users can determine where they are by examining the display. BACK DOOR allows users to go back to the previous activity space and find out where they just came from.

SEED is a session editor implemented on top of the VAX/VMS environment [Holsti, 1989]. In addition to the traditional means of command execution (one command at a time), SEED supports incremental script creation and script processing. A transcript of all executed and unexecuted commands for a session, including user mistakes, recovery actions, and command outputs, may be saved and restored from session to session. A transcript's contents may be copied and manipulated before it is incrementally re-executed. Modifications to executed and unexecuted commands are permitted. However, executed commands are automatically undone if they are modified (i.e., the system backtracks to the state preceding the modified command before it is re-executed). Therefore, the transcript provides a linear record of the history.

Appendix B
Locality Detection Algorithm

As a program executes, it makes references to portions of a computer's memory in units known as *segments*. The sequence of memory references is a *memory reference string*. The set of segments referenced during a program's execution is the *memory reference set*.

Locality is the phenomenon in which a program's memory references are limited to a small subset of segments for an extended time interval. The intuitive notion of a *locality* has two components: a favoured subset of segments (i.e., a *locality set*) and a definite reference interval (i.e., a *phase*) over which this favoured subset remains unchanged.

A least recently used (LRU) stack may be used to construct sets of segments which possess many of the characteristics associated with a locality. The LRU stack can be represented as an ordered vector $L(t) = (L_1(t), L_2(t), \ldots, L_n(t))$ where n is the size of the memory reference set and $L_i(t)$ is the segment identifier for the ith most-recently-referenced segment at time t. If the stack is cut at any position i, then the topmost i segments in the stack $S_i(t) = \{L_1(t), L_2(t), \ldots, L_i(t)\}$ are the i most recently referenced segments at time t. Since the LRU stack can be cut at any point, it defines a hierarchy of localities (i.e., $i = 1, 2, 3, \ldots, n$).

However, the problem with using the LRU stack is that, at each point in time, the number of locality sets is equal to the number of segments in the memory reference set (i.e., n). The reason is that the LRU stack does not contain enough information to allow the selection of certain locality sets as being more ''distinctive'' than others. In order to do this, we need information about the formation time (i.e., $F_i(t)$) and termination time (i.e., $T_i(t)$) of a set of segments $S_i(t)$. Also, we need to maintain the most recent time unit $R_i(t)$ where a reference was made to the ith stack position (i.e., last time where a reference was made to the least recently referenced member of the set of segments).

The formation time of a new set of segments of size i is the termination time of the previous set of segments of size i. Thus, a set of segments of size i at time t, $S_i(t)$, remains intact until a reference is made to a stack position greater than i. It is considered distinctive (i.e., a locality set) if and only if every member of the set has been re-referenced since the set was formed. Specifically, a locality set at time t, $LS_i(t)$, is any $S_i(t)$ for which $R_i(t) > F_i(t)$.

In order to detect such locality sets, an extended LRU stack is needed [Madison & Batson, 1976]. It consists of the LRU stack and two ordered vectors:

$$F(t) = (F_1(t), F_2(t), \ldots, F_n(t))$$
$$R(t) = (R_1(t), R_2(t), \ldots, R_n(t))$$

There are two parts to the locality detection algorithm. In the first part, an active locality set $LS_i(f)$ is terminated at time $t = e$ when $f \neq F_i(e)$. We must check for termination because for a given size i, the termination of a locality set is brought on by the activation of a new set of segments for the same size i. In the second part, a new locality set is formed at time $t \neq f$ when $R_i(f) > F_i(f)$ (i.e., at time $R_i(f)$ when the least recently referenced segment is re-referenced after the locality set is formed at time $F_i(f)$).

The following is an outline of the locality detection algorithm developed from Madison and Batson (1976).

Assumption: Each reference constitutes one time unit

Given:

- a two dimensional array *LRU* for the LRU stack
  (i.e., LRU[i,t] holds the ith stack element at time t)
- a two dimensional array *F* for the formation time
  (i.e., F[i,t] is the formation time of a $S_i(t)$ at time t)
- a two dimensional array *R*
  (i.e., R[i,t] is the most recent reference to *i*th reference at time t)
- variable *n* is the length of the current reference set
- variable *reference* is the current reference
- an array *Active* numbered from 1 to n, holds a boolean indicating whether S_i is an active locality set
- an array *FormTime* , numbered from 1 to n, holds the formation time of LS_i
- a two dimensional array *Sets* holds the references in the current active locality set
  (i.e., Sets[size, i] holds the ith member of locality set of size
- *whereInLRU(argument)* is a function returning position of the *argument* in LRU at time t-1

```
/* process each reference and see if it is part of a locality set */
t = 1
while (true ) {
        reference = readReference()
        if (reference = null )
                exit()
        lastOccur = whereInLRU(reference)
        if (lastOccur <= n) {
                /* the reference appeared previously in LRU[lastOccur, t-1] */
                LRU[1, t] = LRU[lastOccur, t-1]
                R[1, t] = t
```

```
            F[lastOccur,  t] = F[lastOccur,  t-1]

            for (j=1 to lastOccur-1)  {
                  LRU[j+1,  t] = LRU[j,  t-1]
                  R[j+1,  t] = R[j,  t-1]
                  F[j,  t] = t
            }
            for (k=lastOccur+1  to n) {
                  LRU[k,  t] = LRU[k,  t-1]
                  R[k,  t] = R[k,  t-1]
                  F[k,  t] = F[k,  t-1]
            }
      }
      else  {
            /* the reference  is a new reference  */
            LRU[1,  t] = reference
            R[1,  t] = t
            F[1,  t] = t
            if (t > 1)
                  for (j=1 to n) {
                        LRU[j+1,  t] = LRU[j,  t-1]
                        R[j+1,  t] = R[j,  t-1]
                        F[j+1,  t] = t
                  }
            n = n + 1
            Active[n]  = 0
      }

      if (t > 1)
            for (i=1 to n) {
                  /* Part  1: Look for terminated  locality  sets  */
                  if (Active[i]  = 1 && FormTime[i]  ≠ F[i,t])  {
                        /* LS i(t −1) = {LRU  [1, t −1], ... , LRU  [i, t −1]}
                        /* print  LS i(t −1), formation  and termination  times  */
                        print  FormTime[i],  t-1
                        for(m  = 1 to i)
                              print  Sets[i,  m]
                        Active[i]  = 0
                  }
                  /* Part  2: Look for new locality  sets  */
                  if (R[i,t]  > F[i,t]  && Active[i]  = 0) {
                        Active[i]  = 1
                        for (j=1; j<=i; j++)
                              Sets[i,  j] = LRU[j,  t]
                        FormTime[i]  = F[i,  t]
                  }
```

```
                }
        t = t + 1
}

for (i=1 to n)
        if (Active[i] = 1) {
                /* LS_i(t-1) = {LRU[1,t-1],...,LRU[i,t-1]} */
                /* print LS_i(t-1), formation and termination times */
                print form[s], t-1
                for(m = 1 to i)
                        print Sets[i, m]
        }
```