MASSACHUSETTS INSTITUTE OF TECHNOLOGY
ARTIFICIAL INTELLIGENCE LABORATORY

A.I. Memo No. 638 June, 1981

Sniffer: a System that Understands Bugs

by

Daniel G. Shapiro

Abstract:

This paper presents a bug understanding system, called sniffer, which applies inspection methods to generate a deep understanding of a narrow class of errors. Sniffer is an interactive debugging aide. It can locate and identify error-containing implementations of typical programming cliches, and it can describe them using the terminology employed by expert programmers.

The debugging knowledge in Sniffer is organized as a collection of independent experts which understand specific errors. Each expert functions by applying a feature recognition process to the test program (the program under analysis), and to the events which took place during the execution of that code. No deductive machinery is involved. This recognition is supported by two systems; the cliche finder which identifies small portions of algorithms from a plan for the code, and the time rover which provides access to all program states which occurred during the test program's execution.

In a typical scenario, the user interacts with Sniffer to identify a manageable subset of the test program which seems to contain an error. He then issues a complaint describing the expected behavior of that region of the code. The sniffer system then selects and applies the relevant bug experts, and produces a detailed report about any error which is discovered. This report includes a high level summary of the error, an analysis of the intended function of the code in terms of its component parts, and a description of how the particular data values and control paths involved during execution led to the manifestation of the error observed.

This paper was originally submitted as a master's thesis to the MIT Department of Electrical Engineering and Computer Science, on May 8, 1981.

This report describes research done at the Artificial Intelligence Laboratory of the Massachusetts Institute of Technology. Support for the laboratory's artificial intelligence research has been provided in part by the Advanced Research Projects Agency of the Department of Defense under Office of Naval Research contracts N00014-75-C-0643 and N00014-80-C-0505, and in part by National Science Foundation grant MCS-7912179.

Acknowledgements

I would like to thank my advisor, Richard C. Waters, (sometimes called Dick), for his precise, inexhaustable, and kind support throughout the creation of this master's thesis. He is an excellent advisor, and is appreciated well. I would also like to thank my cohort (one Roger Duffey by name) for his moral and technical assistance in this research. Keep plugging, kid. On a slightly larger scale, I would like to thank (Dr.) Barbara White and (Dr.) David McDonald mostly for being around, and Mr. Daniel P. Dolata for his endless comradery, dispersed though it was across 3000 miles. Lastly, I thank Jane Rebecca Katz for living through three years of the altered humanhood emmanating from this particular gourd. She gets to celebrate too.

CONTENTS

FIGURES

1. Introduction

This thesis presents a system, called Sniffer, which deeply understands some errors in code. Starting from a bug description supplied by the user, the system can trace an error to its source, recognize the purpose for the code involved, and describe the problem at a level of detail appropriate to an expert programmer. Sniffer identifies errors in programs regardless of their domain of application, and it employs mechanisms which are language independent in form.

The design of Sniffer was motivated by the observation that debugging is currently an arcane science which provides very little guidance for the task of identifying errors. The process of recognizing bugs requires knowledge from a variety of sources, and typically involves a number of different strategies for localizing errors. A partial list of these sources includes the program, its intended purpose, the execution paths and data states involved in its execution (either inferred or observed), a knowledge of the primitives of the programming language and of the language interpretation process, and the mappings between the symptoms of bugs and their probable causes.

In the face of this diversity, Sniffer employs a generalized production rule format to represent its knowledge about bugs. Each expert (or production) in the system contains all of the information relevant for locating and identifying a specific error. This approach defines an initial theory of bug recognition. It considers errors to be positive entities around which knowledge can be organized, as opposed to representing them as differences from an established norm. This mechanism makes it possible for individual bug experts to contain extensive knowledge about particular errors. At the same time, the production rule format constitutes a default theory of bug recognition; it is a simple mechanism for localizing information which does not restrict the problem solving methods that can be employed. It is also a modular organization in that new bug experts can be introduced with comparative ease.

The expert system methodology is particularly effective in the domain of debugging because it cleanly coordinates the process of obtaining information from a number of independent sources of knowledge. In a more elaborate theory, uniform methods (such as deduction) should be involved, but perhaps as tools, as opposed to the guiding principles of the solution. At the current level of sophistication, Sniffer shows that an expert system is a natural organization for the task of understanding errors.

Sniffer is also a demonstration of the power of inspection methods in program recognition and

analysis. The system generates its understanding of errors by recognizing the pattern of events associated with particular bugs. It identifies algorithms by matching them against programming cliches, and it determines the circumstances surrounding errors by directly examining a history of the execution of the code. This research shows that inspection techniques are a conceptually simple alternative to the creation of deductive engines for discovering facts about code.

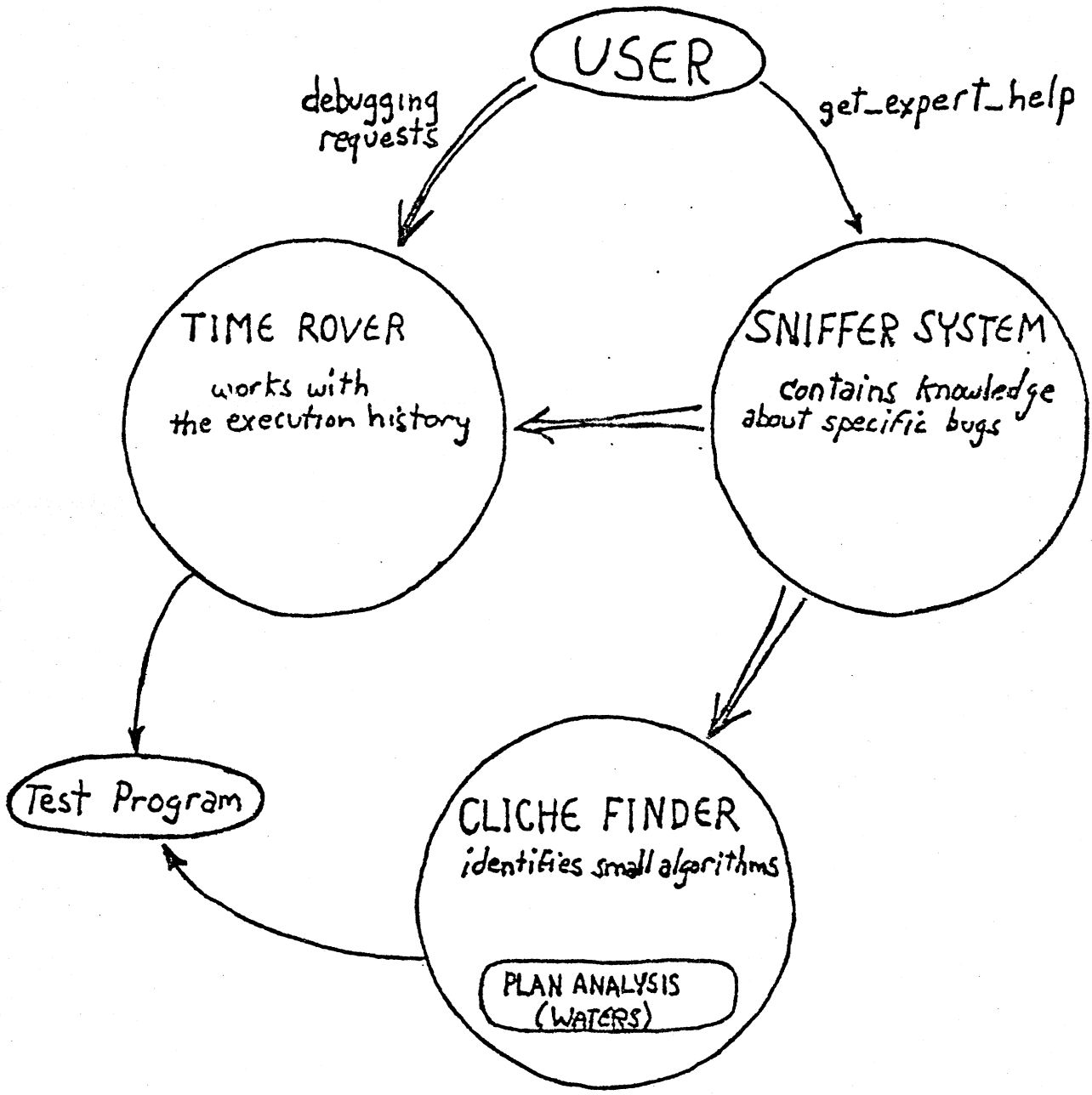
Sniffer is implemented in three major components; the sniffer system which contains all the information relevant for recognizing specific bugs, the time rover which supports queries about a program's history, and the cliche finder, which identifies fragments of algorithms in programs that are used later as a basis for recognizing errors. (See figure 1).

The debugging knowledge in sniffer is organized as a collection of independent experts for specific bugs. Each expert (or sniffer) can examine the user supplied complaint, the suspect piece of code, and the execution history of the program to determine if the bug it knows about is present. The sniffers do not contain background knowledge about the particular program being examined. Their expertise lies in the domain of programming, and concerns typical problems in the use or implementation of programming cliches. In the current version of Sniffer, each expert identifies a narrowly defined error. The generality of the sniffers come from their ability to recognize implementations of typical algorithms independently of the way in which they are coded. This ability is derived from the cliche finder, which in turn is supported by a system, written by Waters [Waters 1978] that transforms programs into a regular and language independent representation called a PLAN (see also [Rich and Shrobe 1976]). The expressive power of PLANs are central to this thesis.

The cliche finder is constructed as a collection of procedures which recognize algorithms as patterns in the PLAN language representation for programs. The object of the system is to raise the level of discourse about a program. Rather than talk about `car` and `cdr` operations, the cliche finder makes it possible to speak about aggregates the size of list enumerations or splice-in operations. The cliche finder operates on the primitive structures of the PLAN language, which include an explicit representation for the data and control flow within a program, and a taxonomy for the building blocks of recursive and iterative routines.

The time rover monitors the execution of the test program (the program undergoing analysis) and provides access to the information it records. It remembers both control information, and the succession of values acquired by all data objects in the code. At every instance of a side-effect operation, the system deposits a record which preserves that information. On every function call and

Fig. 1. The Design of Sniffer



function return it deposits an analogous record as well. The result is a complete picture of the program's state as it evolves through time. The information in this trace is sufficient to rewind the program to an earlier point, or to run it backwards if that is desired. In addition, the time rover can evaluate an expression as if it occurred at an arbitrary moment during the test program's execution. Both the user, and the bug experts make use of this facility.

A general scenario for use of Sniffer is as follows: the user is sitting at a terminal, watching a program run. At some point, he becomes aware that the output is incorrect, although the program is still functioning. He stops the execution and investigates the problem using the facilities of the time rover. He might examine the order of function calls on the stack, the values of several parameters, or events and data in procedures which were invoked *and which successfully returned* some time ago. Eventually, the user finds a particular execution of a region of code which seems to contain a problem. He then makes a complaint to the sniffer system, of the logical form

`(get-expert-help` *expected-result time-t code-region*`)`

The sniffer system analyzes the code for *expected-result* and for *code-region* to obtain a quick understanding of the type of the error. It then invokes all the relevant bug sniffers.

A sniffer might look at a the flow of control through a specific execution of a nested conditional, or compare the values in a list before and after a function was called, or ask the user for further information. If the bug the sniffer knows about is present, it produces a detailed error report. This report includes a high level summary of the error, an analysis of the intended function of the code in terms of its component parts, and a description of how the particular data values and control paths involved during execution led to the manifestation of the error observed.

Sniffer was implemented in Lisp on the MIT Lisp Machine. The Lisp Machine was chosen because it has the high speed and large memory capacity required by Sniffer. The programs submitted to the system were also written in Lisp. This decision simplified the implementation considerations, although it restricted the set of programs which could be analyzed. However, the focus of the research remains in language independent techniques.

2. A scenario using Sniffer

This chapter contains a scenario produced by using Sniffer. However, in order to create a scenario which shows bug detection, one needs a test program that is spiked with errors. This program has to be complex enough to illustrate subtle errors, but also simple enough to avoid becoming a distraction from the main part of the research.

2.1 The test program

The test program is a morphogenesis simulation, called *prosper*, which loosely models the growth of a colony of bacteria. In *prosper*, the user provides an initial pattern of cells and a collection of production rules which govern their division. The simulation outputs a trace of the bacteria colony through time.

The cells live on a rectilinear array called the grid. Each cell occupies one square of the grid and may have up to four neighbors, corresponding to the top, right, bottom and left positions of the array. Every cell has three basic properties, a type, an age, and a division time (which is the next time at which it is expected to divide). The productions cause cell division. They are local transformations that apply to one cell in the context of its immediate neighbors. Productions can access any of the properties of the adjacent cells. For example, a typical transformation (see figure 2) might map a cell of type "c" surrounded by "a" cells into two "c" units. In order to make the necessary room, the neighbors are pushed out of the way.

Prosper is implemented as a production rule system that operates on data kept in a priority queue. This queue, called the events-queue, orders the cells according to their division time. The cell with the next (or lowest) division-time has the highest priority. (See figure 3 for the top level code.) The flow of control is as follows: the grid is initialized with some pattern of cells, and those cells are assigned division times and placed on the events-queue. The central loop removes the first member of the queue, and finds the set of productions which can affect cells of that type. One of these candidates is selected and applied. The transforms are responsible for requeueing any second-generation cells which they produce. *Prosper* terminates when the events-queue is empty.

The grid is implemented as a hash table keyed on the location of cells. (This allows incidental connectivity to be discovered, when separate formations grow together.) The transformations are stored in a library, also in the form of a hash table keyed on the type of the cell affected. The

Fig. 2. Some sample transformations

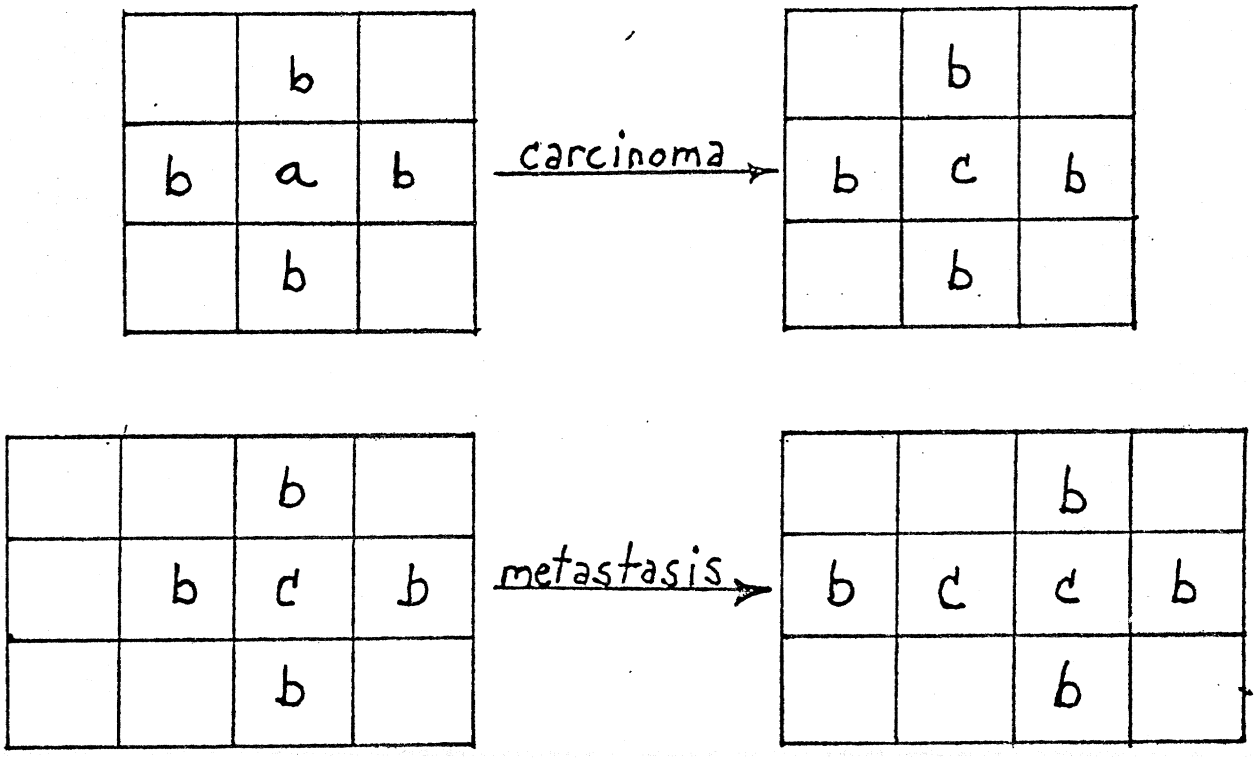


Fig. 3. The code for *prosper*

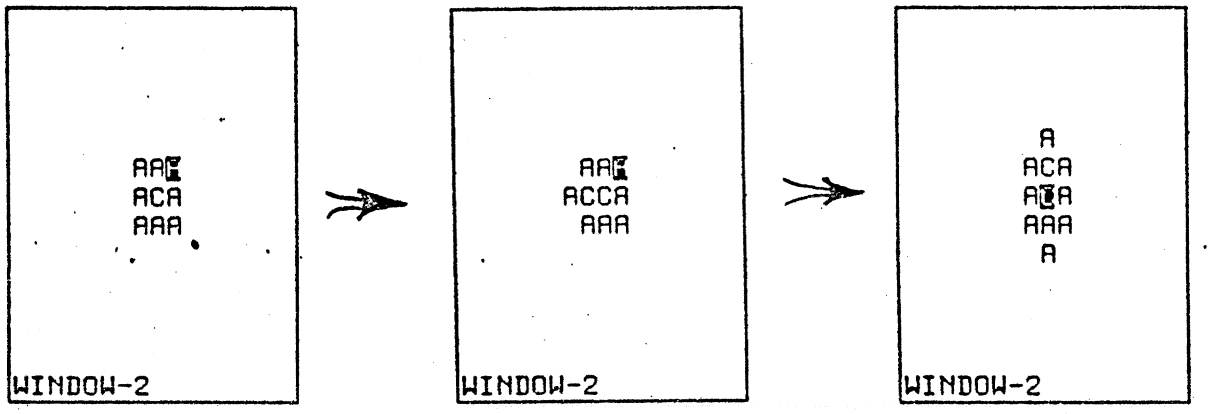
```
(DEFUN PROSPER (EVENTS-QUEUE)
  ((LAMBDA (TRANSFORM-LIB GRID)
     (PROG (MATCHES CELL DIV-TIME)
           (GRID-INIT EVENTS-QUEUE GRID)
      LP   (COND ((NULL EVENTS-QUEUE) (RETURN NIL)))
           (DISPLAY-GRID GRID)
           (SETQ CELL (TOP-CELL EVENTS-QUEUE))
           (SETQ DIV-TIME (TOP-TIME EVENTS-QUEUE))
           (SETQ EVENTS-QUEUE (REST EVENTS-QUEUE))
           (SETQ MATCHES (FIND-TRANSFORMS CELL TRANSFORM-LIB))
           (APPLY-TRANSFORMS MATCHES CELL GRID)
           (GO LP)))
   (CREATE-TRANSFORM-LIB) (CREATE-GRID)))
```

events-queue is implemented as a sorted list, with division-time used as the index.

2.2 The scenario

The following scenario was produced with Sniffer. The dialogue starts after the program, *prosper*, has been running for some time, and has started to generate incorrect output at the terminal. The problem is that the user expected a collection of productions to cause an explosive growth of cancer cells (cells of type "c"), and nothing happened. (The productions are shown in figure 2. Figure 4 shows the output of prosper.)

Fig. 4. The output of prosper



The user's input is in lower case, and is preceded by a "<" prompt. System output is in upper case. I have interspersed comments describing the user's thoughts throughout the scenario.

The user notices that the program is outputting bad data, and interrupts it to find the bug.

```
;Breakpoint BREAK; Resume to continue, Abort to quit.
(examine-history)

focus-time = ~26402, [CDR TRANSFORM]*
```

This indicates that the program was interrupted at time ~26402, which was at the end of the execution of the form (CDR TRANSFORM). *Focus-time* is a system maintained global variable.

The user moves the focus of attention to the most recent point in time at which prosper was being executed.

```
< (move-to (past-when '(in prosper)))
focus-time = ~26373, GRID*
```

This request locates a moment immediately inside of prosper, as opposed to a time within a function that prosper calls.

```
< (print-frame)

Execution time: ~26373, GRID*
Function: PROSPER
Executing at:
(NAMED-LAMBDA PROSPER (EVENTS-QUEUE)
  ((LAMBDA (TRANSFORM-LIB GRID)
     (PROG (MATCHES CELL DIV-TIME)
           (GRID-INIT EVENTS-QUEUE GRID)
      LP   (COND ((NULL EVENTS-QUEUE) (RETURN NIL)))
           (DISPLAY-GRID GRID)
           (SETQ CELL (TOP-CELL EVENTS-QUEUE))
           (SETQ DIV-TIME (TOP-TIME EVENTS-QUEUE))
           (SETQ EVENTS-QUEUE (REST EVENTS-QUEUE))
           (SETQ MATCHES (FIND-TRANSFORMS CELL TRANSFORM-LIB))
           (APPLY-TRANSFORMS MATCHES CELL GRID*)
           (GO LP)))
   (CREATE-TRANSFORM-LIB) (CREATE-GRID)))
```

The function print-frame displays the context of the current execution time. *Focus_time* is at top level during the execution of prosper, at the end of the evaluation of the atom, `GRID`. After this moment, the flow of control enters apply-transforms, and eventually leads to the interrupted execution of `(CDR TRANSFORMS)`.

Since the problem is that cancer cells are not dividing, the user checks to see if any are scheduled for processing. He prints out the contents of the events-queue.

```
< (@ focus-time 'events-queue)
((24 A (-2 0) 2) (24 A (1 0) 2) (24 A (1 1) 2) (24 A (1 -1) 2) ...)
```

The function, `@`, causes a Lisp form to be evaluated in the context of the time supplied as its first argument. The events-queue is a represented as an association list of division-times and cells. The `car` of each item is the division time, and the `cdr` represents a cell.

The user prints out just the types of the cells which are in the queue.

```
< (@ focus-time '(mapcar 'cadr events-queue))
(A A A A ...)
```

The cells near the top of the events-queue should be cancer cells and they are not. However, the cell which is currently being processed has already been removed from the queue. The user examines its value.

```
< (@ focus-time 'cell)
(A (0 -1) 2)
```

The user then finds the most recent time when a cancer cell was being processed. Its division should have instigated explosive growth.

```
< (move-to (past-when '(just-became-true
                        '(@ ? '(eq (cell-type cell) 'c)))))
focus-time = ~00720, [TOP-CELL EVENTS-QUEUE]*
```

This expression returns the moment when the variable, CELL, became a cancer cell. The request is implemented by scanning the execution history for the moment when the predicate, (just-became-true ...) applies. The variable "?" accesses the scan-time.

```
< (print-frame)

Execution time:  ~00720, [TOP-CELL EVENTS-QUEUE]*
Function: PROSPER
Executing at:
(NAMED-LAMBDA PROSPER (EVENTS-QUEUE)
  ((LAMBDA (TRANSFORM-LIB GRID)
     (PROG (MATCHES CELL DIV-TIME)
           (GRID-INIT EVENTS-QUEUE GRID)
      LP   (COND ((NULL EVENTS-QUEUE) (RETURN NIL)))
           (DISPLAY-GRID GRID)
           (SETQ CELL [TOP-CELL EVENTS-QUEUE]*)
           (SETQ DIV-TIME (TOP-TIME EVENTS-QUEUE))
           (SETQ EVENTS-QUEUE (REST EVENTS-QUEUE))
           (SETQ MATCHES (FIND-TRANSFORMS CELL TRANSFORM-LIB))
           (APPLY-TRANSFORMS MATCHES CELL GRID)
           (GO LP)))
   (CREATE-TRANSFORM-LIB) (CREATE-GRID)))
```

Execution is at the end of (TOP-CELL EVENTS-QUEUE), just before the setq function returned.

```
< (@ focus-time 'cell)
(C (0 0) 1)
```

This cell should have metastasized, and yet it did not. The next expression looks forward to a

time when the transformations which could apply to CELL have been selected, and evaluates MATCHES in that environment.

```
< (@ (future-when '(eq (current-function ?) 'apply-transforms))
     'matches)
((OLD-AGED-CELL DIE) (CANCER-CELL-WITH-ONE-NEIGHBOR METASTASIZE))
```

MATCHES is a list of two transformations. Each transformation has two parts, a predicate which determines whether the production can apply, and a function which implements the transformation itself. The first candidate in MATCHES removes old-aged cells from the grid, the second transformation causes explosive growth. The user determines which one was selected.

```
< (@ focus-time '(old-aged-cell cell grid))
NIL
```

This expression reevaluates the predicate for the "die" transformation in the current time-environment. The result is necessarily identical to the one returned by the original invocation of that form in the test program. Since it is NIL, the metastasize function must have been selected instead. The user moves forward in time to a moment when top level code in "metastasize" is being evaluated.

```
< (move-to (future-when '(in metastasize)))
focus-time = ~01751,
           *[NAMED-LAMBDA METASTASIZE (RIGHT-CELL KEY-CELL) ...]
< (print-frame)

Execution time: ~01751,
           *[NAMED-LAMBDA METASTASIZE (RIGHT-CELL KEY-CELL) ...]
Function: METASTASIZE
Executing at:
*[NAMED-LAMBDA METASTASIZE (RIGHT-CELL KEY-CELL)
    ((LAMBDA (NEW-CELL LOCATION)
       (INCREMENT-DIVISION-COUNT KEY-CELL)
       (MAKE-ROOM-BETWEEN KEY-CELL RIGHT-CELL GRID)
       (GRID-INSERT NEW-CELL LOCATION GRID)
       (EVENTS-QUEUE-INSERT NEW-CELL (+ DIV-TIME 2) EVENTS-QUEUE)
       (EVENTS-QUEUE-INSERT KEY-CELL (+ DIV-TIME 2) EVENTS-QUEUE))
     (CREATE-CANCER-CELL) (CELL-LOCATION RIGHT-CELL))]
```

The calls on events-queue-insert should have placed the cancer cells, new-cell and key-cell, on the events-queue with a high priority division time. The user checks to see if the events-queue was modified at any time during the execution of that procedure.

```
< (move-to (future-when '(eq (current-function ?)
                             'events-queue-insert)))
focus-time = ~02672,
      *[EVENTS-QUEUE-INSERT NEW-CELL (+ DIV-TIME 2) EVENTS-QUEUE]

< (unmodified* (@ focus-time 'events-queue)
               (@ (end focus-time) 'events-queue))
T
```

In an environment where different versions of an object can be compared across time, several new types of equality become important. Unmodified* is the strongest test possible. (See the section on equality and coreference for a detailed discussion.) The expression `(end focus-time)` returns the time corresponding to the end of the evaluation of the current function.

The results of the test confirms the user's suspicions. The insert function was called, but the data never entered the events-queue. This is a suitable point to ask the sniffer system for its opinion.

```
< (get-expert-help '(events-queue-member new-cell events-queue)
                   focus-time
                   (end focus-time))
```

The get-expert-help function invokes the sniffers. The first argument is a Lisp predicate that is expected to apply (to be non-nil) after the execution of the region of code specified by the last two arguments has occurred. In this case, that region happens to enclose a single s-expression (the call on events-queue-insert). The sniffers use the predicate as a partial specification for the code in the region. They examine the code for the predicate, and the code inside the region, as well as the control flow and data values involved during those sections of execution. The sniffer which identified the bug produced the following report.

Bug Summary

The bug is a case of violated expectations. The function METASTASIZE called EVENTS-QUEUE-INSERT with the apparent intent of inserting NEW-CELL into the EVENTS-QUEUE by side-effect. The insertion did not occur because EVENTS-QUEUE-INSERT is an insertion function for sorted lists without header cells. It does not act by side-effect when the item sorts to the beginning of the queue. It conses it onto the top of the list instead.

Analysis

The function

```
(DEFUN EVENTS-QUEUE-INSERT (ITEM TIME EVQ)
  (PROG (NEW OLD ENTRY)
        (SETQ ENTRY (CONS TIME ITEM))
        (COND ((OR (NULL EVQ) (BEFORE? ENTRY (CAR EVQ)))
               (RETURN (CONS ENTRY EVQ))))
        (SETQ NEW (CDR EVQ))
        (SETQ OLD EVQ)
   LP   (COND ((OR (NULL NEW) (BEFORE? ENTRY (CAR NEW)))
               (RPLACD OLD (CONS ENTRY NEW))
               (RETURN EVQ)))
        (SETQ OLD NEW)
        (SETQ NEW (CDR NEW))
        (GO LP)))
```

is recognized as a non-header-cell insertion function for sorted lists. In this execution, the item to be inserted was (12 C (-1 0) 1) and the value of EVQ was

```
  ((24 A (0 1) 2) (24 A (0 -1) 2) (24 A (-2 0) 2) (24 A (1 0) 2) ...)
```

The ordering test, (BEFORE? ENTRY (CAR EVQ)) sorted the item to the top of the list, and therefore the splice-in did not occur. EVENTS-QUEUE-INSERT returned (CONS ENTRY EVQ) which evaluated to

```
  ((12 C (-1 0) 1) (24 A (0 1) 2) (24 A (0 -1) 2) (24 A (-2 0) 2) ...)
```

The function

```
(DEFUN METASTASIZE (RIGHT-CELL KEY-CELL)
  ((LAMBDA (NEW-CELL LOCATION)
     (INCREMENT-DIVISION-COUNT KEY-CELL)
     (MAKE-ROOM-BETWEEN KEY-CELL RIGHT-CELL GRID)
     (GRID-INSERT NEW-CELL LOCATION GRID)
     *[EVENTS-QUEUE-INSERT NEW-CELL (+ DIV-TIME 2) EVENTS-QUEUE]*
     (EVENTS-QUEUE-INSERT KEY-CELL (+ DIV-TIME 2) EVENTS-QUEUE))
   (CREATE-CANCER-CELL) (CELL-LOCATION RIGHT-CELL)))
```

ignores the value returned by EVENTS-QUEUE-INSERT on the indicated call, and consequently the results of the insertion were forgotten.

The mechanisms which support this analysis are described in the following chapters.

3. The Time Rover

The purpose of the time roving facility is to allow the user, and the bug experts, to query the execution history of the program undergoing analysis. The system was designed to support the style of investigation displayed in the scenario. In order to do this, the time rover maintains a complete trace of the events which occurred during execution, and allows arbitrary Lisp expressions to be evaluated as if particular program states were in effect.

The best way to explain the issues involved in time roving is to discuss its implementation. This is not intended as an overture to the inclusion of excessive detail. Since Sniffer was written to demonstrate a point rather than as a system utility, it was implemented with a conceptually simple design. Efficiency was not a concern.

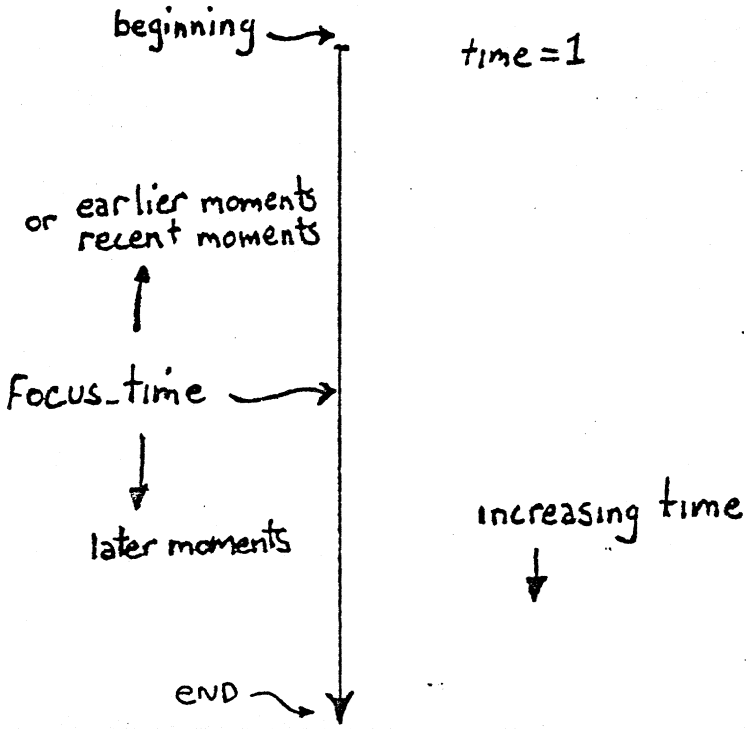
3.1 Terminology

The execution history of a program refers to the sum total of events which occurred while it was running; the flow of control, the sequence of side effect operations, etc. The execution trace refers to the physical structures which are used to represent that history.

Within an execution history there are various named times, or moments. Time can ordinarily be thought of as an integer. It starts at 1 and increases monotonically as execution progresses. The *beginning* and the *end* refer to the first and the last moments during the execution of the user's program. *Focus-time* corresponds to a specific moment in the execution trace. It is the focus of attention within the history.

There is also a convention for naming directions. *Earlier* moments are closer to the *beginning* and *later* moments are nearer the *end*. Figure 5 illustrates these ideas. A time-environment is an abstract object in which one can look up the bindings of variables and their properties, etc., which are in effect at a given time. For lack of a better method, all moments will be referred to in the present tense.

Fig. 5. Vocabulary for discussing time travel



3.2 Implementation

The time rover is composed of two parts, called the *keeper* and the *seer*, both of which are constructed as modified evaluators for Lisp. The keeper is used (primarily) to generate a history for the test program. It can be thought of as a careful evaluator which deposits records as it executes forms. The seer listens to the user's debugging requests. It has the ability to investigate and compare any of the states associated with the test program's history.

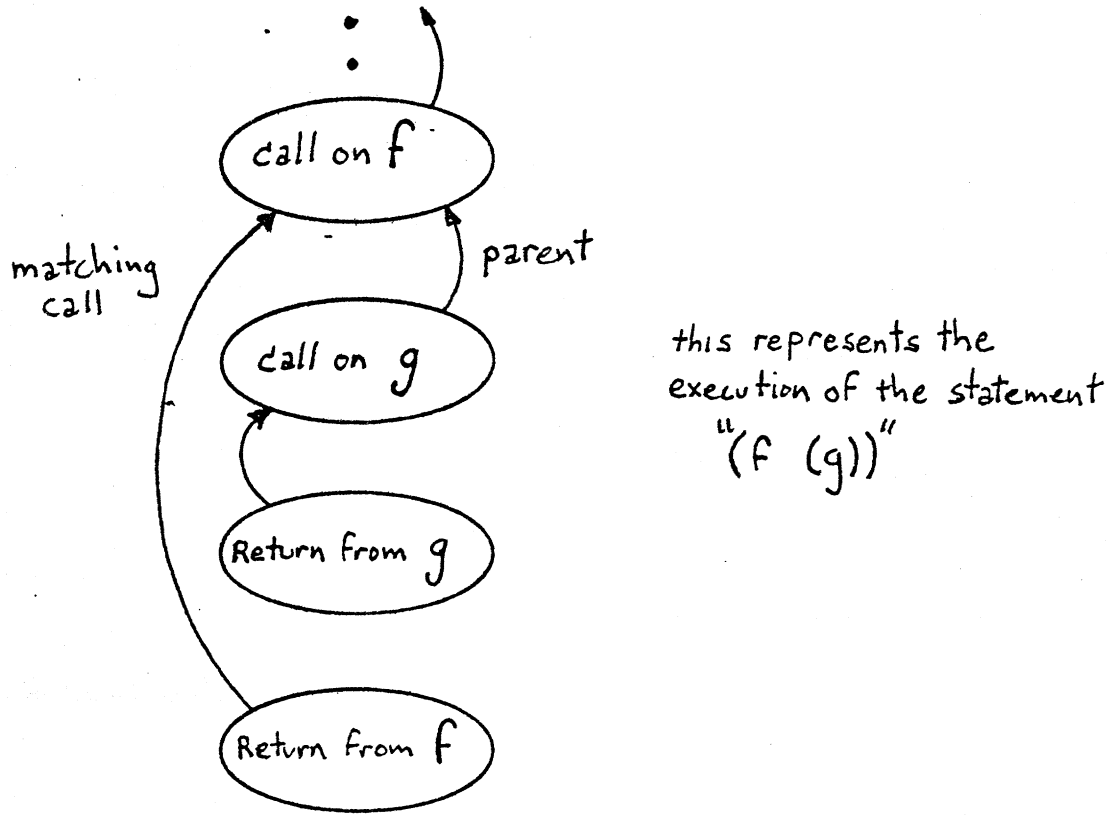
In the scenario, the keeper processed the original execution of `prosper`, and all forms typed by the user were handled by the seer. The special function, `@`, invoked a second usage of the keeper; it caused the keeper to evaluate an expression in the context of a specified time. (In some sense, the biggest distinction between the keeper and the seer is that the keeper can only think about one moment at a time, while the seer knows about all times at one moment.)

3.3 The keeper

The keeper implements a restricted version of Lisp, called K-lisp, which is different from normal Lisp in two ways; it considers code to be an immutable object, and it uses the execution trace as the environment for containing K-lisp objects. This includes "heap" data and variable binding information. The execution trace is a structure which totally orders control flow events, and side-effects events (changes in the contents of memory cells) with time. Conceptually this information is divided into two parts, the control flow history, and the incarnation series.

The control flow history records all calls and all returns from the evaluator. It is a straightforward extension of the Lisp stack, where no information is forgotten. Every *call moment* contains a link to the invocation time of its parent, and every *return moment* contains a link to its matching caller. (See figure 6.) This history contains more information than is necessary to record the control flow unambiguously (only the choices taken at branch points are strictly required), but it was more convenient for my purposes to have the data in this form.

Fig. 6. A control flow history



The incarnation series is a time ordered sequence for the values which each memory cell acquires during the execution of the test program. This information is stored in terms of [name, binding] pairs, called trace-cells. A trace-cell is an immutable object that records the contents of a cons (or the value of an atom) at a particular time. The name component of a trace-cell is analogous to the address of a cons in Lisp. It provides a handle on all of the versions of a given cell. The binding field of a trace-cell contains a *car-part* and a *cdr-part* which represent the car and cdr of the corresponding Lisp cons. Trace-cells are invisible to the programmer.

In the keeper, a value is a *name.* The data associated with a given name (*id* or *cell-id*) at a given time is found by scanning the incarnation series for the most recent trace-cell with the appropriate id. This search fulfills a role which is exactly analogous to looking up an address in normal Lisp. During the evaluation of the test program, the current execution time is used as the starting point for scanning the incarnation series. During debugging, that time is supplied by the seer.

The primitive operations of K-lisp are modified to accommodate trace-cells. The functions which produce side effects cause trace-cells to be deposited, and the information obtaining operations, *car, cdr,* and *symeval* are modified to access these structures via search. (I will discuss the new versions of *eq* and *equal* in a later section.) For example (see figure 7), the function *cons* in the statement

```
(cons 'a 'b)
```

produces the trace-cell [cons-24, a.b], which indicates that the binding associated with the cell-id, *cons-24* represents the (traditional) cons of the atoms *a* and *b.* (The *cons* function is a side-effect operation in the sense that it allocates storage where none was required before.) Trace-cells, like conses, contain the values of Lisp objects. The statement

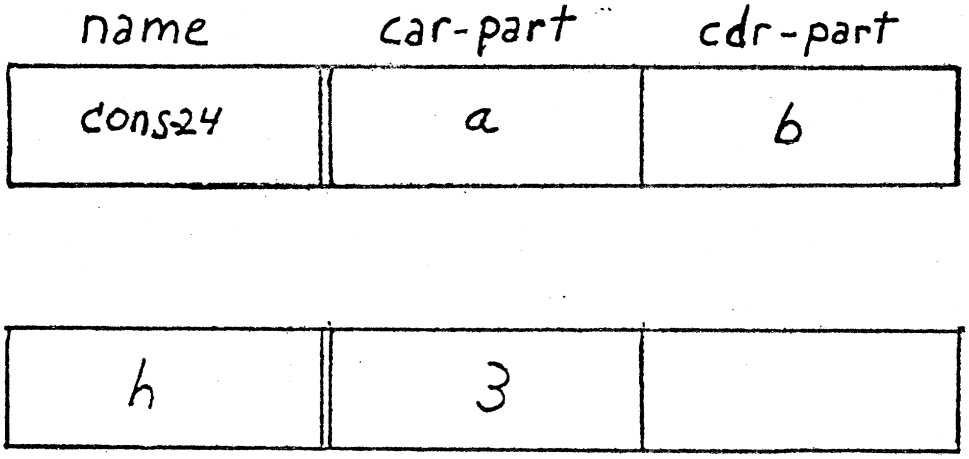
```
(cons a b)
```

would produce a different trace cell, who's *car-part* was the value of *a* and who's *cdr-part* was the value of *b.* The functions *rplaca* and *rplacd* create similar trace-cells, except that the name field contains the id of the cell which is being updated. The function *setq* in the statement

```
(setq h 3)
```

results in a trace-cell who's name is the atom, *h,* and who's binding field has a *car-part* containing the number 3. Only mutable objects need to have trace-cells to record the sequence of their values.

Fig. 7. Some example trace-cells



Numbers and similar constants can appear in the binding parts of trace-cells, but not in name fields.

The operations *car*, *cdr*, and *symeval* each map a cell-id into another cell-id. The *car* of a cell-id is the *car-part* of the corresponding trace-cell (the one in effect at the current time). Similarly, the *cdr* of a cell-id is the *cdr-part* of the associated trace-cell. All of these functions involve an identical search through the incarnation series. For example, the function *symeval* takes in an id (which must be an atom name), scans the incarnation series for the most recent trace-cell with that id in the name field, and outputs the *car-part* of the trace-cell which is discovered.

3.3.1 An example of the evaluation process

Figure 8 shows a collection of snapshots of the incarnation series as the following statements are executed.

```
(setq y (cons 1 nil))
(setq z (cons 2 y))
(rplaca (cdr z) 3)
```

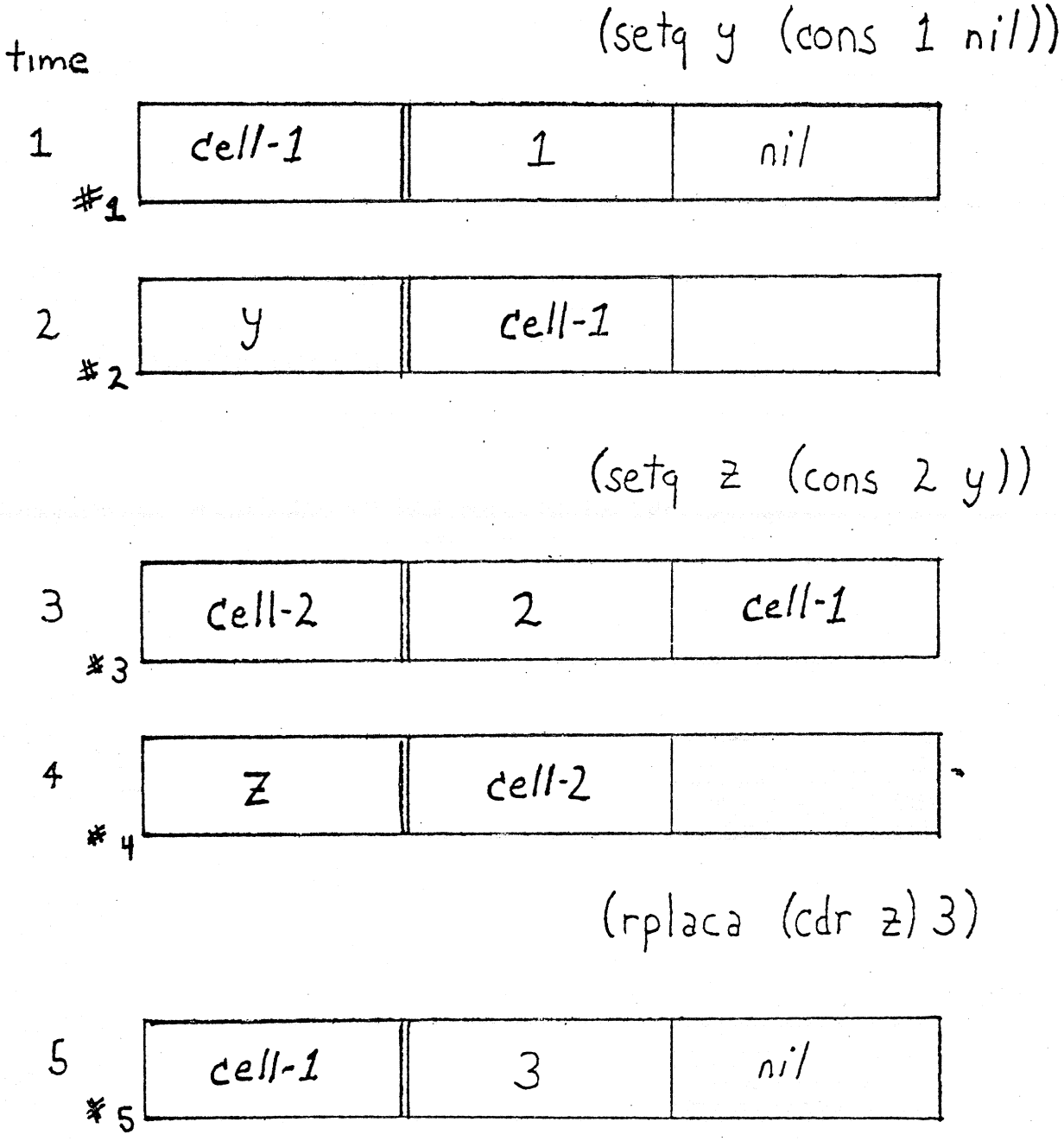
The first event is the creation of the trace-cell for `(cons 1 nil)`. The name field is arbitrarily set to *cell-1*, and the trace-cell is deposited at time 1. The setq operation deposits a trace-cell with the name field `y`, and a binding field who's *car-part* is the cell-id, *cell-1*. No pointers are involved. Similarly, in the trace-cell which is deposited by `(cons 2 y)`, the value of `y` is represented by *cell-1* again. This process continues until `(rplaca (cdr z) 3)` is evaluated. In normal Lisp, this side-effect would have changed the contents of an existing cell. In the keeper, a new trace-cell is deposited with the same name field, *cell-1*.

In order to evaluate Lisp expressions, the keeper has to find the appropriate trace-cell every time a cell-id is referenced (there may be many with the same name). For example, in figure 8, the value of `y` at time-2 is found from trace-cell #2 to be the cell-id, *cell-1*. To print out the value of `y`, the binding of *cell-1* at time-2 has to be printed. In this case, the contents of trace-cell #1 are the correct result. The list "(1)" is printed.

In order to evaluate the predicate `(@ time-5 '(car y))` the keeper has to discover that `y` was changed by an indirect side effect through `z`. This process is accomplished as follows. Starting from time-5, the keeper looks for the most recent setq record for the atom `y`. The value of `y` turns out to be the id, *cell-1*, which was discovered from the trace-cell deposited at time-2. Next, the keeper takes the `car` of *cell-1*, in the context of time-5. It scans backwards from time-5, looking for the most recent version of *cell-1* and returns the *car-part* of the resulting trace-cell. Trace-cell #5 has the appropriate name, and the number "3" is returned.

In order to print out the elements of a list in the context of a given time, the keeper has to interpret each of the cell-ids involved. For example, the value of `z` at both time-4 and time-5 is *cell-2*, but the list it represents at time-4 is composed of trace-cells #3 and #1 (the list "(2 1)"). At time-5, `z` is built from trace-cells #3 and #5, corresponding to the list "(2 3)".

Fig. 8. The development of the incarnation series during execution



3.3.2 Effeciency considerations

The time rover was implemented with a list like representation for its environment in order to make the system easy to code. Once it was implemented, I discovered that it was slow, but not quite so slow as expected. For simple requests, the keeper responded almost as quickly as the normal Lisp interpreter. However, the time requirement for each reference unfortunately increases with the size of the execution trace. At the end of the scenario, the time rover required approximately half of a second to locate each cell-id.

The searches involved in running the test program can be entirely eliminated by introducing a new data structure, called the now-array, to maintain the *end* time-environment. (This environment is the one normally associated with a running program, it always holds the state of the latest moment of execution.) This table would contain a mapping of cell-ids to their current bindings. In different words, the now-array would be a shallow binding of cell-ids to *car-part, cdr-part* pairs from trace-cells. Since cell-ids can be chosen freely, they can be set up as indices into successive memory locations of the now-array. This would essentially eliminate all searches for cell-ids (at a factor of two overhead in space).

The now-array would not speed up the execution of debugging requests. These requests typically access a number of time-environments in rapid succession, which suggests that a search paradigm is more reasonable than the alternative of updating the now-array to contain the time-environment of *focus-time*, whenever *focus-time* changes.

A second improvement would be to move to a non-linear representation for the execution trace. Since the critical issue is to find cell-ids as fast as possible, a hashing scheme on cell names is a possibility. I did not employ this approach because there was some subtlety involved in integrating it with the need to represent alternate evaluation sequences (see below).

In any case, the memory requirement for the keeper grows with the duration of execution. At some point, this will threaten to exceed the capacity of any machine, in which case it would be possible to "forget" about certain portions of the execution history. These regions would then become opaque to the time rover. In running the scenario, no memory capacity problems were encountered.

3.4 The seer

The function of the seer is to provide the user with a uniform mechanism for operating on data from the execution trace, and for manipulating the objects defined in his own local debugging environment. The seer is constructed as an evaluator for Lisp that is extended to contain time-stamped objects, called *t-pairs*, which refer to data from the incarnation series.

A *t-pair* contains two parts, a *reference time* and a *cell-id* where the reference time specifies the time-environment to use for interpreting the cell name. Reference times are sticky, in the sense that the *car* of a t-pair is another t-pair with the same reference part. This approach allows the user to change the perspective used to view an entire Lisp object by altering the reference time attached to its topmost cell-id. A t-pair is represented here as a bracketed pair of the form {*time id*}.

The primitive operations of the seer are modified to accommodate this new data type. If a primitive is called on a normal Lisp object, then it is evaluated in the normal way (this might yield a t-pair). When a primitive is applied to a t-pair, it is evaluated with the aid of the corresponding operation of the keeper. For example, from figure 8, *symeval* of {time-4 z} is the t-pair {time-4 cell-2} where cell-2 was obtained by applying the keeper's *symeval* function to z at time-4.

The function "@" (which invokes the keeper's evaluator on a Lisp form) can be used to state the effect of these primitives in a more concise form.

```
(symeval {t id}) => (@ t '(symeval id))
(car {t id}) => (@ t '(car id))
(cdr {t id}) => (@ t '(cdr id))
```

@ returns a t-pair who's reference time is the time supplied by its first argument.

3.4.1 Alternate time-tracks

It is not immediately clear how to interpret the application of a side effecting primitive to a time-stamped object. The issue is that a t-pair refers to an object from the history of the test program which was never subjected to the side effect that the user is requesting. (Information obtaining operations are benign in this sense. They have no potential for altering the data in the trace.) If the execution trace is intended to record the actual history of the program, the question is how can side effects created by the debugger be factored in?

There are many very confusing ways to resolve this question. If the debugging session is

considered to occur after the test program is executed then a side-effect to a variable, say at time-10, would actually occur at a moment which is later than any moment in the execution history. This implies that a debugging request which accesses the supposedly side-effected data at time-11 finds that nothing has changed.

The approach I take is to interpret all debugging requests that access the history of the code as explorations into alternate time-tracks for the test program's development. These debugging requests are processed as if the test program executed them at the specified time. For example, in the context of figure 8, the effect of the statement

```
(@ time-4 '(rplacd (cdr z) 1))
```

is to grow a branch off of the incarnation series at time-4 (forming an incarnation tree) and to deposit a trace-cell for *cell-1* at that time. The side effects created by the functions *setq, cons,* and *rplaca* are handled in a similar way. (See figure 9.)

This approach implies a small redefinition of the function "@". I have described @ as a utility for invoking the evaluator of the keeper. To be more specific, @, in the statement

(@ *time* '*expression*)

instructs the keeper to form a branch in the incarnation tree, and then hands the *expression* to the keeper to be evaluated in the context of the time-environment defined by *time.* (The seer evaluates the parameters to @.) @ returns a t-pair which packages together the cell-id returned by the keeper and the time at which the keeper finishes its evaluation. A time can be interpreted as a pointer into the incarnation tree, which bi-directionally links trace-cells.

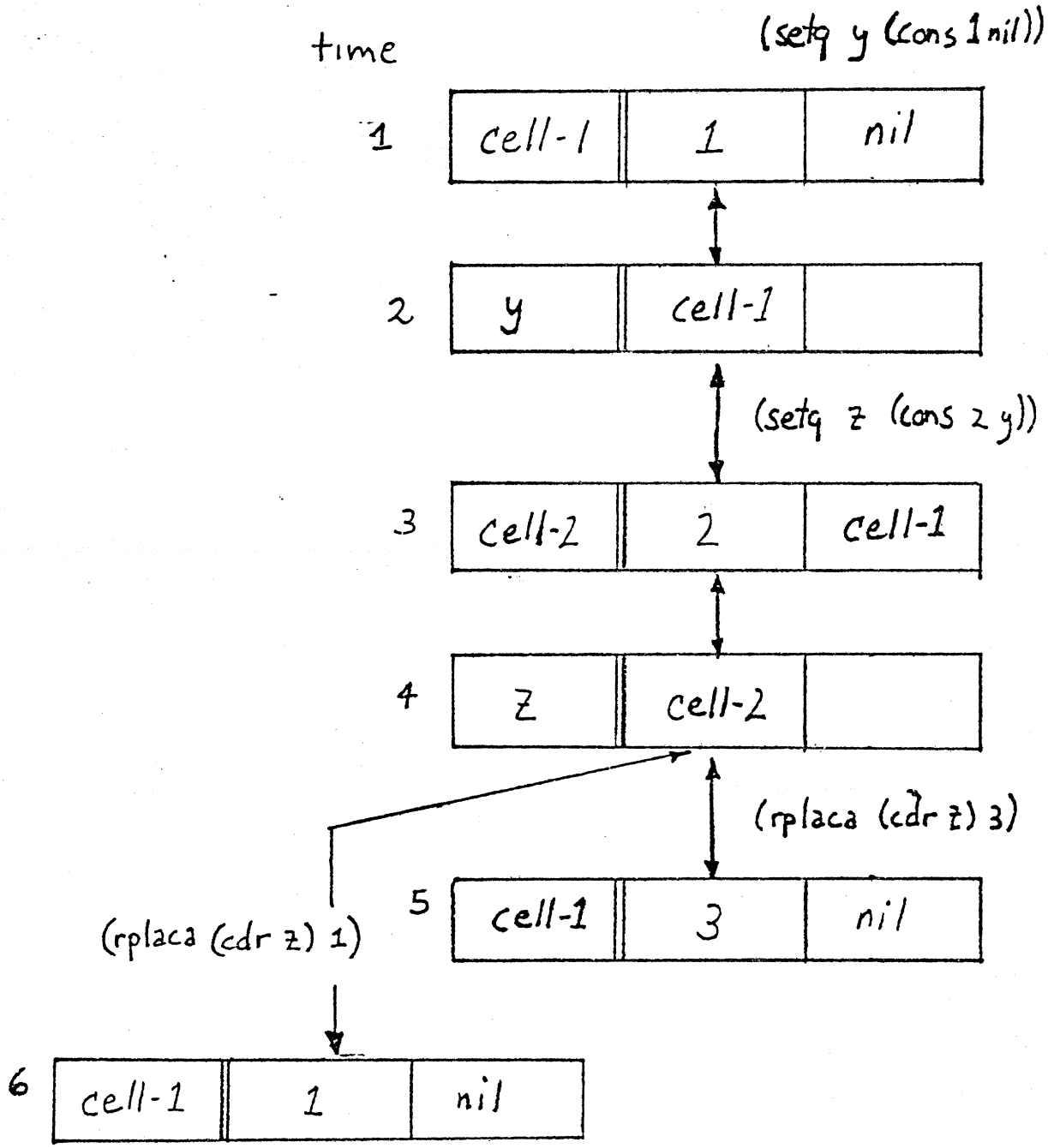
The seer can use the function @ to retrieve information from the environment of the keeper, but the keeper cannot access data defined in the seer. This occasionally causes some confusion. For example, the following expressions (in the seer)

```
(setq D '(a b c))
(@ time-2 '(setq y D))
```

will result in an error when the keeper attempts to *symeval* D at time-2, assuming that D is not defined in the context of the test program at that time. (The keeper does have limited access to the seer, in that it can run functions which the user defines in the course of debugging. These functions, must be runnable in K-lisp. They may not reference t-pairs.)

The definition of @ makes it possible to express the action of the seer's primitives on t-pairs by

Fig. 9. An example of an alternate time-track
This figure shows the growth of a branch in the execution history in response to the code statement shown.



the following rewriting rules.

```
(symeval {time id}) => (@ time '(symeval id))
(car {time id}) => (@ time '(car id))
(cdr {time id}) => (@ time '(cdr id))
(setq {time id} x) => (@ time '(setq id x))
(rplaca {time id} x) => (@ time '(rplaca id x))
(rplacd {time id} x) => (@ time '(rplacd id x))
```

The information obtaining operations create degenerate branches of the incarnation series (the time does not increase), and the side effecting operations augment the data in the trace.[1] Note that the *cons* of two t-pairs within the seer is not implemented in terms of the keeper's primitives. The statement

```
(cons (@ time-4 'z)(@ time-2 'y))
```

simply creates a cons cell in the environment of the seer which contains the resulting t-pairs.

3.4.2 Equality and coreference

The concepts of equality and coreference have to be extended to fit an environment where many versions of data cells are available simultaneously. In normal Lisp, there are only two ways to compare objects. One can ask if they are *eq*, meaning that they have the same name or address (which is equivalent to asking if they are coreferent), or if they are *equal*, meaning that they contain isomorphic data structures.

In the seer, more distinctions are available. One can ask if two t-pairs refer to the same object in the keeper (I call this test *unmodified*), or if two cell-ids are the same (*eq*). These questions arise when objects are compared across times. For example (see figure 8),

```
(eq (@ time-2 'y) (@ time-5 'y))
```

is true. Here, the list contained in `y` is different at the two times although the top level cell-id which is the value of `y` is *cell-1* in both cases. (`y` contains `(1)` at time-2 and `(3)` at time-5.) The statement

---

1. This is not strictly true. Since the structures representing the control flow history are merged into the execution trace, the time does change on every call to the keeper. However, for the purpose of the primitive operations, it does not change in any interesting way.

```
(unmodified (@ time-2 'y) (@ time-5 'y))
```

is false. This test shows that the value of y was changed between the two times.

When these predicates are extended to lists, one can ask if two lists contain the same cell-ids at every level (called *eq**), or if they involve the same trace-cells at every node (*unmodified**). *Unmodified** is the coreference test in the time roving environment. *Eq** is a weaker function. For example, suppose that an identical copy of the variable y is created by executing the statement

```
(@ time-4 '(rplaca (cdr z) 1))
```

(see figure 9). This deposits a record for *cell-1* in a side branch at time-6. In this case, the expression

```
(eq* (@ time-6 'z) (@ time-4 'z))
```

is true, but

```
(unmodified* (@ time-6 'z) (@ time-4 'z))
```

is false.

Note that two lists are not necessarily identical if their top level trace-cells are the same. There is always the possibility that some internal cell has changed across the two times. From figure 9,

```
(unmodified (@ time-5 'z) (@ time-4 'z))
```

is true (z evaluates to *cell-2* in both cases), but

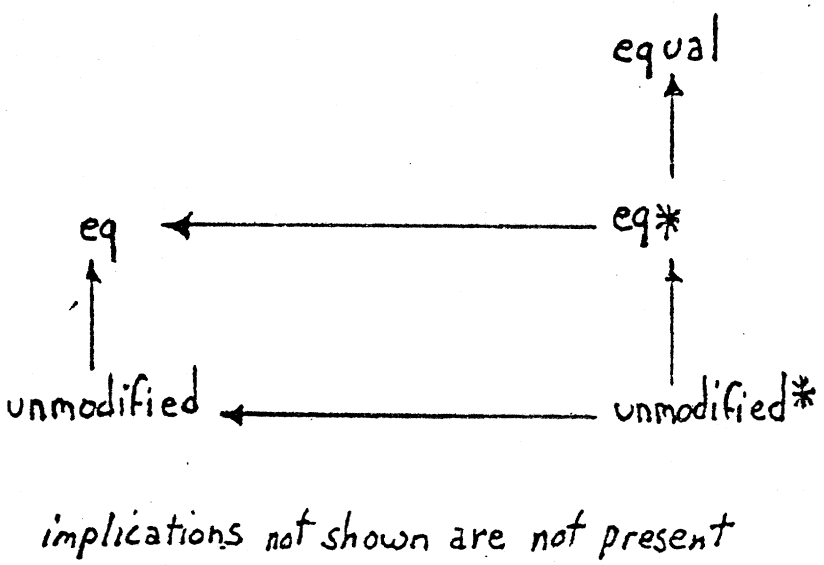
```
(unmodified* (@ time-5 'z) (@ time-4 'z))
```

is false. (*Cell-1* was updated between the two times.)

The function *equal* remains essentially unchanged in the context of the seer. It still tests for isomorphism of structure. There is no requirement that the lists share the same trace-cells or even that the same cell-ids are involved. The atoms at the leaf nodes of the tree must be identical.

The relationship between these functions is summarized in figure 10.

Fig. 10. The heirarchy of equality tests
The equality tests for lists represented in trace structures are stronger than the analogous tests on cell-ids: *eq** implies *eq* and *unmodified** implies *unmodified*. The converse is not true. *Unmodified** implies *eq**, because lists with the same trace-cells must contain the same cell-ids. *Eq** implies *equal* because lists built with corresponding cell-ids must match at the level of atoms.



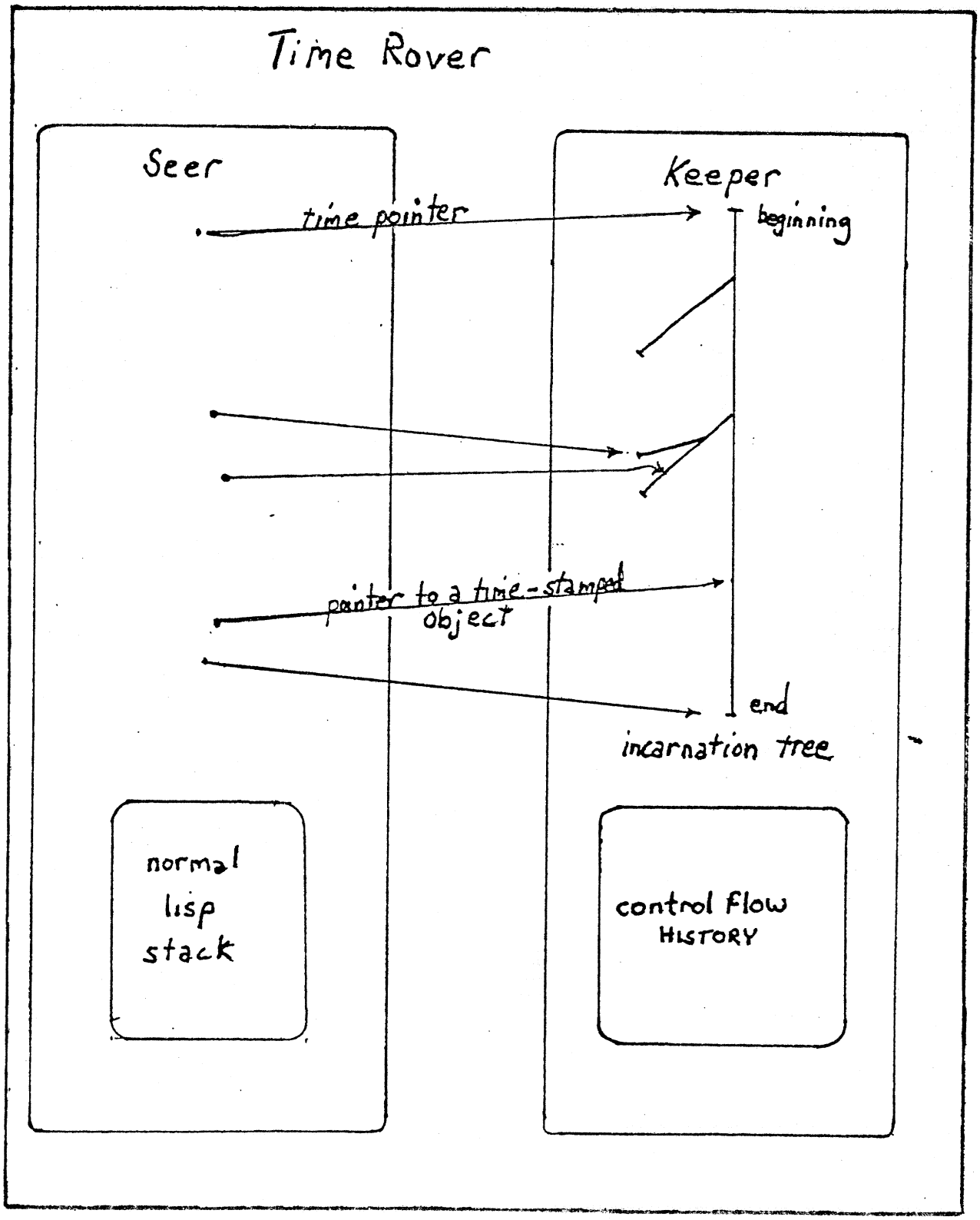
3.5 A summary of the keeper and the seer

The keeper and the seer define a mechanism that allows the user to execute and then examine the history of a test program. The keeper creates the execution history, and evaluates any requests submitted by the seer which access that data. The seer provides the user with a Lisp environment for executing debugging requests. It answers questions about the execution history by employing the facilities of the keeper. Figure 11 shows the relationship between these systems.

The overall environment which the system presents has the user's debugging requests occurring in a kind of a super-time which is not ordered with respect to the execution history. From the user's perspective, all of the information in the trace is equally accessible.

The use of alternate time tracks makes it possible to move to moments in the test program's past and evaluate arbitrary Lisp expressions in those contexts. The user can define functions, and execute them in any time-environment, or explore hypotheses about the test program's behavior by re-executing portions of the code on modified data. The alternate histories which these actions create can themselves be investigated in the same manner.

Fig. 11. An overview of the time rover



The functions of the keeper and the seer could conceivably be combined into a single evaluator that would have an extra degree of freedom, namely time. In this system, called the *time-probe*, it would be possible to write programs that routinely call procedures which will be defined in the future to modify data which was current at some time in the distant past. The difference between the time rover and this hypothetical system is that the time-probe can travel in its own history. Neither the seer nor the keeper has this ability (and it is not clear that they require it).

The creation of the time-probe is left for future research.[1]

3.6 Methods for specifying times

The primitives for locating times are cast in the framework of search through the incarnation series. There is a notion of the focus of attention, called the *focus-time*, which can be moved throughout the execution history. The searches for other moments move either forward or backwards from that time.

Time is a data type recognized by the seer. There are two functions which yield times; **future-when** and **past-when**. The syntax is

(future-when *form*)

where *form* is an arbitrary predicate evaluated by the seer (it may contain calls on `@` which invoke the keeper). The function future-when scans forward in time from *focus-time* and returns the first moment when *form* yields a non-nil (and non-error) result. Past-when performs the analogous function for moving towards earlier moments in the history.

The implementation for these functions is fairly intricate. It would be prohibitive to attempt to apply *form* at every moment in the history which is scanned, so the search functions first compute the reference set of cell-ids accessed by *form*, and then move attention to the nearest moment when one of those cell-ids has a different binding. At the resulting time, *form* is reevaluated and the reference set computed once again. The process repeats until *form* returns a non-nil value (success), or until the search passes beyond the boundaries of the incarnation series (failure).

The search mechanism is also capable of detecting transitions in the values of *form*. For example,

---

1. The time-probe would have to deal with a few very serious problems, including the *temporal fun-arg problem*. This occurs when a function, defined in the past is passed as an argument into a future when its definition is different. Or worse still, a function defined in one time-track can be passed into an alternate branch in which it never will, and never has existed at all.

the expression from the scenario,

```
(move-to (past-when '(just-became-true
                      '(@ ? '(eq (cell-type cell) 'c)))))
```

caused the form

```
(@ ? '(eq (cell-type cell) 'c))
```

to be applied at the moment discovered by the scan, and the immediately preceding moment. The function, `just-became-true`, identifies a particular kind of transition in the value of its form. Since a scan can cause expressions to be applied in time-environments where they yield errors, `just-became-true` looks for a transition from either a nil or error result, to a non-nil value. The implementation of sniffer contains a number of similar functions; `error-to-true`, `error-to-false`, `false-to-true`, etc., as well as two special functions, `just-about-to-become-true` and `just-about-to-become-false` which return the moment immediately before a transition is going to occur. (All transitions are defined to start at earlier times and finish at later ones. The transition functions are not sensitive to the direction of search.)

The search functions can also employ predicates which depend upon data in the control flow history. For example, the expression

```
(future-when '(during metastasize))
```

(not shown in the scenario) returns the next time when execution is within the definition of metastasize. Since the records in the control flow history provide the code associated with each call and return from the evaluator, detecting *during-ness* is not very hard. The procedure ascends the parent hierarchy of function calls to see if it locates the expression which is the definition of metastasize.

It turns out that the interaction between these kinds of requests and the search mechanism is somewhat tricky. In order for the search functions to know when next to apply a form, each predicate on control flow has to identify the borders of its current truth value. In some cases this is easy; `during` knows that it ceases to apply at the endpoints of its span (which are trivially available from the execution trace). However, if `during` does not apply at the current moment, it has to find the bordering times where it does. This partially subverts the purpose of the scan mechanism, which was attempting to find those moments to begin with. Some more sophisticated approach may be called for.

4. The cliche finder

The cliche finder performs two functions within Sniffer; it recognizes small algorithms from the test program in order to provide the bug experts with a context for identifying errors, and second, by identifying algorithms, it raises the level of the vocabulary which the system can use to describe code.

For example, in order to identify the error described in the bug report (see page 18), the cliche finder recognized that `events-queue-insert` implements a particular kind of list insertion (a non-header-cell insertion for sorted lists). It also identified cliches which were components of that insertion, namely a splice-in operation, an ordering predicate test and a list enumeration, some of which it referred to by name in the bug report.

The cliche finder is composed of a collection of algorithm detectors which operate on an alternate representation for programs, called a *PLAN*. PLANs (developed by Waters, Rich and Shrobe [Waters 1978] [Rich and Shrobe 1976]) are a powerful tool for supporting program recognition because they are a language independent notation, and they represent small algorithms in an essentially canonic form. The generality of the cliche finders depends upon these properties of PLANs.

4.1 An overview of PLANs

PLANs identify several critical constraints on the representations of algorithms. (See [Waters 1978] for a detailed discussion.) I summarize the main points below.

PLANs ignore the way in which control and data flow is implemented. For example, it makes no difference if the control structure for a program uses conditionals or goto statements, both map into the same PLAN. Similarly, all the possible methods of using variables to hold partial results or propagate values are judged equivalent. PLANs are based on data flow; they extract only the essential interconnections between operations that produce and consume data in code.

PLANs associate related segments of code which may have been widely separated in the original text. A PLAN is a compound object composed of data flow related segments. The fact that one piece of code outputs data which another consumes is a simple proof that both are working towards some unified goal. The consequence of this organization is that feature detection in PLAN space involves far less search than it would require in the original text for the code.

The PLAN representation is partitioned into fragments which have stereotyped behaviors. This allows complex programs to be understood in terms of simple purposeful parts. For example, iterative and recursive routines are represented by a single PLAN structure (a PLAN Building Method, or *PBM* in Waters' terminology) called a *temporal composition* which can contain five types of components; *initializations*, *generators*, *filters*, *accumulators* and *terminators*. (The output of his analysis system labels the segments which fulfill each of the five roles.) An initialization is a segment that is executed once before a loop is entered. A generator produces a sequence of values that are used in later calculations (a list enumerator is an example of a generator). Filters restrict the sequence of values which are available beyond their location in the code. Accumulators perform calculations, they remember results. Terminators are like filters in that they restrict sequences of values, however, they may also stop the execution of a loop. The remaining plan building methods categorize the program actions in straight line code. Taken together, the PBMs provide a complete parse of a program into these purposeful parts. (The mechanisms which perform this analysis are too lengthy to describe here. See [Waters 1978] for a full explanation.)

The result of features described above is that many textual representations for the same algorithm are mapped into identical (or nearly identical) PLANs. For example, if the function, `events-queue-insert`, is implemented using either of the expressions in figure 12, it analyzes into the exact same PLAN. This is true even though the forms involve different control structures, different variable names, and distinct Lisp primitives.

4.2 An example of cliche recognition

The algorithm recognizers identify procedures by matching their PLANs against known cliches. This match must be essentially exact. (The cliche finders can tolerate variations at the level of ignoring extraneous detail.) For algorithms of complexity of `events-queue-insert` this approach has been successful. The recognition of larger programs will require more sophisticated methods. (I discuss some alternative approaches in the section entitled *extensions*.)

The following three figures present the PLAN for `events-queue-insert` in its entirety. These diagrams explicitly represent a considerable amount of information which is hidden in code, and they contain some special notation as well. However, most of the detail can be safely ignored. The figures are presented in order to motivate specific examples which draw on portions of the PLANs.

Fig. 12. List insertion programs which map into the same PLAN

-[a]-

```
(DEFUN INSERT (DATUM KEY QUEUE)
  (LET ((OBJECT (CONS KEY DATUM)))
    (COND ((OR (NULL QUEUE) (BEFORE? OBJECT (CAR QUEUE)))
           (CONS OBJECT QUEUE))
          ((DO ((NQ (CDR QUEUE) (CDR NQ))
                (OQ QUEUE NQ))
               ((OR (NULL NQ) (BEFORE? OBJECT (CAR NQ)))
                (RPLACD OQ (CONS OBJECT NQ))))))))
```

-[b]-

```
(DEFUN EVENTS-QUEUE-INSERT (ITEM TIME EVQ)
  (PROG (NEW OLD ENTRY)
        (SETQ ENTRY (CONS TIME ITEM))
        (COND ((OR (NULL EVQ) (BEFORE? ENTRY (CAR EVQ)))
               (RETURN (CONS ENTRY EVQ))))
        (SETQ NEW (CDR EVQ))
        (SETQ OLD EVQ)
   LP   (COND ((OR (NULL NEW) (BEFORE? ENTRY (CAR NEW)))
               (RPLACD OLD (CONS ENTRY NEW))
               (RETURN EVQ)))
        (SETQ OLD NEW)
        (SETQ NEW (CDR NEW))
        (GO LP)))
```

4.2.1 Notation

PLAN diagrams contain three kinds of entities; boxes, solid lines and dashed lines. Boxes represent actions which may be either primitive or compound. A primitive action corresponds to a black box in the code, such as a `cons` statement in Lisp. There are eleven types of compound actions, these include *conjunctions*, *predicates*, and *conditionals* for representing straight line code, and *filters*, *accumulations* and *terminations* for representing looping behavior. Dashed lines represent control flow, solid lines represent data flow. For example, the diagram of figure 13 represents the top level PLAN for `events-queue-insert` as the PBM *exclusive or*, where the predicate

```
(or (null evq) (before? entry (car evq)))
```

determines whether the function returns through a `cons`, or enters the expression containing the

Fig. 13. The top level PLAN for events-queue-insert

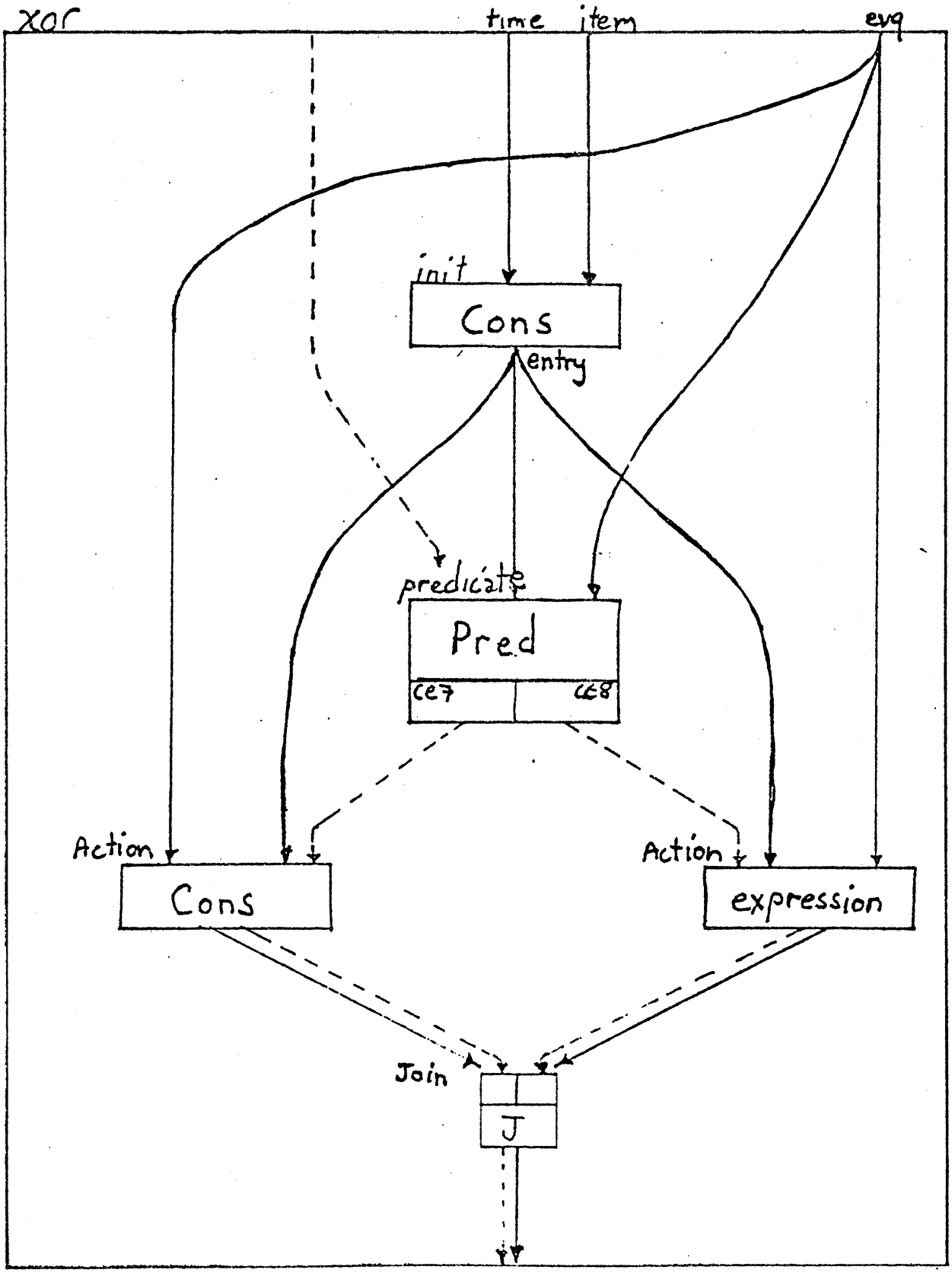


Fig. 14. The predicate for testing list elements

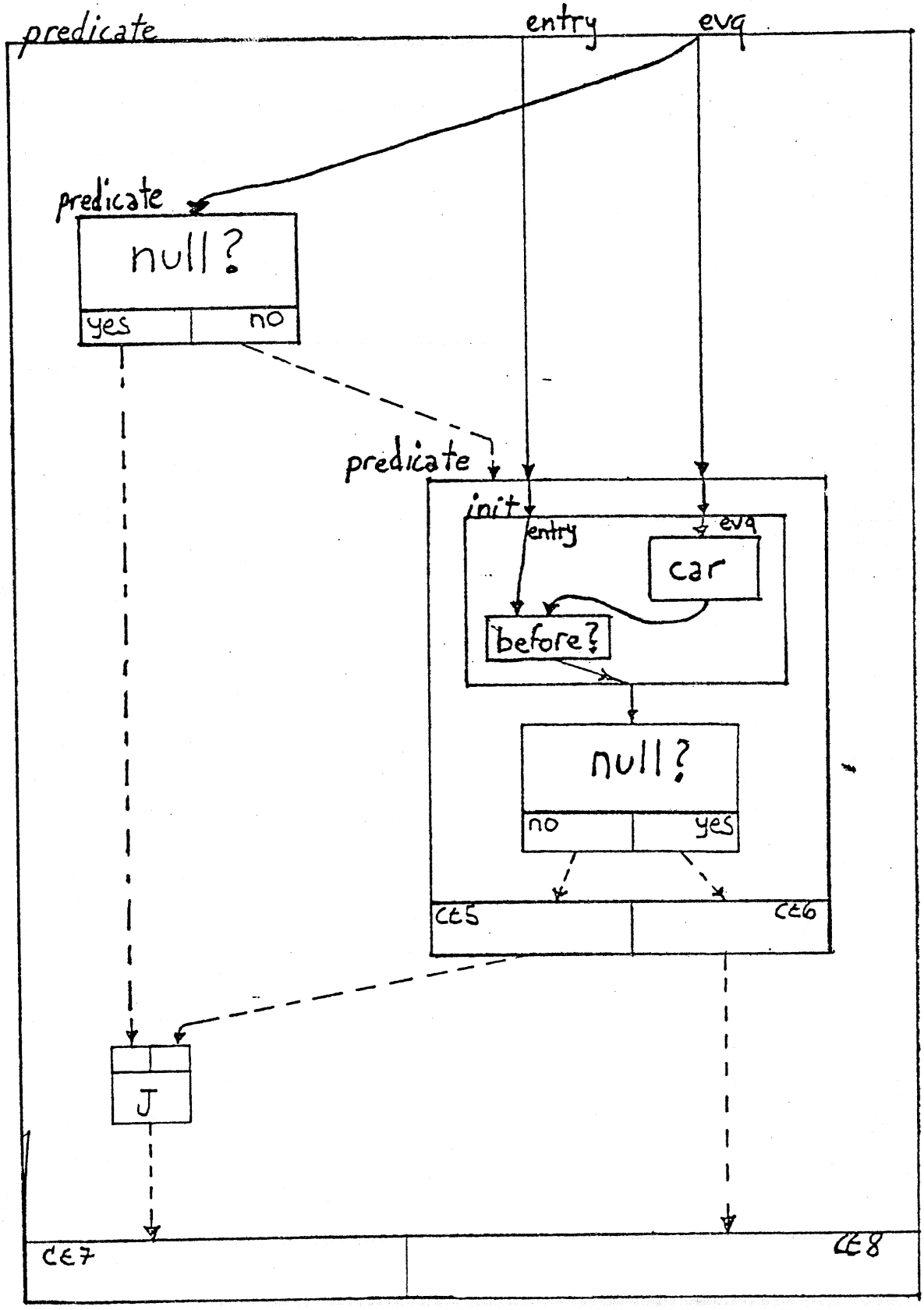
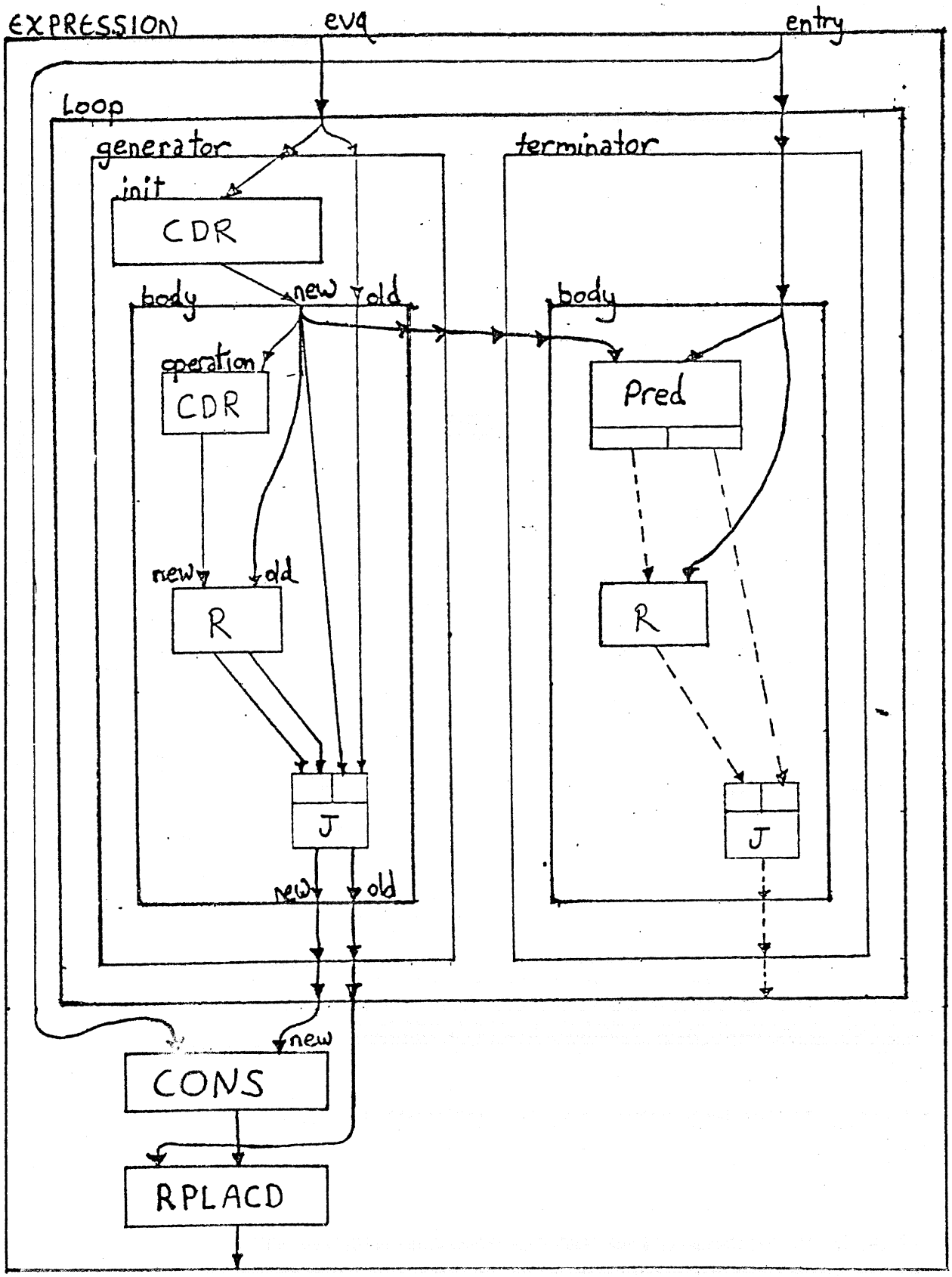


Fig. 15. The PLAN for inserting an element in a list



body of the loop. (See figure 12b for the code for events-queue-insert.) There is data flow from the inputs *item* and *time* to the `cons` function

```
(cons time item)
```

which produces the data value *entry* that is tested by the predicate above. The diagram contains branched control flow to show that there are two possible outcomes of the test. The box at the bottom labeled *join* preserves the one-in one-out property of compound actions.

Each compound action has certain allowable components, called *roles*. There is a grammar (which I will not present here) that restricts the elements which can fulfill a given role, and also determines the number and the types of roles permitted in compound actions. In the figures, the role a component fulfills is printed on its upper left-hand corner.

4.2.2 The PLAN for events-queue-insert

The PLAN for `events-queue-insert` is broken up into a conditional that determines whether the loop is to be entered (figure 13), a compound predicate which represents an ordering test (figure 14) and a PLAN for the loop which contains the splice-in portion of the insertion (figure 15).

The most interesting part of the PLAN is figure 15. This loop is decomposed into a *generator*, which enumerates the elements of the events-queue (`evq` in the diagram), and a *terminator* which controls the execution of the loop body.

The generator represents the code segment

```
(defun events-queue-insert (item time evq)
  (prog (new old entry)
    ...
        (setq new (cdr evq))
        (setq old evq)
     lp ...
        (setq old new)
        (setq new (cdr new))
        (go lp)))
```

Generators are composed of an optional *initialization* and a *body* which is the portion that is executed many times. The body can contain an *operation*, a *recursion* and a *join*, which I explain below.

The sole input to the generator is the variable named `evq`. This data passes through the *initialization*

```
(cdr evq)
```

which outputs the data (labeled *new*). The body of the generator receives two inputs, *new* and *old*, where *old* starts as the unmodified events-queue. The *operation* of the generator body is the function `cdr`, from the code

```
(cdr new)
```

above. At each successive iteration, this operation causes *new* to become successive sublists of the events-queue. The data values *new* and *old* become the output of the generator, emerging from the data join box in the diagram. The join indicates that the output can come from one of two places; it can be the input to the generator body (in case the generator terminates), shown by the data lines that pass straight through the diagram, or it can come from the box labeled "R" which stands for a recursive instance of the enumerator. The cross over of data, where *new* becomes *old* at the next iteration, can be seen from the change of labels on the data flow lines at the input ports of the R segment.

The terminator for the loop is conceptually executed in parallel with the generator. At each iteration, the predicate compares *entry* with the value of *new* that is obtained from the top of the body portion of the generator segment. If the predicate returns through its right hand branch, control passes out of the terminator segment, and iteration of the generator body is stopped as well.

4.2.3 Feature recognition in cliches

The algorithm recognizer for `events-queue-insert` is constructed as a hierarchy of procedures which identify each of the segments in the PLAN. This cliche finder operates via an exact match paradigm; essentially all of the structures present in the diagrams are required for a non-header-cell insertion to be found. The elements of the insertion that were referred to in the bug report (see page 18) were identified by a feature extraction process that was applied after `events-queue-insert` had been identified as a whole.

For example, the input to `events-queue-insert` containing the queue is identified as the source of the data flow line that enters the generator portion of the loop in figure 15. The name of the program variable associated with this input (`evq` in this case) is obtained from an annotation in the PLAN. (Waters' analysis system provides the code associated with PLAN segments whenever

possible.)

The item to be inserted is identified from figure 13 as the first input to the `cons` function fulfilling the *action* role of the PLAN. By tracing this data flow line to its source, the entry can be identified as the output of the `cons` function of the *initialization*. (If the entry had been one of the inputs of `events-queue-insert`, there would not have been an initialization. The source of the data flow line would have been a lambda input in the PLAN.)

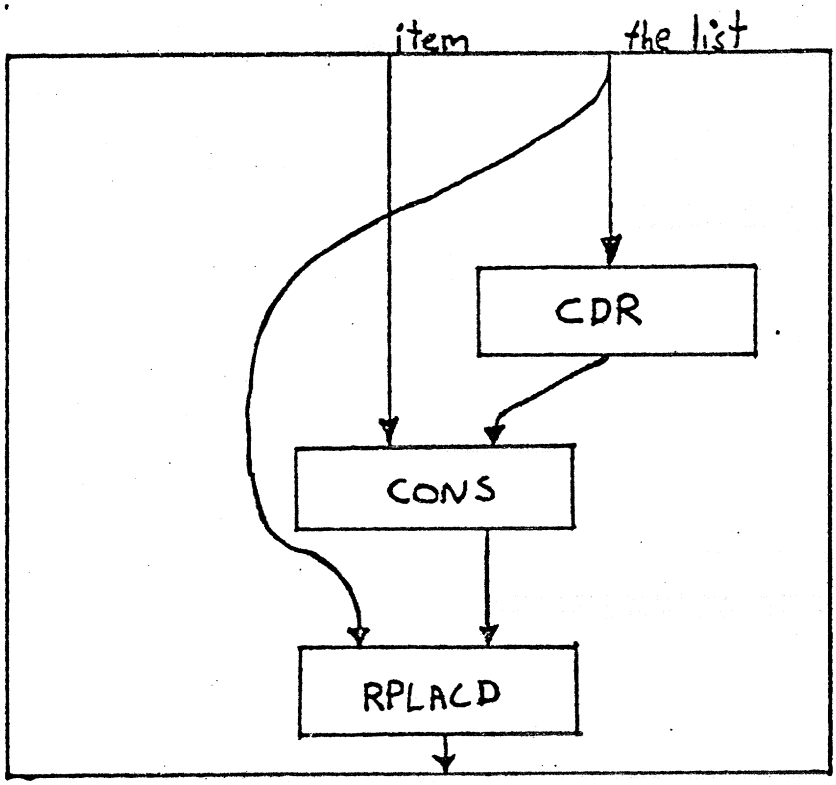
The generator in figure 15 exactly corresponds to the PLAN for the *trailing pointer enumeration* cliche. This cliche is a list enumeration that returns pointers to two successive subsets of a list. It requires `cdr` operations in both the *initialization* and *operation* roles of the generator, and it demands that the data flow line which is the second input of the generator body be the input to the initialization segment as well. These restrictions ensure that successive elements of the list are returned no matter how many times the body is executed.

`Events-queue-insert` also contains a *splice-in* operation which is trivially recognized in figure 15. The PLAN for a splice-in is shown in figure 16. (It does not correspond to a simple piece of code.) This operation is composed of a `cons`, a `cdr` and a `rplacd` function, where the `cons` creates an augmented list, and the `rplacd` attaches it to the end of the immediately preceding portion of the list. The PLAN representation for this algorithm requires that the second input to the `cons`, and the first input to the `rplacd` function start as a single data path. This path must be split by a `cdr` operation just prior to the `cons` and `rplacd` statements involved. In figure 15, the `cons` and `rplacd` operations are evident, while the role of the `cdr` function is fulfilled by the `cdr` in the initialization and the `cdr` in the body of the trailing pointer enumeration.

4.3 Extensions

The generality of the cliche finders could be extended by employing more powerful recognition techniques. The existing version of the system can use an exact match paradigm only because it deals with algorithms that are simple enough to be represented by a single canonical PLAN. As the size of the algorithm increases, the variability associated with its different implementations begins to show up in the PLAN, and the exact match paradigm eventually fails. For the recognition tasks involved in the scenario, this approach has been successful. However, it has not been thoroughly tested. The cliche finder currently contains two algorithm recognizers; one for the membership test and one for

Fig. 16. The PLAN for the splice-in operation



the non-header-cell insertion function used in the scenario.

My original intention was to write the algorithm recognizers as a composition of feature detectors for smaller cliches. The hope was that this more hierarchical design could be scaled up to identify larger functions. However, the logical analysis underlying PLANs actually does a poor job of localizing some cliches. For example, the splice-in function in figure 15 is spread across 4 different segment boundaries. The result was that a considerable amount of search was involved in finding such cliches. (This problem was the motivation for extracting features from `events-queue-insert` after the program was recognized as a whole. It turned out to be *easier* to identify the more complex entity first, and then pull out the meaningful sub-cliches.)

The process of recognizing an algorithm from its parts also has the problem that the interface between the sub-cliches in a PLAN can be complex. For example, the trailing pointer enumeration and the splice-in operation within `events-queue-insert` share substructure. In order to correlate these overlapping parts, more sophisticated data representations have to be involved. Rich [Rich 1980] develops a tool called *overlays* in his thesis which address this issue.

A generalized pattern matching facility for performing PLAN recognition would be the method of choice for identifying cliches. The creation of such a facility is a very difficult task, and it involves both computational and representational issues that are unsolved. It is well beyond the scope of the cliche finder as I envisioned it. Brotsky [to appear] is working on this topic for his Master's degree.

5. The sniffer system

The sniffer system provides a mechanism for representing knowledge about errors in code. It is organized as a collection of independent experts (called sniffers) which localize the information required to identify specific bugs. Each expert can use the facilities of both the time rover and the cliche finder to recognize its particular error. For example, the *cons bug* sniffer (which produced the bug report in the scenario) used the cliche finder to determine that `events-queue-insert` was a non-header-cell insertion, and it employed the time rover to identify the control paths taken during that function's evaluation. In addition, the cons bug sniffer found the values for data objects by causing the time rover to reexecute portions of the test program's code.

The sniffer system currently uses a simple control structure to chose the experts relevant to particular problems. It runs all of its sniffers all of the time, and each expert is designed to fail quickly when it does not apply to the task at hand. In the current version of Sniffer, there is exactly one expert (the one used in the scenario), although a number of extensions are planned. (See the section on future work for a discussion.) When the experts begin to share information, a more complex control strategy will be required.

5.1 A generic bug detector

Each expert in the sniffer system contains three basic parts; a collection of triggers which determine if the expert is relevant, a body, which recognizes an error, and a template report that produces output which describes the bug.

The triggers are filter functions which determine if a given expert should be tried. If they succeed, the body of the expert is executed, and if the body succeeds, the template output is displayed. Triggers are computationally inexpensive tests that fail if some essential feature is not present. For example, the trigger for the sniffer used in the scenario was the cliche finder responsible for identifying `events-queue-insert`. (Other cheaper triggers could also be employed. For example, the presence of keywords such as "member" or "insert" inside of function names within the user's code could cause specific bug experts to be applied.)

The body of an expert contains tests which recognize a particular error. These tests are not restricted in any way; the body can use both the time rover and the cliche finder to detect the critical

features which "implement" a given bug. For example, the body of the sniffer used in the scenario examined the control flow in `events-queue-insert`, the PLAN for that function and specific values of the `events-queue`. It also examined the PLAN and execution sequence in the caller of `events-queue-insert`, which was the function `metastasize`. Once the bug has been recognized, the body determines some additional context elements (such as the text for the programs involved, as opposed to their PLANs) and sends the results to the template report.

The template report mechanism produces the most comprehensive description of the bug which the sniffers can provide. Each template contains two sections; a summary of the error, and an analysis of the events surrounding the specific occurrence of the bug. The summary is a piece of canned text that uses a vocabulary which is justified by the examinations the experts perform. All the cliches it mentions are recognized by the sniffer body in the process of identifying the error. The analysis section explains how the test program acted on specific data values to produce the manifestation of the error observed. It provides the input and output values of procedures, and displays interesting intermediate results that were internal to specific cliches.

The sniffer system employs template reports in order to avoid the need for natural language generation facilities. Each template contains canned text interspersed with slots that are filled with data provided by the sniffers. In the output shown in the scenario (see page 18), the lower case information was produced by the template, and the data in upper case were the parameters which filled in the holes.

5.2 The Cons Bug Sniffer

In the scenario, the sniffer system was invoked by the expression

```
(get_expert_help '(events-queue-member events-queue new-cell)
                 (focus-time)
                 (end focus-time))
```

where the region enclosed by the two times encompassed a single execution of `events-queue-insert`. (The function `get-expert-help` runs all the bug experts, taking the union of their results.) The sniffer which produced the output shown on page 18 was called the *cons bug sniffer for sorted lists.*

The critical actions of the cons bug sniffer are summarized in figure 17. The trigger of the expert

was the cliche finder for identifying a non-header-cell insertion. It was applied to the PLAN for `events-queue-insert`. When this ran successfully, the sniffer body extracted the following features from that PLAN; the ordering predicate test,[1] the header-cell-insertion (which corresponds to the PLAN in figure 15), the splice-in operation, the `cons` function which was evaluated on exit from `events-queue-insert`, and the variables or code fragments which identified the item to be inserted and the queue. These features were identified by simple operation on PLANs. For example, the cons return fills an *action* role of the exclusive-or shown in figure 13. (See the discussion in the section on feature recognition in cliches.)

Fig. 17. The Cons Bug sniffer.

The cons bug sniffer is invoked with a user-supplied predicate describing the error, and a *region* of the test program's execution which specifies a particular piece of code. The following tests define the presence of the cons bug.

Triggers

* The PLAN for *region* must exactly match the PLAN for a non-header-cell-insertion

Body

* The header-cell-insertion, and the splice-in portion of *region* must *not* be executed between the two times.

* The ordering predicate test *was* executed.

* The insertion function returned by consing the item to be inserted onto the list.

* The value returned by the insertion function was not used (in the environment of its caller) to side-effect the list.

---

1. The ordering predicate was identified by the presence of an ordering test of the form (< a b) or (> a b). The enclosing usage of that predicate was ignored, e.g., (> a b), and (not (< a b)), etc., were judged equivalent.

With these features in hand, the cons bug sniffer proceeded to identify the critical events associated with its bug. These tests were principally involved with determining the control path actually taken through the `events-queue-insert`. First, the sniffer determined that the header-cell-insertion in the PLAN was *not* executed. This was accomplished by finding the code attached to the PLAN for that cliche, and submitting a request to the time rover (which was expected to fail) of the form

```
(future-when '(during code) focus-time (end focus-time))
```

This expression translates to the statement, "was this code executed between these two times". (The last two arguments are optional parameters which identify a region of the execution trace to examine.) In the case of the non-header-cell-insertion, there was no single piece of code associated with the entire cliche. The search was conducted for a piece of code attached to an internal segment of the PLAN which had to have been executed if the insertion occurred. The cons bug sniffer performed similar tests to establish that the ordering predicate was executed and that execution led to the `cons` return described above.

The final criteria for the cons bug requires that the list returned by the insertion function cannot be used to side effect the queue. This can be established in several ways. The most direct method is to use the time rover to examine the queue for side-effects. The cons bug sniffer accomplishes this by running the expression

```
(unmodified* (@ focus-time events-queue)
             (@ (end focus-time) events-queue))
```

If the predicate returns true, then the list held by the variable `events-queue` was not side-effected between the two times.

In the example of the cons bug shown in the scenario, the sniffer discovers the same fact (in a more informative way) by examining the PLAN for the function `metastasize`. This PLAN shows that there is no data flow coming from the return value of the insertion function. This can be seen in the body of `metastasize` (see page 18) by the fact that the code fragment

```
(events-queue-insert new-cell (+ div-time 2) events-queue)
(events-queue-insert key-cell (+ div-time 2) events-queue)
```

consists of two independent s-expressions. When the cons bug sniffer detected this information, it

produced the bug report statement

```
the function (defun metastasize ...) ignores the value returned by
events-queue-insert.
```

Once the cons bug sniffer established that the bug was present, it determined a number of specific data values to be used as context in the bug report. This information included the code for `events-queue-insert` and `metastasize` (obtained from an annotation on the top level segments in their PLANs), the value of the variable containing the events-queue within the insertion routine, and the value returned by `events-queue-insert`. Each of these data values was obtained by using the time rover to reexecute portions of the code.[1] For example, the value returned by `events-queue-insert` was duplicated by the request

```
(@ (future-when '(during '(cons entry evq)) focus-time)
   '(cons entry evq))
```

This expression searches forward from `focus-time` to the moment when `(cons entry evq)` was being evaluated, and executes that same expression in an alternate time track branching off from that moment. The results are necessarily equivalent.

The predicate, `(events-queue-member events-queue new-cell)`, which the user supplied to describe the bug, was not employed as a specification for the error. It was used only to provide contextual information for the bug report. (Specifically, if the PLAN for the predicate included a membership test, it was used to extract the variable name for the object which the membership test searched for. The user presumably wanted that object to be stored in the queue. This was the variable `new-cell` in the scenario.) In this particular case, a problem description was not required because the cons bug is essentially a violation of rational form in the domain of programming. It is rare to invoke any function for its side-effects when it does not always produce them, but in the case of a routine known to be a list insertion, the expectations associated with its use are much stronger.

There is an issue here relating to the breadth of knowledge in the bug experts. When the sniffers are attempting to recognize the code associated with an error, they know precisely what they are looking for. In this environment, an exact match paradigm is a reasonable method to employ.

---

1. The execution trace contains the values returned by all expressions executed by the test program, but they are not necessarily easy to identify as *return values*. The time rover records all side-effect events, but a given function can return any cell-id in the trace.

However, there are no constraints on the expression which the user types in to describe the bug. It could be a specific and useful definition of the error, or it might revolve on a fact in the application domain for the test program which would make little sense to the bug experts. This points out that the analysis applied to the user's predicate needs to be flexible. If the predicate cannot be totally recognized, then it can be parsed for features which could be used to select relevant bug experts. Alternatively, if the sniffer system grows considerably larger, the user's predicate could function as a source of hints for the direction which further analysis should pursue.

6. Future work

The top level goal of this research was to develop a system that understands (some) bugs. Sniffer has accomplished a portion of this task by demonstrating a deep understanding of one bug with what appears to be a general mechanism. The next step of the project involves proving that generality, and testing the power of Sniffer's expert system approach. To do this, I intend to expand the sniffer system by implementing a number of additional bug experts. These experts will cover a range of bug types, some related to the cons bug, and some concerned with more abstract programming cliches.

For example, a list data abstraction can contain a number of bugs which involve violated expectations about the maintenance of objects. If the *lookup* function believes there is a header cell, but the *insert* function does not, then any data item at the beginning of the list becomes invisible with respect to the *lookup* operation. If the *insertion* algorithm implements a bag which can contain multiple copies of an item, but the *delete* function removes them all, the user will perceive that inserted data spontaneously disappears. There are a number of similar errors of this type.

Another class of bugs detectors would deal with the interactions involving shared data. These bugs are particularly confusing to programmers (as they involve dynamic list structure and subtle interactions in code) but are ideal candidates for Sniffer because of the facilities provided by the time rover. A typical symptom of unexpected sharing is the unexplained appearance or destruction of data objects. A problem that comes from the absence of shared data is that expected side effects do not occur. Alternatively, items which are expected to be `eq` are not judged to be equivalent.

I suspect that there is also a set of bugs involving violations of rational form in the programming domain. These bugs could be catalogued by examining the expectations associated with specific programming cliches. The cons bug in the scenario is an example. Except for bizarre situations, any time a list insertion returns without side-effecting its input, something is likely to be wrong. (This is especially true when the user decides to complain about it.) In the case of a queue and process cliche (this corresponds to the control structure of *prosper*), it is reasonable to expect that the results of processing an item are inserted back into the queue. The bug in the scenario also relates to this cliche.

The power of the bug recognizers can also be demonstrated by expanding the amount of bug localization each sniffer performs. For example, the sniffer system could have been invoked at an earlier point in the scenario, when the bug had only been traced to the function `metastasize`. The

cons bug sniffer would then identify the presence of an insertion within `metastasize` and proceed to recognize the bug from there. There is also no reason why a sniffer cannot localize a bug to section of code and a region of execution that are completely different from the ones which the user initially provides. The support routines are present, what it requires is a more flexible method for directing a given sniffer. Each bug detector could presumably function by extracting hints from the user's error description, or directly from the user if that input was required.

Once a number of bug detectors have been written, I expect the research to proceed in the direction of formalizing the knowledge which was gained. At that point, it would become appropriate to rewrite the sniffer system to involve sharing of information between experts, a taxonomy of bugs, and perhaps a hierarchical understanding of cliches. Some of these developments rely on more powerful recognition techniques which are not yet available, although they are being developed at this time. (See [Brotsky, to appear].) One potential outcome of this research is an understanding of the constraints involved in the problem of error recognition, which is a step towards a theory of understanding bugs.

7. Related work

To my knowledge, no previous work has had the primary goal of generating a deep understanding of bugs in programs. The most closely related efforts are Hacker [Sussman 1973] and Ruth's thesis entitled "The analysis of algorithm implementations" [Ruth 1973]. (See [Lukey 1978] for a survey article describing work in this general field.)

Hacker is a system that designs and modifies programs to solve problems in the blocks-world domain. It employs an iterative approach. The system proposes a possibly buggy solution for a problem, runs the code, and analyzes any error which is produced. Hacker then applies a method for modifying the code that is believed to correct the error of the type discovered. If the new solution does not work, the process is repeated.

Hacker is primarily an effort in learning and automatic programming, as opposed to a thesis about debugging. (This is emphasized by Hacker's complete name, "A Computational Model of Skill Acquisition".) One of the system's major developments is that it explicitly represents knowledge about coding. There is no doubt that Hacker demonstrates a deep understanding of the programs it writes. It can notice when a program violates one of the subgoals of a blocks-world task, and it can use the information associated with this error to generate a complex program that avoids the bug. However, the process of bug classification is the least well-defined portion of the system.

Hacker gains a considerable amount of its leverage from the use of a toy domain which allows only a limited set of well understood operations. For example, each of the primitive blocks-world functions has a known purpose, which can be cited in the process of analyzing errors. The bugs which Hacker recognizes also involve constraints in this domain. For example, one of the errors discussed in Sussman's thesis involves two sub-goal problems which exactly undo one another's effects in the act of building a tower.

Sniffer applies similar domain constraints to generate its understanding of errors. In Sniffer's case however, the expertise lies in the domain of programming, and concerns the implementations and use of programming cliches. As a result of this approach, the system can recognize bugs in arbitrary programs, regardless of the tasks they perform.

Greg Ruth's dissertation describes a system that can recognize implementations for algorithms of a given class, and can also recognize buggy versions of those procedures. The system is based on a grammar which defines a class of programs. It inputs a grammar, and a function which it then

attempts to parse using that grammar. If it succeeds, the code is recognized as a member of the set of correct programs. Ruth extends the number of programs which can be identified by applying a collection of behavior preserving transformations to the code being analyzed. If the transformed function can be parsed by the grammar, it is also recognized as correct. Much of the system's knowledge concerns these rewriting rules.

In a similar way, Ruth's analyzer can identify errors. It does this by applying corrective transformations to the input code and then attempting to recognize the resulting routine. If the new function is within the set defined by the grammar, the error is analyzed as the inverse of the corrective transformation which was applied.

The kinds of bugs which Ruth's system can discover have a very syntactic feel. It treats programs as textual objects, without any detailed representation for their composition or the purpose of their parts. Sniffer, on the other hand, generates its power from an in depth analysis of the building blocks involved. The two programs also take fundamentally different approaches to the task of recognizing errors. Ruth's thesis diagnoses bugs as deviations from a predefined norm, whereas Sniffer searches for specific error-defining patterns. Sniffer uses this mechanism to represent extensive knowledge about particular bugs.

The programmer's apprentice project at MIT has produced a good deal of work in the domain of program understanding. Rich and Shrobe [1976] laid down the basics for the decomposition of Lisp programs into purposeful parts. Waters [1978] developed an analyzer which translates programs into PLANs. (This thesis relies heavily on the system which Waters implemented.) Rich's dissertation [Rich 1980] develops a mathematical foundation for the PLAN representation, and creates a library of PLANs for programming cliches. The complexity of the PLANs in this library range from the level of a variable interchange to the the queue and process strategy employed by *prosper*. (The library also includes the insertion plan discussed in the scenario.) Rich makes concrete suggestions for the construction of a PLAN recognition system which a more general version of the cliche finder would require. Brotsky [to appear] is working on this topic as the subject of his Master's thesis.

There has been some work towards an abstract theory of bugs in programs [Miller and Goldstein 1977] which goes beyond the domain dependent classifications developed in Hacker. The authors develop a planning grammar which can be used to describe programs, where statements in the grammar can be refined into runnable code. The authors then define a semantic error as a violation of the problem description, and a syntactic error as a bug in the use of the grammar. This grammar

does not have the conceptual richness of the PLAN representations used in the apprentice project, and the relation between their bug types and errors in more complicated programs is unclear.

There has been a considerable amount of work on expert systems (analogous to Sniffer) which perform complex tasks. That method of organization is one of the most successful paradigms in AI. The unifying characteristics of the systems which use this approach are that they rely on a number of independent methods for gathering information and they deal with a large number of facts in the process of finding solutions.

The Simulation and Evaluation of Chemical Synthesis project (SECS) [Wipke 1969], the Dendral project, and much of the work in AI and medicine are in this class. SECS is an expert in the design of organic syntheses. The information relevant to this task includes empirical facts about reaction conditions and the sensitivities of functional groups, the 3-dimensional shape of the target molecule (the one to be synthesized), the composition of the target, and electronic energy levels of both the product and the reactants. To coordinate these different sources of information, SECS confines a great deal of its expertise to a set of productions which examine these facts and determine if a given chemical reaction (applied to a particular molecule) will succeed or fail. The system performs at the level of a skilled chemist.

Sniffer also provides a tool for supporting the debugging process. There are two basically different approaches to this task. First, there are systems that simplify the process of tracking down bugs, and second, there are methods that prevent bugs from happening in the first place. The first category includes debugging environments similar to the one implemented on the Lisp Machine, which provide a single step evaluator and predicates for examining the data in the execution stack. The time rover is a straight forward extension of this environment. (Every major programming installation provides some support for activity of this kind.)

Bug prevention methods exist primarily in the domain of software engineering. Many of the ideas included under this term relate more to the process of coding than to the structure of the code which is produced. However, data abstraction techniques [Liskov 1977] are particularly relevant to the kinds of errors which Sniffer detects.

Data abstractions occupy the borderline between program understanding methods and programming language techniques, since abstraction mechanisms build the level of vocabulary used to discuss a program. Research in verification uses this fact, in that it tends to rely heavily on data abstractions as a place to attach restrictions about the properties for segments of code.

Data abstractions also imply a very strong form of type checking which makes certain kinds of errors much harder to commit. For example, the cons-bug error (which concerns the integrity of an object and the division of responsibility for maintaining its properties) can only be committed within the confines of a particular abstraction. These kinds of errors can not be totally avoided, but their frequency can be diminished by employing these techniques.

8. Bibliography

Brotsky, D.C., Program Understanding Through Cliche Recognition (M.S. proposal), to appear.

Corey, E.J. and Wipke, W.T. Computer Assisted Design of Complex Organic Syntheses, Science v.166 no.10, pp 178-192, (October 1969)

Liskov, B., Snyder,A., Atkinson,R. and Schaffert, C. Abstraction Mechanisms in Clu, MIT/LCS Computation Structures Group Memo 144-1, (January 1977)

Lukey, F., Features, AISB pp 10-14 (December 1978)

Miller, M.L., and Goldstein, I.P., Structured Planning and Debugging, IJCAI5, MIT,pp 773-779, (1977)

Rich, C., A Library of Plans with Application to Automated Analysis, Synthesis and Verification of Programs, MIT Ph.D. thesis, (1980)

Rich, C., and Shrobe, H.E., An Initial Report on a LISP Programmer's Apprentice, MIT/AI/TR-354 (1976)

Rich, C., and Shrobe, H.E., An Initial Report on a LISP Programmer's Apprentice (summary of TR-354), IEEE Trans. on Soft. Eng. V4 #5 (November 1978)

Rich, C., Shrobe, H.E., and Waters, R.C., Computer Aided Evolutionary Design for Software Engineering (NSF proposal), MIT/AIM-506 (1979)

Rich, C., Shrobe, H.E., and Waters, R.C., An Overview of the Programmer's Apprentice, IJCAI-79 (1979)

Ruth, G., Analysis of Algorithm Implementations, MAC/TR-130, (1973)

Shrobe, H.E., Waters, R.C., and Sussman, G.J., A Hypothetical Monolog Illustrating the Knowledge Underlying Program Analysis (Appendix to NSF proposal), MIT/AIM-507 (1979)

Sussman, G.J., A Computational Model of Skill Acquisition, MIT/AI/TR-297, (1973)

Waters, R.C., Automatic Analysis of the Logical Structure of Programs, MIT/AI/TR-492 (1978)

Waters, R.C., A Method for Analyzing Loop Programs (excerpt from thesis), IEEE Trans. on Soft. Eng., V5 #3 (May 1979)