CONTENTS

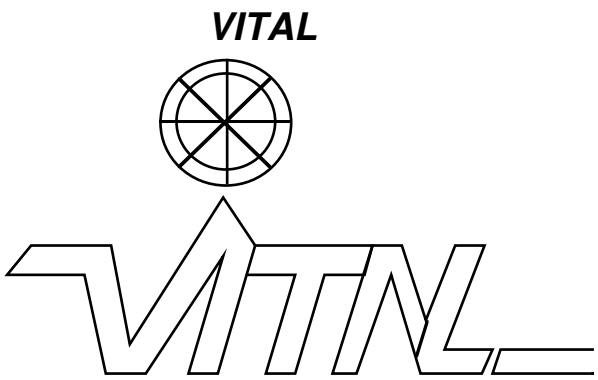
Acknowledgements.....1
Trademarks.....1
Preface.....1
Document Scope.....1
Publication of Work.....1
PART I REVIEWS AND PRELIMINARY INVESTIGATIONS.....2
1. Introduction.....2
1.1 Document Structure.....2
2. Related Work in Debugging KBS.....3
2.1 Verification.....3
2.2 Facilitating the Extension of Knowledge Bases.....4
2.3 KBS Debugging Tools.....5
3. Related Work in Automated Program Understanding and Debugging.....5
3.1 PLAN Diagram Code Representation.....6
3.2 Debugging Systems using PLAN Diagrams.....9
3.3 PROUST.....10
3.4 Program Slicing.....11
3.5 TALUS.....12
3.6 MRE.....13
4. A Fine Grained Study of a 'Friendly' Software Environment.....15
4.1 Background and Motivation.....15
4.2 The HyperTalk Debugging Environment.....16
4.3 Debugging Scenario 1.....16
4.4 Analysis of Problems Encountered in Scenario 1.....18
4.5 Analysis of Goal Decomposition for Scenario 1.....19
A. What the Programmer Really Wanted To Do:.....19
B. First Major Decomposition of Goals:.....19
C. Second Major Decomposition of Goals:.....20
D. Third Major Decomposition of Goals: Locating the Source Code....20
E. Fourth Major Decomposition of Goals: Access Privileges.....20
F. Fifth Major Decomposition of Goals: Performing the Action Steps.....20
4.8 Debugging Scenario 2.....21

4.9 Putting the Problems in Perspective....24
4.9.1 Representativeness....25
4.9.2 Solutions....25
5. An Investigation of Real Systems Debugging 27
5.1 Introduction....27
5.2 The Trawl 28
5.2.1 Raw Data....28
5.2.2 Condensed Data....29
5.3 Analysis of the Anecdotes....30
5.3.1 Dimensions of Analysis: Why Difficult, How Found, and Root Cause....30
5.3.2 Dimension 1: Why Difficult....32
5.3.2.1 Categories....32
5.3.2.2 Results....33
5.3.3 Dimension 2: How Found 34
5.3.3.1 Categories....34
5.3.3.2 Results....35
5.3.4 Dimension 3: Root Cause....35
5.3.4.1 Categories....35
5.3.4.2 Results....37
5.4 Relating the Dimensions 37
5.5 Discussion: Lessons Learned 39
5.5.1 Comparison with fine-Grained Study....39
5.6 Solutions....40

PART II THE VITAL BUG LOCATION METHODOLOGY 42
Preface 42
6. The VITAL Approach to the Solutions Proposed in the Studies 43
6.1 S2 and S3 43
6.2 S4, S5 and S8 43
6.3 S1=S6=S7 43
7. A Framework for Creating Software Visualization Systems 43
7.1 What is Software Visualization? 44
7.2 Classifying SV Systems 44
7.3 From Taxonomy to Framework and System: "what goes on?" 45
7.4 Programming Language Visualization vs Algorithm Animation 45
7.5 Viz Fundamentals 46
7.6 The Viz Architecture 47
7.6.1 Histories 48
7.6.2 Views 49
7.6.3 Mappings 51
7.6.4 Navigators 52
7.7 Examples Defined in Viz 53
7.7.1 A Prolog Visualizer 55
7.7.2 An OPS5 Visualizer 57
7.7.3 Additional Systems 59
7.8 A Comparison of Viz, BALSA, and TANGO Terminology 59
8. Bug Location Agents 60
8.1 Inter-Process Product Bug Location 61
9. Conclusions 68
References 70

ESPRIT PROJECT P5365
VITAL

Task 2.3.1 Bug Location Methodology



A methodology - based workbench for KBS Life cycle support

Document Reference:	OU/DD231/D1.0	Date: 1/4/93
Activity:	T2.3.1	Status: Internal Release
Distribution:	All VITAL Partners	
Title:	The VITAL Bug Location Methodology	
Abstract:	In this document we discuss the methodology for locating bugs within the VITAL workbench.	
Authors:	John Domingue, Marc Eisenstadt, and Blaine Price	

IMPORTANT NOTE: 1 **Internal Release**

The status of this document is Internal Release. That means that it has been reviewed and accepted by the internal QA procedure in VITAL but has not been accepted by the VITAL project as the final release of the deliverable. Therefore, the deliverable is meant for limited distribution only.

Internal release status granted by the QA reviewer - Kieron O'Hara - March 1993

IMPORTANT NOTE: 2 **Printing**

This document contains numerous colour screen snapshots and diagrams. In order for these to be legible this document MUST be printed with the **Colour/Greyscale** switch **ON**. This switch is a radio button in the print dialogue.

Collaborators (VITAL Partners):

SYSECA - SYSECA TEMPS REEL (Coordinator)
BULL - BULL CEDIAG
ONERA - ONERA
*OU - THE OPEN UNIVERSITY
UoH - THE UNIVERSITY OF HELSINKI
NOTT - UNIVERSITY OF NOTTINGHAM
AC - ANDERSEN CONSULTING
PTT - ROYAL PTT NEDERLAND NV
NOKIA - NOKIA RESEARCH CENTER

*marked partners are involved in this task

ACKNOWLEDGEMENTS

Apple Computer, Inc.'s Advanced Technology Group provided a supportive and productive environment during the summer of 1992, enabling Marc Eisenstadt to undertake the studies described in sections 4 and 5, along with other concomitant activities. Parts of this research were also funded by the UK SERC/ESRC/MRC Joint Council Initiative on Cognitive Science and Human Computer Interaction.

The rule base shown in section 8.1 was implemented by Enrico Motta and Mauro Gaspari.

Thanks go to Kieron O'Hara who did a fine job of QA'ing this document. All the remaining errors are ours.

TRADEMARKS

Symbolics is a registered trademark of Symbolics Inc. Sun is a registered trademark of Sun Microsystems Inc. LispWorks is a registered trademark of Harlequin Ltd.

PREFACE

This is the final deliverable for task T2.3.1 Bug Location Methodology. Because of its length we have split the document into two parts. The first part covers the reviews and preliminary investigations into finding bugs. The second part covers the VITAL Bug Location Methodology (VITAL-BLM).

Document Scope

The majority of KBS debugging work is based around verification and validation. As task T2.3.2 Verification and Validation (V & V) covers those areas we've concentrated here on *locating the root cause of bugs from an execution trace.* As task T2.3.2 is closely related to this task it is worth emphasising the different aims of each task. The final deliverable for Task T2.3.2 (together with task T1.2.1) is concerned with operationalising the VITAL approach to quality. Specifically, it reviews various approaches to V & V (including KADS and VALID) and then defines the VITAL V & V methodology.

Within this task we are concerned with how the Knowledge Engineer (KE) can be best aided when trying to move from the symptom of a bug to its root cause. By root cause we mean the specific part of a process product which is erroneous.

For this work we've used two main sources: existing work on understanding and debugging procedural (traditional) code, from an psychological and artificial intelligence point of view; and our own studies on how programmers debug programs.

Publication of Work

Section 4 has been submitted for publication to the British Human Computer Interaction Conference (HCI'93), section 5 has been submitted to the Fifth Workshop on the Empirical Studies of Programmers Workshop (V), December 3-5, 1993, in Palo Alto, California. Section 8 has been published in the Canadian Graphics Interface '92 Conference [Domingue *et al*, 1992] and in "User Centred Software Environments" [Domingue *et al*, in press].

© The Open University, 1993

PART I

REVIEWS AND PRELIMINARY INVESTIGATIONS

1. INTRODUCTION

Knowledge Engineers (KEs) will use the VITAL workbench to develop large Knowledge Bases. During the course of the development of a KB bugs will be produced. The task of locating bugs within large software modules has long been acknowledged as a difficult and time consuming chore. We will alleviate this, in the VITAL tradition, by providing methodologically based software support. Two future tasks (T3.4.1 Bug Location Support Prototype and T3.4.2 Visualization Tool) will provide the bug location software. This document describes the VITAL Bug Location Methodology (VITAL-BLM).

1.1 Document Structure

In the first part of this document we present reviews of relevant areas and two preliminary investigations into how bugs are located. The first (short) review covers KBS specific debugging work. This review is deliberately brief to avoid duplication of work with task T2.3.2 Verification and Validation. The second review covers work on understanding and debugging procedural languages. The techniques covered in the review are language independent and thus can be used within a KBS environment.

The fourth and fifth sections cover preliminary investigations into tracking bugs. These were carried out to provide a concrete grounding for our methodology. The first of these looks in detail at what it's like to work with an apparently 'modern' and 'friendly' environment: HyperCard. A detailed diary of several lengthy debugging sessions, was kept and then analysed. Eight fundamental problems were observed in the use of HyperTalk's debugging facilities: indirect access to troublesome source code; disruptive intermediate actions required; poor interpreter access during breaks; poor monitoring of built-in functions; no coarse-grained view of execution; no data flow analysis; no control flow analysis; deceptive view of inner states. The section discusses the broader implications of these eight problems, as well as possible ways to alleviate the problems.

The second investigation comprised of a world-wide trawl for debugging anecdotes from 'real programs'. This elicited replies from 78 respondents, including a number of implementors of well-known commercial software. The stories included descriptions of bugs, bug-fixing strategies, discourses on the philosophy of programming, and several highly amusing and informative reminiscences. Experiences included using a steel ruler to debug a COBOL line printer listing, browsing through a punched card deck to debug an early FORTRAN compiler, and struggling in vain to find intermittent bugs on popular commercial products. An analysis of the anecdotes reveals three primary dimensions of interest: *why the bugs were difficult* to find, *how* the bugs were found, and *root causes* of bugs. Half of the

© The Open University, 1993

difficulties arose from just two sources: (i) large temporal or spatial chasms between the root cause and the symptom, and (ii) bugs that rendered debugging tools inapplicable. Techniques for bug-finding were dominated by reports of data-gathering (e.g. print statements) and hand-simulation, which together accounted for almost 80% of the reported techniques. The two biggest causes of bugs were (i) memory overwrites and (ii) vendor-supplied hardware or software faults, which together accounted for more than 40% of the reported bugs. The section discusses the implications of these findings for the design of the VITAL-BLM.

The second part of this document contains the VITAL Bug Location Methodology. After describing how the VITAL-BLM will address the problems found in our studies we describe the VITAL-BLM in two parts. The first part describes a framework for describing and implementing software visualization systems. The second part shows how agents will be used to track bugs within and between the VITAL Process Products [Jonker *et al,* 1991].

2. RELATED WORK IN DEBUGGING KBS

In this section we briefly review debugging from a knowledge based systems point of view. As we say in the preface this review is brief because an extensive review of verification (and validation) techniques has already been carried out as part of task T2.3.2 Verification and Validation.

We first look at knowledge base verification techniques then we review work on facilitating the extension of knowledge bases. Finally, we review a debugging system which is part of a KBS development environment. We do not review KBS validation work because although related area to KBS debugging we consider it outside the scope of this work.

2.1 Verification

In [Adrion, Branstad, & Cherniavsky, 1982] verification is defined as “the demonstration of consistency, completeness, and correctness of the software at each stage and between each stage of the software development life-cycle”. Various techniques have been suggested for performing this task [Rousset, 1988]; [Perkins, Laffey, Pecora, & Nguyen, 1989]; [Reubenstein, 1985]; [Ayel, 1988]; [Evertsz, 1991]; [Evertsz & Motta, 1991]. CHECK [Nguyen *et al,* 1985] is a program that can find certain inconsistencies in a knowledge base. The categories of inconsistencies it can find are:

• Redundant rules - rules with equivalent LHS and one or more equivalent RHS clauses,

• Conflicting rules - rules with equivalent LHS but one or more RHS clauses are contradictory,

• Subsumed rules - rules with equivalent RHS clauses but one rule has fewer constraints and/or clauses in the LHS,

• Circular rules - a set of cyclic rules.

The analysis carried out by CHECK is not complete in the sense that it can only find certain classes of inconsistencies and redundancies. The KB-reducer [Ginsberg, 1988] uses assumption based truth maintenance techniques to perform a full analysis of a

© The Open University, 1993

KBS and is therefore ‘complete’. Like most ‘complete’ approaches however the KB-reducer embodies a number of assumptions which limit its effectiveness in analysing forward chaining production system interpreters. The assumptions are:

• *monotonicity* - new facts added to a knowledge base do not invalidate previously deduced conclusions, and assertions cannot be retracted,

• *no conflict resolution* - the interpreter does not perform conflict resolution.

A system which overcomes these limitations is Hybabs [Evertsz & Motta, 1991]. Hybabs is based on the *procedural semantics* of the underlying knowledge representation languages. Currently, Hybabs deals with a linked forward chaining production system and frame system. We shall first describe the functionality of the production system abstract interpreter. The algorithm for this is described in detail in [Evertsz, 1990]. Hybabs views a rule base and its inference engine as a partial function. The domain of the function is the set of possible inputs which the rule base can accept. The ‘range’ is the set of final databases which the rule base can generate from the input domain. In order to characterise the partial function Hybabs takes an abstract description of the input domain and generates a description of the set of final databases which can be generated, which is called the rule base's 'I/O mapping' (input-output mapping). In generating the I/O mapping, a full analysis is made of all possible routes through the rule base. This analysis enables Hybabs to identify many types of errors, for example, the rules which do not contribute to the output of the rule base. An example of an input specification and output description is given below:

Input Specification: (goal (assess-weight ?X)), (height ?X ?H),
where (c:primate ?X) ^ (c:integer ?H) ^ ?H>50 ^ ?H<100.

Positive Instance: (goal (assess-weight chimp)), (height chimp 76).

Negative Instance: (goal (assess-weight mandrill)), (height mandrill 27).

Output Description: (weight ?X (/ ?H 2)),
where (c:primate ?X) ^ (c:integer ?H) ^ ?H>75 ^ ?H<85.

Figure 1 – An input specification and output description in Hybabs.

The input specification contains two facts (predicates: goal and height); in addition there is a set of constraints on the variables which define the variables' domains (in these examples constraints are prefixed with ‘c:’ or are either ‘>‘ or ‘<‘).

2.2 Facilitating the extension of Knowledge Bases

There are many different aspects of a knowledge base that affect its ease of extension. In [Soloway, Bachant, & Jensen, 1987] two such aspects are described, *homogeneity* and *predictability.* Homogeneous code uses a small number of stereotypic language constructs called *plans* [Rich, 1981; Soloway & Ehrlich, 1984; Johnson & Soloway, 1985] to achieve desired goals. Predictable code is defined as having the following three properties: i) users know where to look for answers to questions, ii) users are not surprised by what they find, and iii) nothing untoward is carried out behind the scenes. Another aspect affecting ease of extension is *modularity*. A system is said to

© The Open University, 1993

be modular if the interactions between components are minimised and they use well specified protocols. This notion comes from traditional software engineering methodology and has long been advocated in AI programming as well [Winograd, 1975]. Production systems provide an example of a modular programming formalism, as interactions between procedures (rules) are forced through a very narrow channel (working memory). Further modularity can be achieved by clustering rules into groups, in order to limit the number of possible interactions. In [Jacob & Froscher, 1988] the authors describe a method for automatically partitioning a rulebase into groups in a way that achieves a high degree of modularity. Exceptional predicates that occur in more than one partition need to be documented by the programmer.

2.3 KBS Debugging Tools

Although, as described earlier, various techniques have been specified for the static verification of a knowledge base, less work seems to have been carried out in providing knowledge base debugging facilities. One notable exception is the Oregon Rule Based System (ORBS) [Fickas, 1987], a rapid prototyping environment for experienced rule based programmers. ORBS relies on an Interlisp style package, which allows programmers to insert breaks into their rulebase. ORBS also provides tools such as a micro matcher which displays a low level account of the generation of rule instantiations.

3. RELATED WORK IN AUTOMATED PROGRAM UNDERSTANDING AND DEBUGGING

Automatic program debuggers typically carry out the three distinct tasks shown in the figure below.

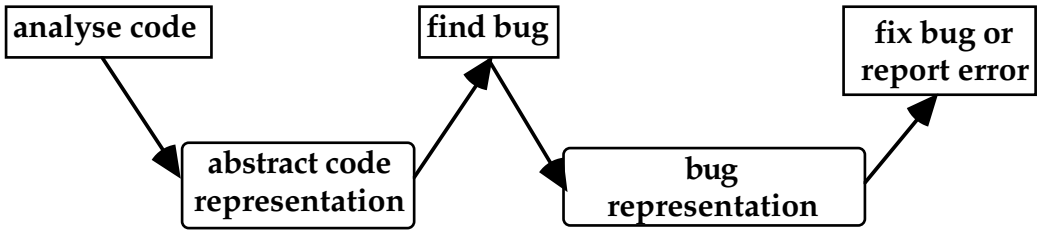


Figure 2. The broad tasks carried out by automatic debugging systems.

The code analysis phase produces an abstract code representation, typically a graph, which is then traversed to find the bug. Some debuggers will search for anomalies within the representation, whilst others compare the abstract code representation with some internally stored artefact.

Ruth [1976] used a program generating model to try and generate the programmer's program. The generator was a high level description of the correct program, and was only able to generate the programmer's code if the code was correct. If it could not do so it then tried to match on simple variations of the code. Variations might include code with the arms of conditionals swapped, or the signs in algebraic expressions switched. Adam and Laurent [1980] describe the program solution dynamically

© The Open University, 1993

using graphs; transformations of the graph are used to prove the equivalence of the programmer's program and the correct program. These transformations are similar to but more powerful than those used by Ruth [1976]. Any mismatches between the high level description and the programmer's code are taken to mean that there is a bug in the code.

Some debuggers use a specification given by the user as the artefact to be compared with the abstracted code representation. MYCROFT [Goldstein, 1975] was able to find errors in LOGO programs which drew shapes. The specification described the relationships between the components of the shapes drawn. The bug was then deemed to be in the section of code that constructed the part of the drawing that conflicted with the specification.

Code may also be understood by means of meta or symbolic evaluation. PHENARETE [Wertz, 1982] uses meta-evaluation to analyse the programmer's code. The main difference between evaluation and meta-evaluation is that in meta-evaluation every possible branch of the code is taken (as in Hybabs [Evertsz & Motta, 1991] see section 2.1). PHENARETE meta-evaluates the code, until every branch has either terminated, or has come to a repetition. This method can only find a certain class of errors. These errors are typically syntax errors, unreachable statements, endless recursion and non-terminating loops. The errors that are spotted are not deep semantic or conceptual errors; finding such errors requires knowledge about the actual task being attempted.

PTP [Eisenstadt, 1984] is a debugger which flags suspicious behaviours in the execution of Prolog programs. The suspicious behaviours are based on "footprints" (predefined execution paths) including specific types of failure such as failing to match the arity of any clauses or backtracking into a cut. As well as flagging suspicious behaviours PTP could be used as an interactive textual tracer.

One code abstraction representation which has been used by a number of the systems we review is the 'PLAN diagram' or PLAN formalism ([Rich, Shrobe, Waters, Sussman & Hewitt, 1978], [Waters 1978, 1979, 1982 & 1985], [Rich, 1981] and [Zelinka, 1986]) developed at the Massachusetts Institute of Technology (MIT). Before reviewing these systems we will first describe this formalism.

3.1 PLAN Diagram Code Representation

A PLAN diagram represents code segments as boxes, each box giving a specification for the code segment. Control flow and data flow are represented by hashed and solid lines respectively. So for example, the LISP code:

```
(cond ((< x y) x)
      (t (+ x y))
```

would be represented as figure 3.

© The Open University, 1993

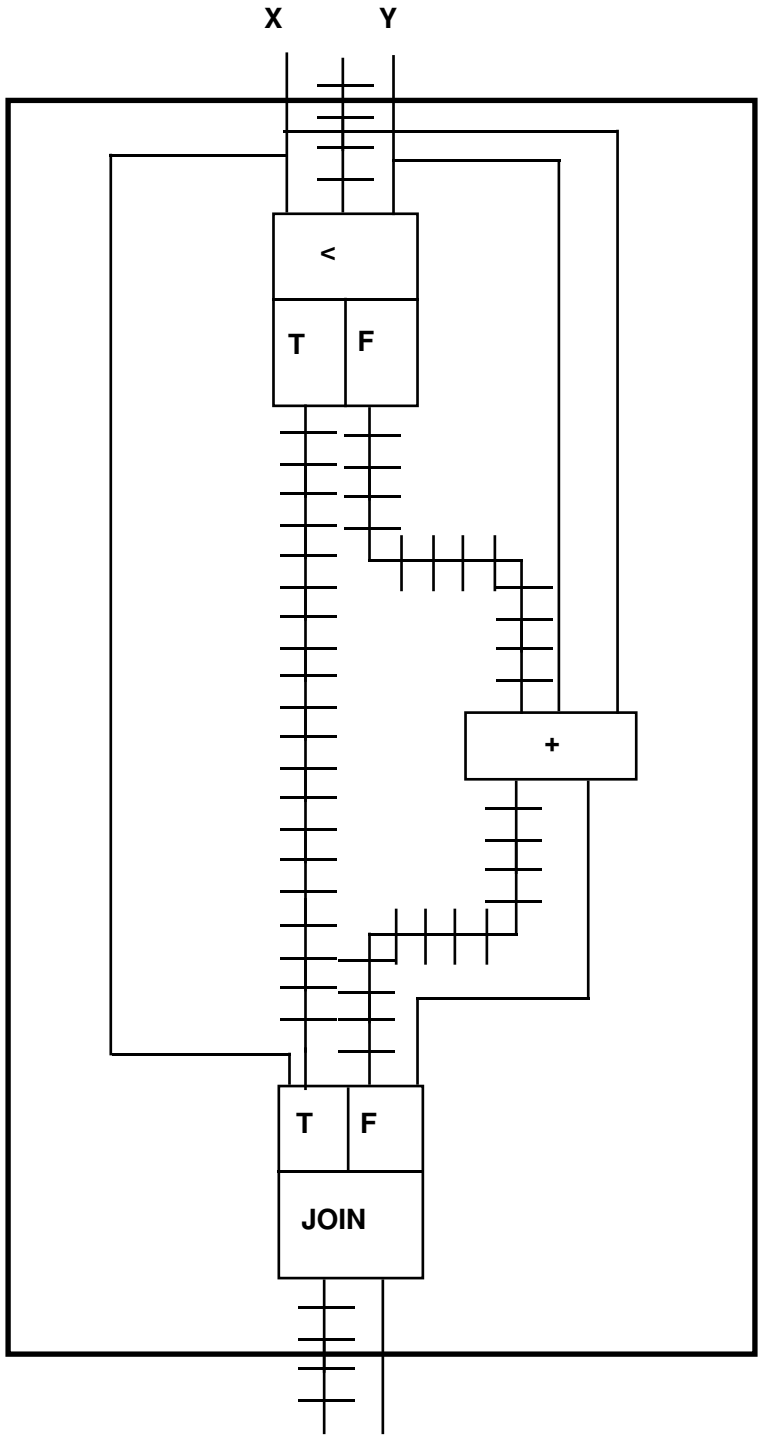


Figure 3. An Example of a PLAN Diagram.

The function + is shown as a box. The two arguments, x and y, are represented by the two solid arcs connected to the top of this box. The output of the function is represented by the solid arc coming out of the bottom of the box. The predicate < is represented by the box containing the symbols <, T and F. The two possible paths for control flow, after the predicate, are represented by the two hashed lines from the T and F sections of the box. The lowest box represents a join. A join specification is a mirror image of a predicate specification. Unlike the predicate specification,

© The Open University, 1993

however, the join does not represent any real computation. Joins are used to rejoin the two control-flow branches of a predicate block.

Recursion is represented as a looping line to the outside of the box. Figure 4 represents the (infinitely recursive) code:

```
(defun fib (n)
  (* n (fib (- n 1))))
```

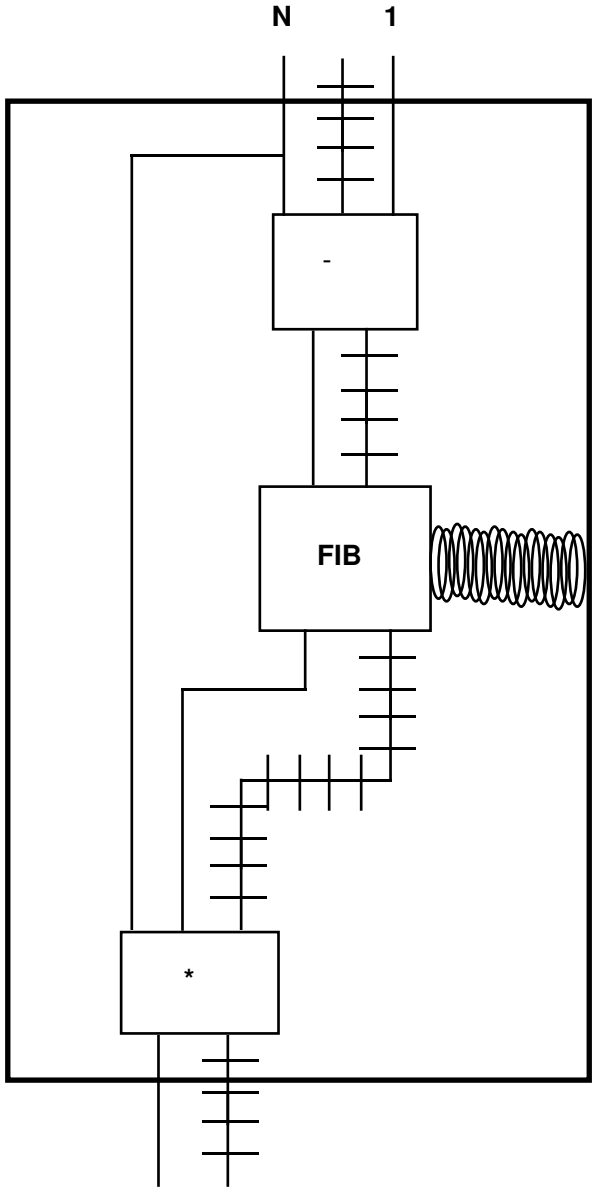


Figure 4. Representation of Recursion in PLAN Diagrams.

Iterative loops are converted into their tail recursive counterparts, and temporal decomposition [Waters, 1979] is then applied. Temporal decomposition is a technique for abstracting iterative loops or tail recursive functions. Each operation in the loop becomes a vector operation that acts on a vector of data objects. A vector of data objects is a vector where each element contains the values of all the variables of the loop for a particular iteration. The boxes in the PLANs can also be high level PLANs. This allows PLANs to be as abstract as need be.

© The Open University, 1993

3.2 Debugging Systems using PLAN Diagrams

Laubsch and Eisenstadt [1982] used temporal abstraction to analyse a subset of recursive SOLO programs written by novices.

The Recognizer [Zelinka, 1986] [Wills, 1990] is a system that performs program recognition by parsing. Programs are converted into a PLAN like graph representation. The library of structures to be recognised are translated into a graph grammar (currently performed by hand) and the program is parsed using the grammar. The graph parser is an extension of Brotsky's flow graph parser [Brotsky, 1981]. The extensions cope with some of the features of PLANs. Other features of PLANs, which cannot be dealt with by these extensions, have been transferred to attributes on the nodes and edges of the flow graph.

SNIFFER [Shapiro, 1981] uses a cliché finder, a time rover and sniffers to find bugs in a program's execution history. Each sniffer contains information about a particular type of bug. This information is represented by a set of rules. The program's execution history is recorded by a time rover. This stores all the intermediate states of variables and the effects of side effecting functions (enough information is stored to run a program backwards if required). A cliché is a recognisable algorithm fragment which may have many different implementations. The cliché finder identifies small algorithms by recognising patterns in a PLAN diagram representation of the code. Each sniffer uses the cliché finder and the time rover. A sniffer will use the time rover to obtain the value of a variable at different times during the evaluation. The recognition of typical algorithms by the cliché finder gives the sniffers a context for identifying errors, and raises the level at which SNIFFER can describe code. The architecture (taken from [Shapiro, 1981] p. 9) is shown below:

© The Open University, 1993

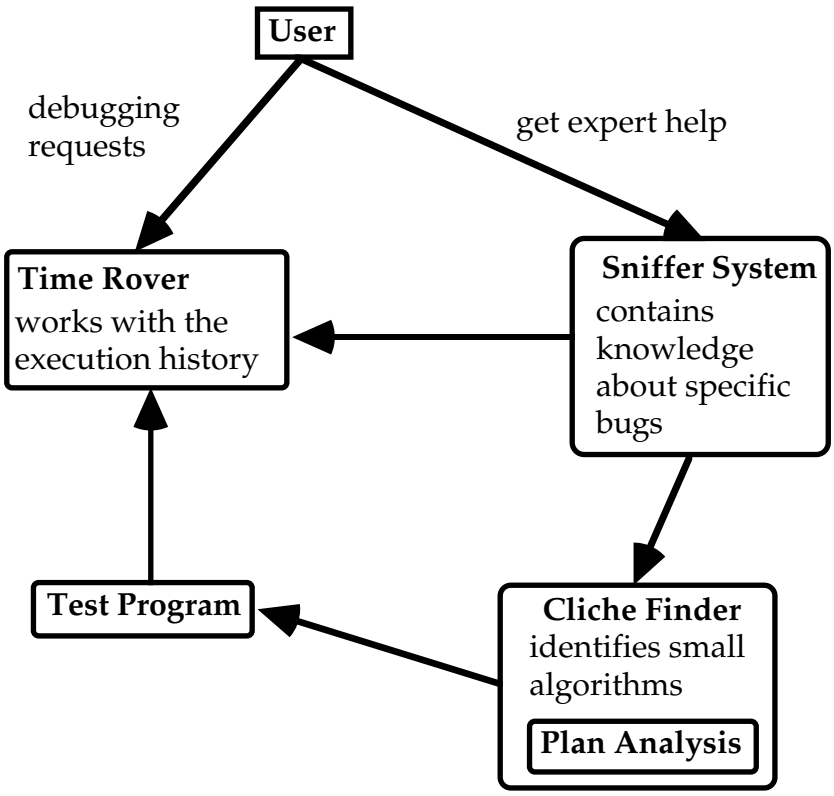


Figure 5. The architecture of the SNIFFER system.

ITSY [Domingue, 1987, 1992] is a system which finds bugs in novice Lisp programs. The programs are first transformed into a PLAN diagram form and which are then matched against *error clichés.* An error cliché is similar to a standard cliché except that instead of representing an algorithm fragment the cliché represents a typical novice bug.

We shall now describe four debugging systems in more detail.

3.3 PROUST

PROUST [Johnson, 1985] is an intention based PASCAL debugger. Johnson claims that debugging requires knowledge of the intentions of the programmer. Programming knowledge in PROUST is frame based and is contained in problem descriptions. Problem descriptions in PROUST consist of programming goals and sets of data objects. Programming goals are the principal requirements that must be satisfied and the sets of data objects are the data manipulated by the program. Data objects can either be constant-valued or variable-valued. Goal statements consist of a name of a type of goal followed by arguments. The problem descriptions describe what the programs must do but not how they are supposed to do it, these are described by plans.

PLANs [Waters, 1978] and the plans used in PROUST are similar but there is a subtle difference. The plans used in PROUST are derived from a psychological theory of programming plans being developed at Yale whereas PLANs are a program representation optimised for its utility for automatic systems. The main goal of

© The Open University, 1993

PLANs is to represent a program completely, making as much information as possible explicit.

Plans are stereotypic methods for implementing goals. Plans are compared to the programmer's program to determine which fits best. Plans contain a template slot which describe the form the PASCAL code should take. Plan templates consist of PASCAL statements, subgoals and labels. This representation is low-level and PASCAL dependent. Johnson's reason for this is [Johnson, 1985 pp. 85]:

> "If concrete plan and program representations are used, then some high-level errors are harder to identify, because the syntax gets in the way. If abstract representations are used, some low-level errors are impossible to identify, because relevant evidence has been abstracted away. Given the choice, a concrete representation must be used, since PROUST must be able to identify as wide a range of bugs possible."

Programmers' programs are parsed into a tree. PROUST selects, from the problem description, one goal at a time. The values of any data objects known at this point are substituted into the goal description. PROUST then tries to match each of the goal's plans in turn with the parse tree, using the plan's template slot. This is analysis by synthesis: PROUST generates possible implementations and matches these against the programmer's. If PROUST is unable to match a plan against the programmer's code then a bug is present. PROUST tries to interpret these plan differences using bug rules. Each bug rule has a test part which matches against the differences if the rule applies, and an action part which explains the plan differences.

3.4 Program Slicing

Weiser [1990] found that one of the techniques that programmers use when debugging their code is to work backwards from a line of code containing a variable, looking for the lines which contributed to the value of the variable. He termed this debugging technique *slicing*. A formal definition (actually 3 definitions) of a slice is [Weiser, 1990] p. 446, 447:

> "**Definition:** An elementary slicing criterion of a program *P* is a tuple <*i, V*>, where *i* denotes a specific statement in *P* and *V* is a subset of variables in *P*."
>
> "**Definition:** A slicing criterion is a set of elementary slicing criteria."
>
> "**Definition:** A slice *S* of a program *P* on a slicing criterion *C* is any executable program with the following two properties.
> (1) *S* can be obtained from *P* by deleting zero or more statements from *P*.
> (2) Whenever *P* halts on an input *I* with state trajectory *T'*, then *S* also halts on input *I* with state trajectory *T'*, and PROJ(T'), where PROJ is the projection function associated with criterion *C*."

The following example of slicing is given on p. 447

> "The original program
>
> ```
> 1 BEGIN
> 2 READ(X, Y)
> 3 TOTAL := 0.0
> 4 SUM := 0.0
> 5 IF X ≤ 1
> 6   THEN SUM := Y
> 7   ELSE BEGIN
> ```

© The Open University, 1993

```
8       READ(Z)
9       TOTAL := X*Y
10      END
11 WRITE(TOTAL, SUM)
12 END
```

Slice on the variable Z at statement 12.

```
BEGIN
READ(X, Y)
IF X ≤ 1
   THEN
   ELSE
       READ(Z)
END
```

Slice on the variable Z at statement 9.

```
BEGIN
READ(X, Y)
END
```

Slice on the variable TOTAL at statement 12.

```
BEGIN
READ(X, Y)
TOTAL := 0.0
IF X ≤ 1
   THEN
   ELSE TOTAL := X*Y
END
```

Slice on the variable Z at statement 9."

Although finding the minimal slice (a slice with the fewest lines of code) of an algorithm is uncomputable several algorithms exist for finding non-minimal slices [Weiser, 1990], [Horwitz *et al*, 1990].

3.5 TALUS

TALUS [Murray, 1986] is able to detect and correct errors at the algorithmic, functional and implementation level. TALUS takes a programmer's program and a reference program and tries to prove them equivalent using a theorem prover. TALUS is a debugger that works in a limited context and has eighteen task descriptions stored in a task library. Each task description has the following information:

a) The task assignment - instructions to the programmer,

b) Algorithms - identifiers naming acceptable algorithms for the solution of the task,

c) Algorithm Representations - frame representations of the above algorithms,

d) Reference Functions - functions that correctly implement the algorithm they are associated with.

Debugging takes place in four stages: program simplification, algorithm recognition, bug detection and bug correction.

© The Open University, 1993

TALUS uses a theorem prover to prove various conjectures involving Lisp code. Because the theorem prover can only deal with a subset of Lisp TALUS uses a sequence of program simplification transforms to reduce programmers solutions. These transforms eliminate CONDs, PROGs, lambdas and mapping functions.

The simplified code is parsed into frames. These frames are matched against the frame representations of the various algorithms stored in the task structure. A heuristic evaluation function computes a weighted sum of the frame slot differences. The algorithm with the highest score is chosen. This stage also pairs reference and programmer functions.

The equivalence of the reference and a programmer's program forms a conjecture. If the conjecture cannot be proved then the programmer's program is considered buggy. Conjectures are first checked by a conjecture disprover. This contains a pre-stored set of counter-examples. If a conjecture passes all the examples (which are in fact sets of bindings of formal variables for each function in a stored task algorithm) then it is matched against a reference function. Functions are represented as binary trees, the nonterminal nodes representing conditional tests. The collection of terms that must be true or false for a terminal node to be reached are the terms governing the node. Each set of terms governing a terminal node is a case. Each case of the programmer and reference code is compared by symbolic evaluation. A theorem prover is used to check the equivalence of symbolic values. If the programmer and reference values cannot be proved equivalent the programmer's program is considered buggy, the bug occurring in the case where the proof of equivalence breaks down. This means buggy implementations are always detected but some false alarms are generated.

Because TALUS uses a theorem prover which can only deal with a subset of Lisp, programmers' programs are first simplified. There are some constructs which TALUS cannot simplify however:

- Free variables in function definitions

- Side effects in conditional tests

- Side effects in the actual arguments of lambda expressions.

- Destructive functions such as NCONC. TALUS replaces NCONC with APPEND when the arguments are fresh list structures (that is lists that have are CONSed up within the program). When the arguments are not fresh list structures TALUS has to rely on heuristics.

3.6 MRE

MRE [Brayshaw, 1991] is a PARLOG (a parallel extension to Prolog) debugging system based on the concept that understanding the execution of a program is a task of manipulating and managing an information space (the execution trace of the program). Three methods of searching the information space for bugs are included. Each method consists of a set of agents, where each agent type has its own searching technique.

© The Open University, 1993

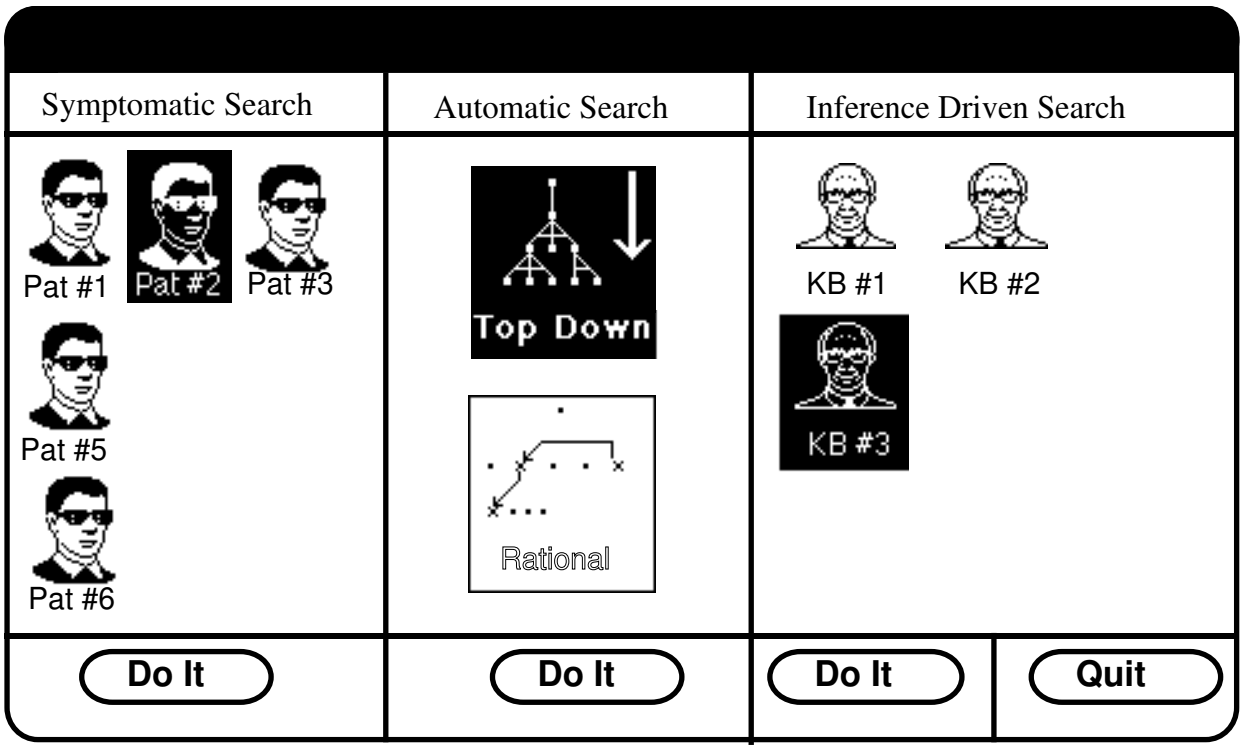


Figure 6. The debugging agent control panel in MRE.

The three types are:

• *symptomatic search* - symptom agents can be viewed as private investigators. Each will search for a symptom or set of symptoms within the information space. The user is able to define new symptom agents by selecting segments of the visualization as shown below:

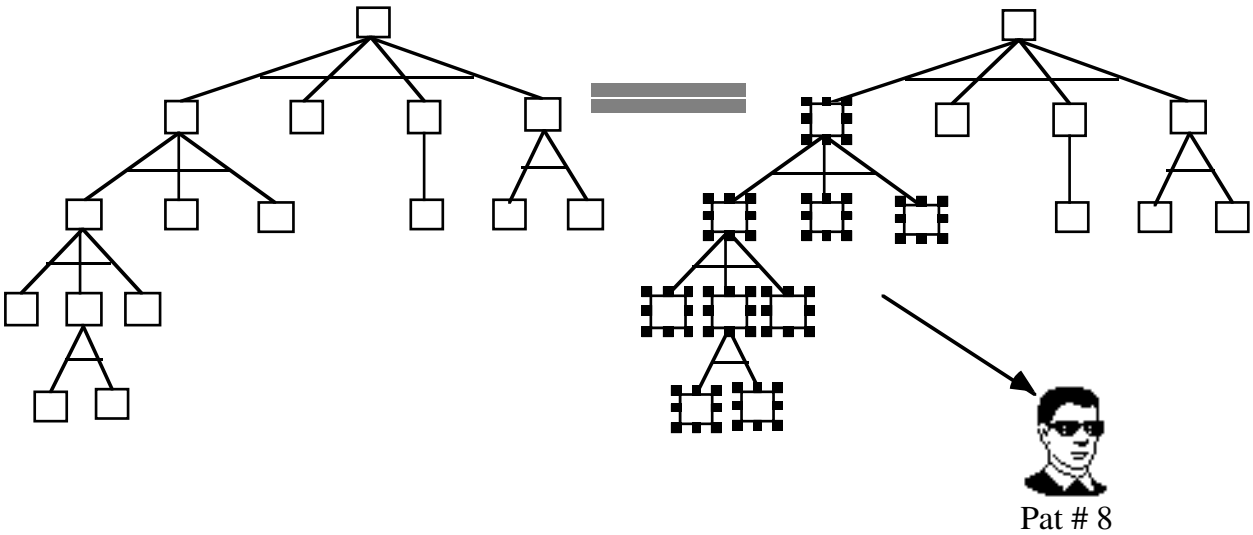


Figure 7. Specifying a symptomatic agent in MRE.

or by specifying a pattern in Prolog form.

© The Open University, 1993

• *diagnostic search* - these agents traverse the execution history asking the user whether intermediate parts of the trace are correct or not. The user's answer and the semantics of PARLOG determine the next point in the history. MRE contains several types of diagnostic agents, the searching pattern of the simplest is shown below:

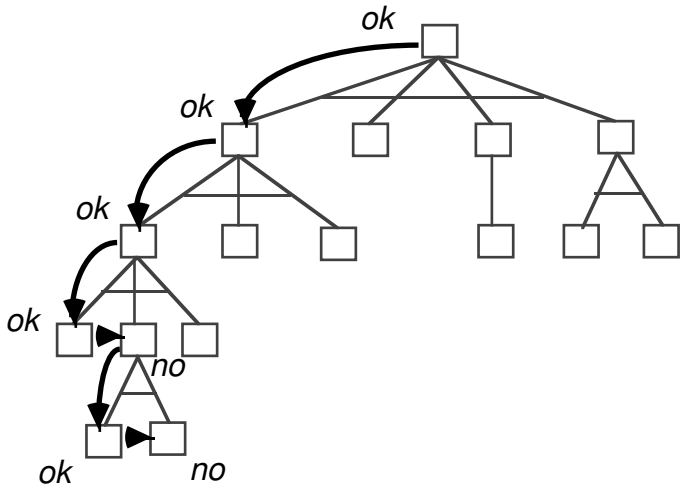


Figure 8. The searching pattern of the simplest diagnostic search agent in MRE.

• *inference driven search* - this is an extension to symptomatic search. Instead of searching for patterns within the execution history inference driven agents reason about the trace using a forward and backward chaining rule interpreter.

That concludes our review of debugging systems. We shall now describe the two studies we carried out to provide pragmatic grounding for the VITAL-BLM.

4. A FINE GRAINED STUDY OF A 'FRIENDLY' SOFTWARE ENVIRONMENT

4.1 Background and Motivation

In order to provide a solid grounding for the VITAL-BLM the self-report approach of [Knuth, 1989] was adopted in order to investigate some problems with existing debugging tools. In particular, we were interested in understanding in greater detail precisely what shortcomings exist in programming and debugging environments, even those which are considered quite 'modern' in design and conception.

The specific results reported here emanate from the work carried out by one of the authors as a Visiting Scholar at Apple Computer Inc.'s Advanced Technology Group in Cupertino, California, during the summer of 1992. As part of the investigations there, time was spent debugging HyperCard stacks and HyperTalk scripts, and keeping a detailed diary. The rest of this section reports the results, and attempts to place them in a broader context. In the interests of brevity, we have assumed that the reader is at least casually familiar with the Apple Macintosh, HyperCard, and HyperTalk programming/debugging. The psychological background for this work is presented in [Eisenstadt, 1993].

© The Open University, 1993

4.2 The HyperTalk Debugging Environment

HyperTalk scripting allows users easy access to the advanced user-interface features of the Apple Macintosh. For debugging purposes, the HyperTalk programmer has access to menu items which allow such things as setting of breakpoints and single-stepping. Separate windows can be enabled to provide variable watching and message monitoring (illustrated in Figures 9 and 10 below). However, on closer inspection these debugging facilities turn out to be superficial front ends to what is essentially a collection of teletype facilities. Debugging HyperTalk scripts turns out to be disproportionately difficult by comparison with the ease of use and expressiveness of HyperTalk itself.

As a way of elaborating on and understanding the nature of this apparent difficulty, the next sections of the paper work through some sessions involving real bugs. Along the way, the problematic nature of the HyperTalk-specific debugging process is analysed.

The debugging scenarios below are written from the point of view of the programmer/debugger (one of the authors), and consequently the first person singular is used.

4.3 Debugging Scenario 1

I was modifying an “Appointment Calendar” stack to add audio memos, when I encountered the following apparently simple bug: I couldn’t play an audio memo, because a dialogue box told me: “sound check ted cannot be played”. I had the following simple debugging scenario in mind: Stop execution just before & after that error message is displayed and inspect the state of relevant variables. This is a pretty routine goal for a user to want to achieve. I wasn’t sure what would happen after that, but I was more than happy to ‘wait and see’. My actual debugging session was a bit more indirect, however. To trace execution, I had to go through the following precise steps (described in the present tense to reflect the diary entry I was keeping while debugging):

1. bring up the message box
2. set the userlevel to 5
3. select the button tool
4. open (double-click) the sound button
5. open the sound button’s script (actually I combined 4 & 5 by holding down the shift key, but I still consider this to be two actions)
6. search for string “cannot be played”
7. read a little, try to make sense of surrounding code provided by the “Appointments with audio” stack implementors:

```
if the sound is not theSoundName then
  answer "Sound" && quote & theSoundName & quote && "cannot be played." --Δ
end if
```

8. set checkpoint on that line
9. (re)select troublesome card

© The Open University, 1993

10. (re)select the browse tool

11. click on troublesome button (which then executes right up to checkpointed line)

12. from debug menu, select “Variable Watcher” & “Message Watcher”

13. look (in cursory manner) at messages in message window:

```
mouseDown
  put
  put
  set
  get
  IsSoundButton
mouseUp
  put
  put
  SndExists
  play
  play
```

14. look at variable watcher (see figure 9 below):

(local to handler) mouseup

```
inRealm             true
theSoundName        check ted
```

15. I need to know why ‘the sound’ is not equal to ‘theSoundName’, since that is the only reason the error message should appear. So I need to find where ‘the sound’ gets set. Try to find string ‘the sound’, but I’m not allowed to search, because “can’t choose from HyperCard’s menus here”, i.e. while in the middle of a break, I can’t cruise around the script (automatically, anyway... I am allowed to scroll it)... damn.

16. Open stack “HyperTalk reference”, search for ‘sound’, brings up card describing sound (function) as follows:

```
the sound

Value returned: A text string equal
to the name of the sound resource currently playing (such as "boing")
or the string "done" if no sound is currently playing.

The sound function enables you to synchronize sounds with other actions,
because scripts continue to run while sounds are playing.
```

17. Look at the enclosed example(s) provided:

```
if the sound is not done then ...
wait until the sound is done
```

18. Bring up message window, type in **`the sound`**, observe output = **`done`**. Hmmm... this could lead to an interesting bug if the user chooses to label a sound “done”, because the error message (useless as it is) would then *fail* to pop up for the wrong reason!! I test this out by recording a new sound, saving it under the name “done”, and then trying to play it. Sure enough, it doesn’t play (since whatever my problem has been is still the case), AND the error message fails to pop up this time (because `the sound = theSoundName`, since both = `done`.

© The Open University, 1993

19. End of the road: I need to observe when, how, where, & why 'the sound' function returns 'done' rather than 'check ted', but I'm not allowed.

20. Solution: control panel volume was set to 0 [which causes the sound to return "done"]. Found this in other ways, notably by trying to play sounds directly from the sound recorder, at which point a more meaningful message is displayed ("volume must be greater than 0"), although it is necessary to set the volume in the control panel (rather than the sound editor) to cure the original bug.

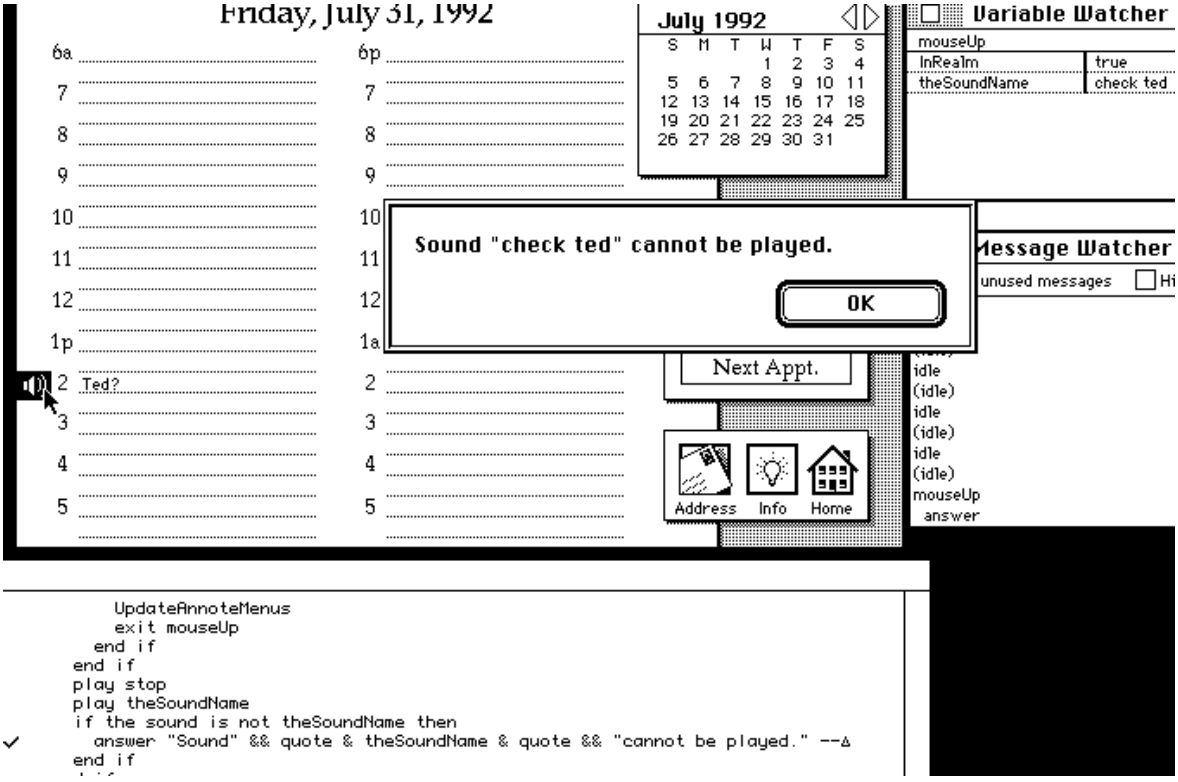


Figure 9: State of play at step 14 in scenario 1.

4.4 Analysis of Problems Encountered in Scenario 1

The above scenario is really indicative of old-fashioned debugging technology. This is even more remarkable when you consider how advanced HyperCard is meant to be. Other apparently less-advanced environments still allow the user to accomplish the same thing with a couple of break points and TTY print statements. Certainly, steps 1-11 are unacceptably cumbersome, because the goal to be achieved (stop execution just before & after that error message is displayed and inspect the state of relevant variables) is a standard programming/debugging cliché. Indeed, these steps ought to be condensed into a single operation or 'chunk'. Most of those steps actually achieve subgoals which are subservient to my main goal, and thus a lot of time is spent chasing subgoals, and sub-sub-goals, etc. These create diversions which are analogous to those you would encounter in the following (domestic) scenario: your main goal is to operate some battery-powered appliance/toy, but your batteries

© The Open University, 1993

are dead, so you decide to go the store to buy new batteries, so you hop in your car to go the store, but your car won't start, so you .... (etc. etc.), then you finally get to the store and find that you forgot your wallet, so you... (etc. etc.). A deep stack of diversionary subgoals is the hallmark of slapstick comedy, but of course this is extremely disruptive for human problem solvers. In a moment we take a look at the goal/subgoal hierarchy in detail (see the section "analysis of user's goal decomposition"). Generically, the problem can be summarised as follows: Performing a routine debugging/inspecting action involves dealing with many (disruptive) subgoals.

A specific problem with step 6 (searching for string "cannot be played") is that the user forced to do some detective work. That is, the link between an error message and the offending source code line has to be deduced by the user (whereas it could be trivially provided for free).

Step 13 is where an attempt to get a global overview of what's been happening made. But this is not easy. There is no meaningful coarse-grained view of execution.

At step 15, certain things need to be done while in the middle of a break, but access to the interpreter (and other features such as script searching) in the middle of a break is disallowed.

The problem at step 19 is that the behaviour of built-in functions (such as 'sound') can not be monitored easily.

To understand why the detailed performance of steps 1-11 is so disruptive, the next section works through those steps in some detail.

4.5 Analysis of Goal Decomposition for Scenario 1

Let's look again at steps 1-11 in terms of the goals they are trying to accomplish. The goal/subgoal decomposition of this debugging task is laid out hierarchically below. By the time of the final goal expansion, you'll see that there is a goal hierarchy of 28 nodes, nested no fewer than 6 layers deep. The goal expansion is best presented breadth-first to get a feel for what a pain it is to keep having to decompose the goal hierarchy. Note that the goals do not actually get decomposed in the order presented below... this is for expository purposes only.

A. What the programmer really wanted to do:

Top Goal(s): Stop execution just before & after that error message is displayed and inspect the state of relevant variables (and hope that the answer 'appears').

B. First major decomposition of goals:

Top Goal(s): Stop execution just before & after that error message is displayed and inspect the state of relevant variables (and hope that the answer 'appears').

1. Find where in the source code the error message is displayed
2. plant a breakpoint
3. re-run the code
4. inspect the variables

© The Open University, 1993

C. Second major decomposition of goals:

Top Goal(s): Stop execution just before & after that error message is displayed and inspect the state of relevant variables (and hope that the answer 'appears').

1. Find where in the source code the error message is displayed
   1.1. Locate source code
   1.2. Look for line containing (part of) the error message
2. plant a breakpoint
3. re-run the code
4. inspect the variables

D. Third major decomposition of goals: locating the source code

Top Goal(s): Stop execution just before & after that error message is displayed and inspect the state of relevant variables (and hope that the answer 'appears').

1. Find where in the source code the error message is displayed
   1.1. Locate source code
      1.1.1. Obtain access priveleges
      1.1.2. Get in correct ('open the hood') mode
      1.1.3. Invoke 'open script' sequence
   1.2. Look for line containing (part of) the error message
      1.2.1. eyeball small code or use Find command for large code
2. plant a breakpoint
3. re-run the code
4. inspect the variables

E. Fourth major decomposition of goals: access privileges

Top Goal(s): Stop execution just before & after that error message is displayed and inspect the state of relevant variables (and hope that the answer 'appears').

1. Find where in the source code the error message is displayed
   1.1. Locate source code
      1.1.1. Obtain access priveleges
         1.1.1.1. Invoke master 'monitor' through which things can be obtained
         1.1.1.2. recall the appropriate command
         1.1.1.3. enter the appropriate command
      1.1.2. Get in correct ('open the hood') mode
      1.1.3. Invoke 'open script' sequence
   1.2. Look for line containing (part of) the error message
      1.2.1. eyeball small code or use Find command for large code
2. plant a breakpoint
3. re-run the code
4. inspect the variables

F. Fifth major decomposition of goals: performing the action steps

Top Goal(s): Stop execution just before & after that error message is displayed and inspect the state of relevant variables (and hope that the answer 'appears').

1. Find where in the source code the error message is displayed
   1.1. Locate source code
      1.1.1. Obtain access priveleges

© The Open University, 1993

1.1.1.1. Invoke master 'monitor' through which things can be obtained
A1. bring up the message box
1.1.1.2. recall the appropriate command
1.1.1.3. enter the appropriate command
A2. set the userlevel to 5
1.1.2. Get in correct ('open the hood') mode
A3. select the button tool
1.1.3. Invoke 'open script' sequence
A4. open (double-click) the sound button
A5. open the sound button's script
1.2. Look for line containing (part of) the error message
1.2.1. eyeball small code or use Find command for large code
A6. search for string "cannot be played"
A7. read a little, try to make sense of surrounding code

2. plant a breakpoint
A8. set checkpoint on that line
3. re-run the code
3.1. restore 'up-front' environment
A9. (re)select troublesome card
3.2. restore 'behind-the-scenes' state
3.3. get in correct ('close the hood') mode
A10. (re)select the browse tool
3.4. invoke the code
A11. click on troublesome button
4. inspect the variables

Because this 28-node 6-layer hierarchy is an expansion of a debugging cliché, it represents something that really should be considered a simple atomic action. In reality, the user is forced through a large diversionary path along the way.

The next scenario studies problems with inspecting variables. Other scenarios have been omitted in the interests of brevity, but the main finding from the other studies was that the 'inner state' of an object can be deceptively different from its apparent state, requiring extra detective work to uncover. This problem is inherent in most programming tasks, especially since programmers have a tendency to 'hallucinate' or project what they think the state of an object or variable ought to be. Because HyperCard is so accessible and visible, this can be especially misleading. In one of the studies, a card field self-evidently contained a '1', yet it was not really a '1' at all... it only looked that way (actually it contained a '15', but the font size was too big and therefore the display was truncated) The moral of this story is that it would be useful to have a way to ensure that the 'true' (inner) state of an object be apparent upon request (as advocated by many researchers, e.g. [du Boulay, *et al*., 1981]).

4.8 Debugging scenario 2

Playing with the stack "Train Set", I found that the train mysteriously crashed on an apparently perfectly good section of track. I had the following (usual) debugging scenario in mind: Stop execution just before & after that error message is displayed and inspect the state of relevant variables. As before, my actual debugging session was a bit more indirect. To trace execution of the main handler (listed in Appendix A), I had to go through the following steps, which are deliberately enumerated in detail:

© The Open University, 1993

1. Search for occurrences of “crash”

2. Put checkpoints on all occurrences

3. Re-execute until trouble

4. Work out conditions prevailing when crash happens:
TheNextMove is empty AND BtnIconName is not “rotateTrain”...
so (assuming that “rotateTrain” is not relevant to my current problem), why on earth is TheNextMove empty?

5. Search for “into TheNextMove” to see when it’s set... found:
put item offset(Dir,Choices) of TheMoves into TheNextMove

6. Put checkpoint on this line (see figure 10 overleaf)

7. Re-execute until trouble

8. Enable watch variable window, notice values:

   Dir = “-”
   Choices = RLUD (Right, Left, Up, Down I suppose)
   so why is Dir = “-”

9. Search for “into Dir” to see when it’s set... found:

```
put char offset(Dir,"RLUD") of BtnIconName into Dir
```

10. Put checkpoint on this line

11. Re-execute until trouble

12. Notice values, both before & at moment of crash {window management already tricky... message & variable watcher block my text window; can’t type top-level ‘eval’ tests into message window while in middle of break; can’t search script while in middle of break}:

   normally (when on straight downward run),
   BtnIconName = --UD
   offset(D, RLUD) = 4,
   char 4 of BtnIconName is D
   counter gets incremented ok, so test for counter <> 1 is ok
   But, at moment before crash.. BtnIconName gets set to ----, [four hyphens]
   offset(D, “RLUD”) = 4,
   so char 4 of “----” is “-”, which clobbers Dir ,
   so who sets BtnIconName?

© The Open University, 1993

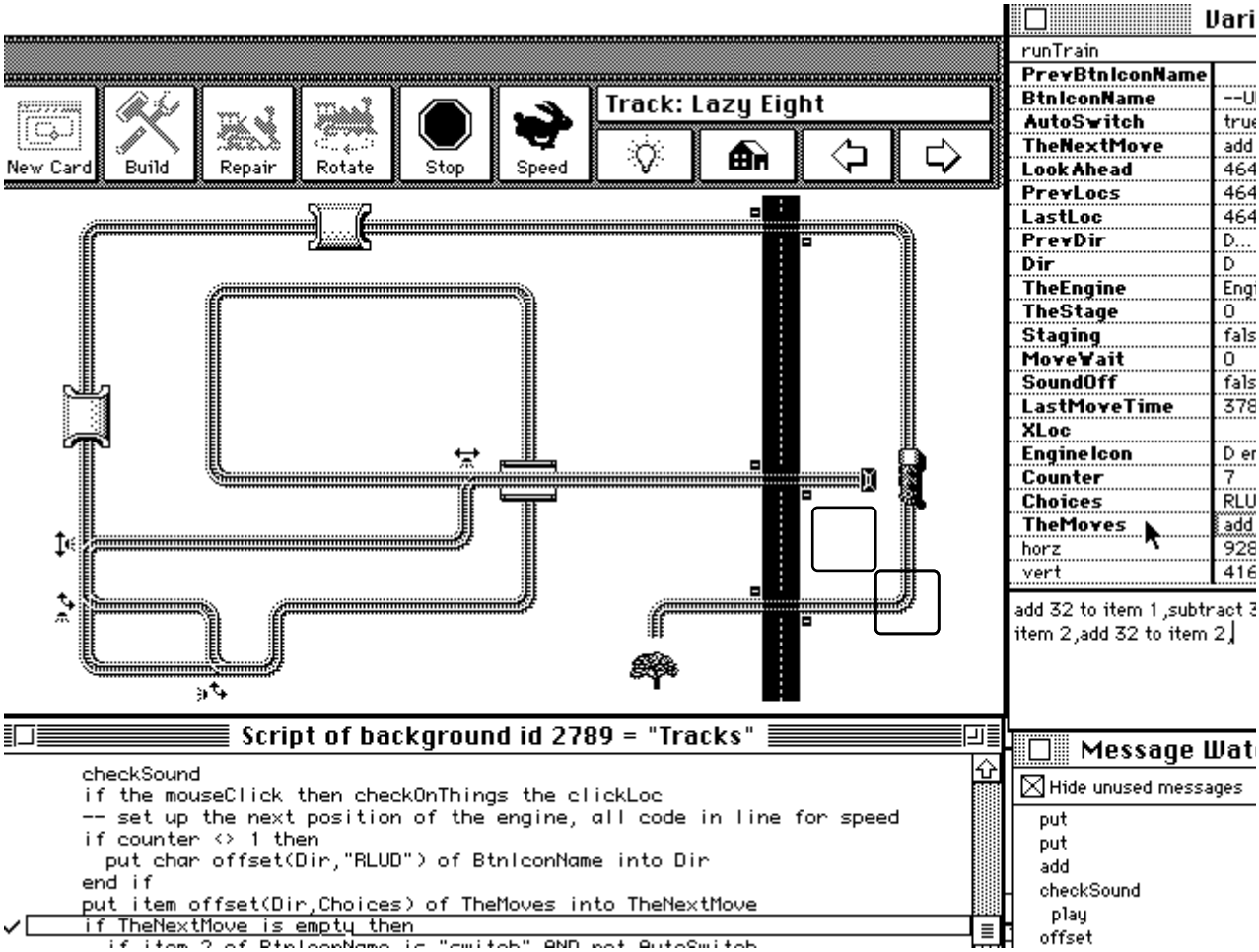


Figure 10. State of play at step 6 of scenario 2. The diagonally-adjacent rectangles beneath the train engine were not visible during normal running, but are shown here to indicate the locations of the problematic hidden buttons described in the text.

13. Search for "into BtnIconName" to see when it's set... found:

```
put iconName(icon of cd btn LookAhead) into BtnIconName
```

near the beginning of the main 'repeat' loop

14. Put checkpoint on this line

15. Re-execute until trouble

16. Step through, notice variable LookAhead is 464, 288 (which should be the next section of track, which itself is just a button)

17. Inspect the actual physical next section of track [i.e. the transparent button which displays itself using a track icon]

    it has id: 432, 256, with icon 4016 = "U--L"...
    although it is apparently located at position 464, 288
    [the ID is the eight-character string "432, 256", which is what the program actually uses, rather than its physical position, which is 464, 288]

18. Test out attributes of 'culprit' button in the message box...

    icon of cd btn LookAhead = 4025
    iconName(icon of cd btn LookAhead) = "----"

© The Open University, 1993

> hmmm... LookAhead seems to be returning the wrong thing, i.e. (432, 256) instead of (464, 288) alternatively, there's something weird about that button.. its ID is 432, 256... whereas all the others in that column were 464...

19. see how LookAhead is computed... seems to need delta 32 on X axis, 32 on Y axis,

20. search for "32"... not very informative...

21. search for "into LookAhead"... no luck

22. search for "of LookAhead"...aha.. found it:
`do (the Nextaction && "of LookAhead")`... does the arithmetic,

23. work out what happened to the physical button 464, 288... try "set the style of cd btn LookAhead to shadow" (this took a few diversionary experiments to work out that the key words needed to be "set the style...")

24. presto.. the button is in the wrong place!!! The two buttons labelled (464, 288) and (432,256) are diagonal neighbours which have been physically transposed [shown highlighted beneath the train engine in figure 10]. The section of track which appears to be "next" is not really next at all... it is empty, hence the train crash.

All of the problems of scenario 1 (and other scenarios not reported here) were present in scenario 2. The most overriding problem was that *traversal of indirect data influences (data flow) requires too much detective work.* The reasoning was isomorphic to that which is required to chase indirect pointers and other tricky references in C or Pascal.

To start with, locating "crash" was tedious because the *link between an error message and the offending source code line has to be deduced by the programmer (whereas it could be trivially provided for free).*

The detailed breakdown of the steps taken indicates again that *performing a routine debugging/inspecting action involves dealing with many (disruptive) subgoals.* The deductions needed at step 4 (working out prevailing conditions) suggest that *understanding the precise conditions under which a particular event happens (control flow logic) requires too much detective work.* The complaint at step 12 was that *access to the interpreter (and other features such as script searching) in the middle of a break is disallowed.* Repeated re-execution of the code was necessary because *there is no meaningful coarse-grained view of execution.*

The 'bottom line' was the moved card button, which was troublesome for the same underlying reason as that encountered in scenario 2: *the 'inner state' of an object can be deceptively different from its apparent state, requiring extra detective work to uncover.*

4.9 Putting the Problems in Perspective

Eight problems make the debugging scenarios much more troublesome than they need to be, given the powerful computing facilities available in the 1990's. The problems which emerged from the preceding analysis are as follows:

1. The link between an error message and the offending source code line has to be deduced by the user (whereas it could be trivially provided for free).

© The Open University, 1993

2. Performing a routine debugging/inspecting action involves dealing with many (disruptive) subgoals.

3. Access to the interpreter (and other features) in the middle of a break is disallowed.

4. The behaviour of built-in functions can not be monitored easily.

5. There is no meaningful coarse-grained view of execution.

6. Traversal of indirect data influences (data flow) requires too much detective work.

7. Understanding the precise conditions under which a particular event happens (control flow logic) requires too much detective work.

8. The 'inner state' of an object can be deceptively different from its apparent state, requiring extra detective work to uncover.

The following questions arise in relation to the above problems:

1. Are the problems reported here representative of problems encountered in other programming languages/environments

2. How can the problems be solved?

The next two sections address these questions in turn.

4.9.1 Representativeness

The question about representativeness is important, because it might just be the case that HyperTalk is particularly awful, or that the account is very idiosyncratic and doesn't touch on bigger issues. Far from being particularly awful, HyperTalk was selected specifically because it is widely hailed as being an easy-to-use programming environment "for the masses". The extensive range of public domain, shareware, and commercially available HyperCard stacks attests to its power and popularity. Thus, we considered it to be a strong a priori candidate for investigation, yet still felt personally that it exhibited some shortcomings which needed to be explored.

What about the relationship of the problems experienced in HyperTalk to those of other programming environments? There is strong preliminary evidence that these problems are representative. In the next section, we report on a survey of professional programmers we conducted, asking them to provide stories describing their most difficult bugs involving large pieces of software. The results of that survey tie in well with the reports here. We give a comparison of the two studies at the end of the next section.

4.9.2 Solutions

Regarding the solution of the problems, it is fair to say that at least some of the problems can by solved 'by fiat'. For example, problem 4 can be resolved simply by insisting that the behaviour (at least the output) of all built-in functions be monitorable.

One way or another, all of the problems mentioned in the introduction are connected with 'directness' and 'navigation'. For example, the need to go through indirect steps, intermediate subgoals or obtuse lines of reasoning plagues the user encountering problems 1, 2, 3, 6, 7, and 8. Table 1 shows the relationship between

© The Open University, 1993

the eight problems and either 'directness' or 'navigation', along with some suggested remedies:

HyperTalk Problem	'Directness'/'Navigation'	Solution
1. The link between an error message and the offending source code line has to be deduced by the user (whereas it could be trivially provided for free).	The connection is indirect rather than direct.	S1: Computable relations should be computed on request, rather than be deduced by the user.
2. Performing a routine debugging/inspecting action involves dealing with many (disruptive) subgoals.	Intermediate subgoals get in the way of directly attending to my main goal.	S2: Atomic user-goals should be mapped onto atomic actions.
3. Access to the interpreter (and other features) in the middle of a break is disallowed.	I can't do something right now, but must wait until the machine is in another state.	S3: Allow full functionality at all times.
4. The behaviour of built-in functions can not be monitored easily.	Involves both directness and navigation (because the user can't see what's happening at all).	S4: Viewers should be provided for 'players' (any evaluable expression) rather than just 'variables'.
5. There is no meaningful coarse-grained view of execution.	Navigation: user doesn't know where he/she is well enough to place meaningful break points or inspect meaningful data.	S5: Provide a variety of navigation tools at different levels of granularity.
6. Traversal of indirect data influences (data flow) requires too much detective work.	Chasing variable influences is very indirect.	S6=S1: Computable relations should be computed on request, rather than be deduced by the user.
7. Understanding the precise conditions under which a particular event happens (control flow logic) requires too much detective work.	Chasing control flow conditions is very indirect.	S7=S1: Computable relations should be computed on request, rather than be deduced by the user.

© The Open University, 1993

8. The 'inner state' of an object can be deceptively different from its apparent state, requiring extra detective work to uncover.	I can't directly perceive the true state of an object.	S8: Displayable states should be displayed on request, rather than be deduced by the user.

Table 1. The eight problems observed (left column), generalisations involving 'directness' or 'navigation' (centre column) and proposed solutions (right column).

The proposed solutions are not necessarily easy to implement, but there are an increasing number of tools appearing both in the research community and in the marketplace which illustrate aspects of these solutions. For example, consider the idea of computing and displaying important relations and states on request, rather than relying on the programmer's deductive skills (solutions S1=S6=S7 and S8). The software tool Purify [Hastings & Joyce, 1992] analyses run-time memory leaks in C programs on Sun workstations by patching the object code at link time, and pinpoints the root cause of the leak by traversing many indirect dataflow links back to the offending source code. Thus, it already solves a much harder dataflow traversal problem than that required to deal with indirect pointer traversing such as that illustrated in Scenario 2 above. Solution S3 (allowing full functionality at all times) is effectively provided in many modern Lisp implementations. Solutions S4 and S5 (viewers and granularity) have been explored at length in [Brayshaw & Eisenstadt, 1991] and [Eisenstadt *et al*, 1993]. That leaves S2 (atomic user-goals should be mapped onto atomic actions), which is increasingly addressed in commercial debugging tools, but still requires significant research input. We will discuss further how the above solutions will be implemented within VITAL at the start of Part II.

It is not the responsibility of HyperTalk's developers to solve all debugging environment problems, nor should it be. Rather, the purpose of this study has been to take an in-depth look at a number of outstanding issues, and to see what needs to be addressed within the VITAL Bug Location Methodology.

5. AN INVESTIGATION OF REAL SYSTEMS DEBUGGING

5.1 Introduction

The work reported in this section attempts to expand the single-user-log-book approach we adopted in the previous section to investigate the phenomenology of debugging across a large population of users. In particular, we were concerned about the problems of scalability: our work on debugging environments would remain vulnerable to criticisms of "toy examples only" unless an attempt was made to address the problems faced by professional programmers working on very large programming tasks.

Toward this end, we conducted a survey of professional programmers, asking them to provide stories describing their most difficult bugs involving large pieces of software. The survey was conducted by electronic mail and conferencing/bulletin board facilities with world-wide access (BIX, CompuServe, Internet Usenet, AppleLink). Our contribution is to gather, edit and annotate the stories, and to

© The Open University, 1993

categorise them in a way which may help to shed some light on the nature of the debugging enterprise. In particular, we look at the lessons learned from the stories, and discuss what they tell us about what is needed in the design of the VITAL-BLM. The discussion throughout concentrates on the relationships among three critical dimensions: (a) why the bugs were hard to find (c) how the bugs were found, and (c) the root causes of the bugs.

5.2 The Trawl

5.2.1 Raw Data

On 3 March, 1992, a request for debugging anecdotes was posted on an electronic bulletin board called BIX. BIX is The "BYTE Information Exchange" hosted on a computer network centred in Boston, Massachusetts, and accessed by approximately 60,000 users, mostly in the USA. BIX has a reputation for strong technical expertise, and it was for this reason that we chose BIX to start with. The original message is shown in figure 11.

```
c.language/tools #2842, from meisenstadt
771 chars, Tue Mar  3 06:09:28 1992
Comment(s).

TITLE: Trawl for debugging anecdotes (w/emphasis on tools side)...

I'm looking for some (serious) anecdotes describing debugging experiences. In
particular, I want to know about particularly thorny bugs in LARGE pieces of
software which caused you lots of headaches. It would be handy if the large
piece of software were written in C or C++, but this is not absolutely
essential. I'd like to know how you cracked the problem-- what
techniques/tools you used: did you 'home in' on the bug systematically, did
the solution suddenly come to you in your sleep, etc. A VERY brief stream-of-
consciousness reply (right now!) would be much better than a carefully-
worked-out story. I can then get back to you with further questions if
necessary.

Thanks!

-Marc
```

Figure 11. The original "trawl" request, posted on BIX. c.language is the name of the conference (major subject category), and "tools" is the name of the topic (minor interest sub-category). BIX has 60,000 members, hundreds of conferences and thousands of topics, each with thousands of messages and replies. A second message, explaining our motivation, was also posted.

During the ensuing months, similar messages were then posted to AppleLink, CompuServe, various Usenet newsgroups on Internet, and the Open University's own conferencing system (OU CoSy). The trawl request elicited replies from 78 "informants", mostly in the USA and UK. The group included implementors of very well-known commercial C compilers, members of the ANSI C++ definition group, and other known commercial software developers.

Figure 12 shows a typical reply to the original request. In that reply, the informant describes many important facets of his debugging experience: (a) the background context (MS-DOS 8086 compiler), (b) the symptom (wrong value on stack); (c) how

© The Open University, 1993

he approached the problem (single-stepping using assembly-level debugger); (d) why it was hard (the change happened more quickly at run-time than he could observe while single-stepping, plus he had the wrong model of which way the stacks grew, which made it harder to understand the behaviour); and (e) what the root cause of the problem was (the address *below* the stack pointer was being wiped out by the operating system interrupt handlers, and the pointer was being decremented too late in the compiled code).

```
I had a bug in a compiler for 8086's running MSDOS once that stands out in my
mind. The compiler returned function values on the stack and once in a while
such a value would be wrong. When I looked at the assembly code, all seemed
fine. The value was getting stored at the correct location of the stack.
When I stepped thru it in the assembly-level debugger and got to that store,
sure enough, the effective address was correct in the stack frame, and the
right value was in the register to be stored. Here's the weird thing --- when
I stepped through the store instruction the value on the stack didn't change.
It seems obvious in retrospect, but it took some hours for me to figure out
that the effective address was below the stack pointer (stacks grow down
here), and the stored value was being wiped out by os interrupt handlers
(that don't switch  stacks) about 18 times a second. The stack pointer was
being decremented too late in the compiled code.
```

Figure 12. A typical debugging anecdote. A representative sample of the raw data is presented in Appendix B.

A total of 110 messages were generated by 78 different informants. Of those, 50 informants specifically told a story about a nasty bug. A few informants provided several anecdotes, and in all a total of 59 bug anecdotes were collected. A second collection of anecdotes, which will be used subsequently for a top-down analysis by comparing it against the first collection, yielded an additional 58 messages from 47 more informants, of whom 45 told specific debugging stories. A few representative raw anecdotes are presented in Appendix B. The next sections presents a detailed summary of the raw data, followed by an analysis and discussion of the lessons learned.

5.2.2 Condensed Data

The stories and discussions which were posted on the various networks made fascinating reading, but they needed to be digested in some way to make them more meaningful and accessible. Summaries of the incoming data were entered into a large spreadsheet-cum-database, experimenting with the use of a variety of different data fields in an attempt to succinctly characterise the data. Table C-1 in Appendix C is an excerpt from that database, selected to show (a) the five fields (other than ID number) which emerged as persistent and relevant throughout the study, and (b) all 36 of the anecdotes which contained enough detail to yield an entry in each field. The relevant entry for the anecdote shown earlier in Figure 12 can be found in Table C-1 alongside ID “U6”.

The contents of Table C-1 is itself the result of several iterations of the analyses reported in the next sections. The reader may find it useful to browse the appendices before proceeding further in order to gain an overview of the style and content of the informants reports.

© The Open University, 1993

5.3 Analysis of the Anecdotes

5.3.1 Dimensions of Analysis: Why Difficult, How Found, and Root Cause

Although the "root cause" of reported bugs is of *a priori* interest, in order to fully characterise the phenomenology of the debugging experiences we needed to look at more than the causes of the bugs. In particular, it was necessary to say something about why a bug was hard to find (which might or might not be related to the underlying cause), and how it was found (which might or might not be related to the underlying cause and the reason for the difficulty). Thus, three dimensions became the critical focus for the ensuing analysis:

- *Why difficult:* This identifies the reason that the debugging experience itself was tricky or painful, e.g. perhaps the bug rendered the debugging tools unusable.
- *How found:* This identifies the diagnostic methodologies or techniques that were used in resolving the problem, e.g. single-stepping the code or inserting hand-tailored print statements.
- *Underlying cause of bug:* This identifies the root cause of the bug which, when fixed, means that the programmer has either totally solved the problem or else has gone far enough to regard the problem as being "in hand".

We know something about each of these dimensions from previous studies, although only Knuth's study really addresses the phenomenology of "debugging-in-the-large". Vesey [1989] attempted to address the first dimension (why difficult) by asking how the time to find a bug depended upon its location in a program's structure and its level in a propositional analysis of the program (answers: location in serial structure has no effect, and level in propositional structure is inconclusive). Regarding techniques for bug finding (second dimension), Katz and Anderson [1988] reported a variety of bug-location strategies among experienced Lisp subjects in a laboratory setting. In particular, they distinguished among (i) strategies which detected a heuristic mapping between a bug's manifestation and its origin, (ii) those which relied on a hand simulation of execution, and (iii) those which resorted to some kind of causal reasoning. Goal-driven reasoning (either heuristic mapping or causal reasoning) was predominant among subjects who were debugging their own code, whereas data-driven reasoning (typically hand simulation) was predominant among subjects who were debugging other programmers' code. In all cases bug fixing was not particularly problematic, once the bugs were located. However, that study involved programs which were typically about 10 lines long, and it is worth noting that a whole new set of problems arise for programming-in-the-large (the informants who mention program size typically speak of thousands of lines). In particular, the need for a bottom-up data gathering phase, which helps the programmer get some approximate notion of where the bug might be located, becomes apparent.

As far as root causes are concerned (dimension three), two main approaches to the development of bug taxonomies have been followed: a deep plan analysis approach (e.g. [Johnson, 1983]; [Spohrer *et al*., 1985] and a phenomenological account (e.g. [Knuth, 1989]; [du Boulay, 1986]). As discussed in section 3.3 Johnson worked on the premise that a large number of bugs could be accounted for by analysing the high level abstract *plans* underlying specific programs, and specifying both the possible

© The Open University, 1993

fates that a plan component could undergo (e.g. missing , spurious , misplaced) and the nature of the program constructs involved (e.g. inputs, outputs, initialisations, conditionals). Spohrer *et al* [1985)] refined this analysis by pointing out the critical nature of bug interdependencies and at problem-dependent goals and plans. An alternative characterisation of bugs was provided by Knuth's study. In particular, Knuth's analyses uncovered the following nine (problem-independent) categories: A= algorithm awry; B= blunder or botch; D= data structure debacle; F= forgotten function; L= Language liability, i.e. misuse or misunderstanding of the tools/language/hardware ("imperfectly knowing the tools"); M= Mismatch between modules ("imperfectly knowing the specifications", e.g. interface errors involving functions called with reversed args); R= Reinforcement of robustness (e.g. handling erroneous input); S= surprise scenario (bad bugs which forced design change, unforeseen interactions); T= Trivial typo.

For both approaches (plan analysis vs. phenomenological) the "true" cause of a bug can really only be resolved by the original programmer, because it is necessary to understand the programmer's state of mind at the time the bug was spawned in order to be able to assess the cause properly. For example, using the wrong variable could occur because the programmer really misunderstood the design of the algorithm, (i.e. he or she entered precisely the intended variable, but the intentions were mistaken, thereby falling into Knuth's category "A"), or it could be a lowly typographical error (Knuth's "T"). The manifestation is the same, but the root cause is different.

We found it informative to evolve our own categories in a largely bottom-up fashion after extensive inspection of the data, and then compare them specifically with the ones provided by Knuth. The comparison is provided in the far right hand column of Table C-1 (Appendix C) by means of a bracketed label such as "{A}" or "{L}" following relevant entries. In some cases, there is a straightforward mapping, but in others it is more subtle. For instance, there is sometimes a one-to-many mapping between our categories and those of Knuth. The reason is that Knuth knew his state of mind at the time he committed the errors reported in his log, whereas we have to rely on our interpretation of the programmer's anecdote. Even when the programmer's state of mind is clearly assessable, the assignment of blame can be awkward, because a given cause may always have an even deeper root cause. For example, the apparent root cause of a crash may be a memory address being overwritten by faulty data, and that faulty data itself may have been caused by a low level hardware fault (which itself may have a deeper cause relating to the time when coffee was spilled on the hardware in question, etc.).

The criterion we have adopted for identifying root causes is as follows: when the programmer is essentially satisfied that several hours or days of bewilderment have come to an end once a particular culprit is identified, then that culprit is the root cause, even when deeper causes can be found. We have adopted this approach (a) because a possible infinite regress is prevented, (b) because it is consistent with our emphasis on the phenomenology of debugging, i.e. what is apparently taking place as far as the front-line programmer is concerned, (c) it enables us to concentrate on what the programmers *reported*, and not try to second-guess them. In practice, the main consequence of this is that the category "memory clobbered" is identified as the root cause of numerous other problems, even though the clobbering may itself have been caused by, say, a faulty array declaration. If the faulty declaration is

© The Open University, 1993

reported in the anecdote, then we select that as the root cause, but if some variation of memory clobbering is deemed to be the culprit, then we accept that as the root cause.

The subsections which follow describe the three dimensions of analysis (why difficult; how found; root cause) in turn.

5.3.2 Dimension 1: Why Difficult

5.3.2.1 Categories

The reasons that the bug was hard to trap fell into five categories, as described below:

- **cause/effect chasm:** Often the symptom is far removed in space and/or time from the root cause, and this can make the cause hard to detect. Specific instances can involve *timing* or *synchronisation* problems, bugs which are *intermittent*, *inconsistent*, or *infrequent,* and bugs which materialise "far away" (e.g. thousands of iterations) from the actual place they are spawned. In this general category we have also included debugging episodes in which there are *too many degrees of freedom*. As an example of such an episode, consider the case in which a piece of software which works perfectly in one environment, yet fails to work in another environment. If many things have changed (e.g. different hardware, different compiler, different linker), then there are simply too many degrees of freedom to enable systematic testing under controlled conditions to isolate the bug. Such testing can be done, given enough time and resources, but it is difficult—hence this category.

- **tools inapplicable or hampered:** Most programmers have encountered so-called "Heisenbugs", named after the Heisenberg uncertainty principle in physics: the bug goes away when you switch on the debugging tools! Other variations within this category are: *long run to replicate* (i.e. the bug takes a long time to replicate on a fresh execution, so if switching on the debugging tool significantly slows down execution, then it can not really be used); *stealth bug* (i.e. the error itself consumes the evidence that you need to find the bug, or even clobbers the debugging tool); *context precludes* (i.e. some configuration or memory constraints make it impractical or impossible to use the debugging tool).

- **WYSIPIG (What you see is probably illusory, guv'nor):** We have coined this expression to reflect the cases in which the programmer stares at something which simply is not there, or is dramatically different from what it appears to be. This can range from syntactic problems (e.g. hallucinating a key word in the code which is actually not there) to run-time observations (looking at a value on the screen which is displayed in a different way, e.g. "10" in an octal display being misinterpreted as meaning 7+3 rather than 7+1). Such observations lead the programmer on a wild-goose chase, and can be the reason why certain otherwise simple bugs take a long time to track down.

- **faulty assumption/model or mis-directed blame:** If you think that stacks grow up rather than down (as did the informant in Figure 12), then bugs which are related to this behaviour are going to be hard to detect. Equally, if you bet your life on the known correct behaviour of a certain function (which is actually faulty), you are going to spend a lot of time looking in the wrong place.

© The Open University, 1993

- **spaghetti (unstructured) code:** Informants sometimes reported that the code "was in a mess" when they were called in to deal with it. There is, unsurprisingly, a 100% correlation between complaints about "ugly" code and the assertions that "someone else" wrote the code.

5.3.2.2 Results

In this and subsequent sections, we report the frequency of occurrence of the different categories, not because it supports an *a priori* hypothesis at some level of statistical significance, but rather because it gives us a convenient overview of the nature of the problems that the informants chose to share with us. Knuth argues that the categories and "raw records" themselves are just as informative as summary statistics (e.g. showing the frequency of occurrence of particular error types): "The concept of scale cannot easily be communicated by means of numerical data alone; I believe that a detailed list gives important insights that cannot be gained from statistical summaries." In a similar vein, we are mainly interested in categorising the findings, but feel that a count of frequency of occurrence is of interest as a reflection of what types of problems the population of informants chose to report.

The frequency of occurrence of the different reasons for having difficulty is shown in Table 2.

Category	Occurrences
cause/effect chasm	15
tools inapplicable or hampered	12
WYSIPIG: What you see is probably illusory, guv'nor	7
faulty assumption/model or mis-directed blame	6
spaghetti (unstructured) code	3
??? (no information)	*8*

Table 2. Why the bugs were difficult to track down.

Thus, 53% of the difficulties are attributable to just two sources: (i) large temporal or spatial chasms between the root cause and the symptom, and (ii) bugs that rendered debugging tools inapplicable. The high frequency of reports of cause/effect chasms accords well with the analyses of [Vesey, 1989] and [Pennington, 1987] which argue that the programmer must form a robust mental model of correct program behaviour in order to detect bugs—the cause/effect chasm seriously undermines the programmer's efforts to construct a robust mental model. The relationship of this finding to the analysis of the other dimensions is reported below.

© The Open University, 1993

5.3.3 Dimension 2: How Found

5.3.3.1 Categories

The informants reported four major bug-catching techniques, as follows:

- **gather data:** This category refers to cases in which the informant decided to "find out more", say by planting print statements or breakpoints. In fact, a variety of specific data collection techniques were reported. The important thing that distinguishes these data collection techniques from the "controlled experiments" category described below is their bottom-up nature. That is, the informants may have had a rough idea of what they were looking for, but were not explicitly testing any hypotheses in a systematic way. Notice that this category is different both from Katz and Anderson's, [1987] causal reasoning and from their data-driven hand simulation: it is really a hybrid of the two, because it is bottom-up, on the one hand, yet can lead directly to a causal analysis once the data has been gathered. Here are the six sub-categories reported by the informants:
    - *step & study:* the programmer single-steps through the code, and studies the behaviour, typically monitoring changes to data structures,
    - *wrap & profile:* tailor-made performance, metric, or other profiling information is collected by "wrapping" (enclosing) a suspect function inside a one-off variant of that function which calls (say) a timer or data-structure printout both before and after the suspect function,
    - *print & peruse:* print statements are inserted at particular points in the code, and their output is observed during subsequent runs of the program,
    - *dump & diff:* either a true core dump or else some variation (e.g. voluminous output of print statements) is saved to two text files corresponding to two different execution runs; the two files are then compared using a source-compare ("diff") utility, which highlights the difference between the two execution runs,
    - *conditional break & inspect:* a breakpoint is inserted into the code, typically triggered by some specific behaviour; data values are then inspected to determine what is happening,
    - *specialist profile tool (MEM or Heap Scramble):* there are several off-the-shelf tools which detect memory leaks and corrupt or illegal memory references, and the experts who relied on these also tended to rave about their value.
- **"inspeculation":** This name is meant to be a hybrid of "inspection" (code inspection), "simulation" (hand-simulation), and "speculation", which were among a wide variety of techniques mentioned explicitly or implicitly by informants: cogitation, meditation, observation, inspection, contemplation, hand-simulation, gestation, rumination, dedication, and inspiration. In other words, they either go away and think about something else for a while, or else spend a lot of time reading through the code and thinking about it, possibly hand-simulating an execution run. "Articulation" (explaining to someone else how the code works) also fits here. The point is that this family of techniques does not involve any experimentation or data gathering, but rather involves "thinking about" the code.

© The Open University, 1993

- **expert recognised cliché:** These are cases where the programmer called upon a cohort, and the cohort was able to spot the bug relatively simply. This recognition corresponds to the heuristic mapping observed by Katz and Anderson. The very nature of our data gathering exercise invariably requires a cohort (rather than the informant) to have detect the bug: the informant would not have been stumped, nor would have bothered telling us about the bug, had he or she been able to identify the solution quickly in the first place!
- **controlled experiments:** Informants resorted to specific controlled experiments when they had a clear idea about what the root cause of the bug might be.

5.3.3.2 Results

The frequency of occurrence of the different debugging techniques is shown in Table 3.

Category	Occurrences
gather data	27
inspeculation	13
expert recognised cliché	5
controlled experiments	4
??? (no information)	*2*

Table 3. Techniques used to track down the bugs.

Techniques for bug-finding are clearly dominated by reports of data-gathering (e.g. print statements) and hand-simulation, which together account for 78% of the reported techniques, and highlight the kind of "groping" that the programmer is reduced to in difficult debugging situations. Let's now turn to an analysis of the root causes of the bugs before we go on to see how the different dimensions interrelate.

5.3.4 Dimension 3: Root Cause

5.3.4.1 Categories

The bug causes reported by the informants fell into the following nine categories:

- **mem:** Memory clobbered or used up. This cause has a variety of manifestations (e.g. overwriting a reserved portion of memory, and thereby causing the system to crash) and may even have deeper causes (e.g. array subscript out of bounds), yet is often singled out by the informants as being the source of the difficulty. Knuth has an analogous category, which he calls "D = Data structure debacle".
- **vendor:** Vendor's problem (hardware or software). Some informants report buggy compilers or faulty logic boards, for which they either need to develop a workaround or else wait for the vendor to provide corrective measures.
- **des.logic:** Unanticipated case (faulty design logic). In such cases, the algorithm itself has gone awry, because the programmer has not worked through all the cases

© The Open University, 1993

correctly. This category encompasses both those which Knuth labels as “A = algorithm awry” and also those labelled as “S=surprise scenario”. Knuth’s A-vs.-S distinction can only be resolved by in-depth introspection, and is too fine-grained for the purposes of this study.

- **init:** Wrong initialisation; wrong type; definition clash. A programmer will sometimes make an erroneous type declaration, or re-define the meaning of some system keyword, or incorrectly initialise a variable. We refer to all of these as “init” errors, since the program begins with its variables, data structures, or function definitions in an incorrect starting state.

- **var:** Wrong variable or operator. Somehow, the wrong term has been used. The informant may not provide enough information to deduce whether this was really due to faulty design logic (**des.logic**) or whether it was a trivial lexical error (**lex**), though in the latter case trivial typos are normally mentioned explicitly as the root cause.

- **lex:** Lexical problem, bad parse, or ambiguous syntax. These are meant to be trivial problems, not due to the algorithm itself, nor to faulty variables or declarations. This class of errors encompasses Knuth’s “B=Blunder” and “T=Typo”, which are hard to distinguish in informant’s reports.

- **unsolved:** Unknown and still unsolved to this day. Some informants never solved their problem!

- **lang:** Language semantics ambiguous or misunderstood. In one case, an informant reports that he thought that 256K meant 256000, which is incorrect, and can be thought of as a semantic confusion. In another case, an informant reported a mismatch between the way a manual described some maximum value and the way in which it was actually dealt with by the compiler.

- **behav:** End-user's (or programmer's) subtle behaviour. For example, in one case the bug was caused by an end-user mysteriously depressing several keys on the keyboard at once, and in another case the bug involved some mischievous code inserted as a joke. These are really manifestations of behaviour external to the normal programming arena, but still warrant a category in their own right.

© The Open University, 1993

5.3.4.2 Results

Table 4 displays the frequency of occurrence of the nine underlying causes.

Category	Occurrences
mem: Memory clobbered or used up	13
vendor: Vendor's problem (hardware or software)	9
des.logic: Unanticipated case (faulty design logic)	7
init: Wrong initialisation; wrong type; definition clash	6
lex: Lexical problem, bad parse, or ambiguous syntax	4
var: Wrong variable or operator	3
unsolved: unknown and still unsolved to this day	3
lang: language semantics ambiguous or misunderstood	2
behav: end-user's (or programmer's) subtle behaviour	2
??? (no information)	*2*

Table 4. Underlying causes of the reported bugs.

Table 4 indicates that the biggest culprits were memory overwrites and vendor-supplied hardware/software problems. Even ignoring vendor-specific difficulties, one implication of Table 4 is that 37% of the nastiest bugs reported by professionals could be addressed by (a) memory-analysis tools and (b) smarter compilers which trapped initialisation errors. But what about the interaction between the cause of the bug, the reason for the debugging difficulty, and the debugging technique? That is precisely the focus of the next section.

5.4 Relating the Dimensions

To understand the ways in which the three dimensions of analysis interrelate, we can place every anecdote precisely in our three-dimensional space. For expository purposes (and because multi-dimensional diagrams are hard to discuss) let's consider just the following two-dimensional comparisons: (a) root cause vs. how found, and (b) how found vs. why difficult.

Table 5 compares root causes (row labels) against bug-finding techniques (column labels). Each cell entry shows the number of anecdotes with the given attributes. In cases where an anecdote reveals multiple attributes (say, it belonged partly to column 3 and partly to column 4), it is simply split evenly across the appropriate cells, hence the fractional entries. Tables 2, 3 and 4, incidentally, did *not* use this splitting technique, and correspond to tallies of the primary (first-reported) categories only.

© The Open University, 1993

CAUSE vs. HOW	gather data	inspeculation	expert recognised cliché	controlled experiments	??? (no info)	TOTALS
mem	8.50	4.00		1.00		**13.50**
vendor	4.00	3.00	1.00	3.00		**11.00**
des.logic	4.00	3.00				**7.00**
init	5.00	1.00				**6.00**
lex		1.00	2.00		1.00	**4.00**
var	2.00		.50		1.00	**3.50**
unsolved	3.00		.50	.50		**4.00**
lang	1.00		1.00			**2.00**
behav		1.00	1.00			**2.00**
??? (no info)	2.00					**2.00**
TOTALS	**29.50**	**13.00**	**6.00**	**4.50**	**2.00**	**55.00**

Table 5. Tally of root causes of bugs (rows) vs. how found (columns). Each cell entry (e.g. 8.50) is a tally of the number of anecdotes reporting that cell's row label (i.e. root cause) and column label (i.e. how found). Fractional entries reflect anecdotes which have been divided into multiple categories, so that an anecdote reporting both a "controlled experiment" and "expert recognised cliché" scores .50 in each cell. Note that tables 2, 3, and 4 only tallied the primary (first-reported) category for each dimension.

Of most interest is the relative density of anecdotes in the upper left-hand corner of the table, suggesting particularly that memory-clobbering errors could usefully be dealt with by better data-gathering tools. The density of the cell entries is *not* greater than that predictable by chance from the row and column totals alone (X^2 , df:36, = 45.30, ns), suggesting no reliable relationship between root cause and how found, though the cell densities are nevertheless of *a priori* interest to us as we try to develop the VITAL-BLM.

Table 6 compares reasons for difficulty (rows labels) against bug-finding techniques (column labels). Once again, the need for data-gathering tools is highlighted, this time for dealing with cause/effect chasms and cases in which other debugging tools are inapplicable. In this case, the density of certain cell entries *is* greater than that predictable by chance from the row and column totals alone (X^2 ,df:20, = 33.50, p. < .05), suggesting in particular that data-gathering activities are of special relevance when a cause/effect chasm is involved or when the built-in debugging tools are somehow rendered inapplicable.

© The Open University, 1993

WHY vs. HOW	gather data	inspeculation	expert recognised cliché	controlled experiments	??? (no info)	TOTALS
cause/effect chasm	9.83	3.00	1.50	2.50		**16.83**
tools hampered	9.83	2.00		2.00		**13.83**
WYSIPIG	2.00	2.00	1.50		2.00	**7.50**
faulty assumption	2.50	3.00	1.00			**6.50**
spaghetti	1.33	1.00				**2.33**
??? (no info)	4.00	2.00	2.00			**8.00**
TOTALS	**29.50**	**13.00**	**6.00**	**4.50**	**2.00**	**55.00**

Table 6. Tally of why bugs were difficult (rows) vs. how found (columns). Each cell entry (e.g. 9.83) is a tally of the number of anecdotes reporting that cell's row label (i.e. root cause) and column label (i.e. how found). Fractional entries reflect anecdotes which have been divided into multiple categories, so that an anecdote reporting *three* reasons for difficulty scores .33 in each of three relevant cells.

A niche of potential interest (and profit) to tool vendors is highlighted by looking at the relationship among the three dimensions: the most heavily populated cells are those involving data-gathering, cause-effect chasms and memory or initialisation errors. The implications of this finding are discussed in the next section.

5.5 Discussion: Lessons Learned

5.5.1 Comparison with fine-Grained Study

In the section 4 we reported on a fine-grained study of debugging in HyperTalk. In particular, the "why difficult" dimension was analysed at a more a fine-grained level of detail, revealing eight fundamental problems. We shall now compare the fine-grained study with the one reported here. Table 7 below lists each of those problems (left-hand column), shows which coarse-grained "why difficult" category they correspond to (middle column), and reminds us of the possible solution proposed in section 4 (right-hand column).

© The Open University, 1993

Fine-grained Problem	Coarse-grained "Why difficult" Category	Proposed Solution
1. The link between an error message and the offending source code line has to be deduced by the user (whereas it could be provided for free).	Cause/effect chasm; Tools inapplicable or hampered (long run to replicate);	**S1:** Computable relations should be computed on request, rather than be deduced by the user.
2. Performing a routine debugging/inspecting action involves dealing with many disruptive subgoals (e.g. switching modes to enable specific machine states).	Faulty assumption (typically about what "mode" the debugging tool is in)	**S2:** Atomic user-goals should be mapped onto atomic actions.
3. Access to the interpreter (and other features) in the middle of a break is disallowed.	Tools inapplicable or hampered (context precludes using debugger)	**S3:** Allow full functionality at all times.
4. The behaviour of built-in functions can not be monitored easily.	Tools inapplicable or hampered (context precludes using debugger)	**S4:** Viewers should be provided for "players" (any evaluable expression) rather than just "variables".
5. There is no meaningful coarse-grained view of execution.	Tools inapplicable or hampered	**S5:** Provide a variety of navigation tools at different levels of granularity.
6. Traversal of indirect data influences (data flow) requires too much detective work.	Cause/effect chasm	**S6=S1:** Computable relations should be computed on request, rather than be deduced by the user.
7. Understanding the precise conditions under which a particular event happens (control flow logic) requires too much detective work.	Cause/effect chasm	**S7=S1:** Computable relations should be computed on request, rather than be deduced by the user.
8. The "inner state" of an object can be deceptively different from its apparent state, requiring extra detective work to uncover.	WYSIPIG: what you see is probably illusory, guv'nor	**S8:** Displayable states should be displayed on request, rather than be deduced by the user.

Table 7. Relationship between fine-grained problems identified using self-report (left column) and coarse-grained categories of the current broad survey (middle column). Prospective solutions to each of the eight fine-grained problems are also identified (right-hand column).

Although the scope of the two studies was rather different, it is nevertheless gratifying that most of the problems found in the fine-grained study fell into the two most popular studies reported in this coarse-grained study of the current paper (namely, tools hampered and cause/effect chasm).

5.6 Solutions

These studies have identified several areas which need attention. Firstly, the most heavily populated cell in our three dimensional analysis suggests that a winning tool would be one which employed some data-gathering or traversal method to

© The Open University, 1993

resolved large cause/effect chasms in the case of memory-clobbering errors. Secondly, we can propose solutions to the "why difficult" problems by considering the specific cases brought to light by the fine-grained study described above. One way or another, all of the problems mentioned in Table 7 are connected with "directness" and "navigation". For example, the need to go through indirect steps, intermediate subgoals or obtuse lines of reasoning plagues the user encountering problems 1, 2, 3, 6, 7, and 8, and each of these problems can be addressed specifically.

In the next part of this document we shall describe the VITAL Bug Location Methodology and say how we aim to implement the solutions proposed in Table 7.

© The Open University, 1993

PART II

THE VITAL BUG LOCATION METHODOLOGY

PREFACE

In the first part of this document we first reviewed the work carried out on debugging KBS and traditional programs. We then reported on two studies which were aimed to provide concrete input from 'a state of art programming environment' and from 'the real world'. The first study took the form of an in-depth self-report. This study revealed eight fundamental problems in the debugging facilities provided by HyperTalk's programming environment: indirect access to troublesome source code; disruptive intermediate actions required; poor interpreter access during breaks; poor monitoring of built-in functions; no coarse-grained view of execution; no data flow analysis; no control flow analysis; deceptive view of inner states.

The second study analysed the debugging anecdotes collected from a world-wide email trawl which revealed three primary dimensions of interest: *why the bugs were difficult* to find, *how* the bugs were found, and *root causes* of bugs. Half of the difficulties arose from just two sources: (i) large temporal or spatial chasms between the root cause and the symptom, and (ii) bugs that rendered debugging tools inapplicable. Techniques for bug-finding were dominated by reports of data-gathering (e.g. print statements) and hand-simulation, which together accounted for almost 80% of the reported techniques. The analysis pinpointed a winning niche for VITAL bug location tools: data-gathering or traversal methods to resolve large cause/effect chasms which made some of the bugs so hard to locate. Other specific solutions, all of which emphasise issues of "directness" and "navigation" were developed by comparing the second study with the first.

In the second part of this document we present the VITAL Bug Location Methodology. The methodology rests on two principles derived from the two studies:

i) bug location should be based on visualizations of the execution, and

ii) the relations between entities, potentially separated by a large spatial or temporal distance, within an execution should be computed by specialist bug location agents.

The first principle leads to the construction of Viz, a framework and software tool to describe and implement software visualization systems, which we describe in section 7. By providing a descriptive abstraction for internally expressing the state of a program execution (players, events, and states) Viz augments earlier frameworks. The use of Viz to implement systems which differ widely in terms of their scope, content, form, method, interaction, and effectiveness, suggests that the framework is sufficient to design and implement all the visualizations required within VITAL.

The second principle leads to the inclusion of VITAL Bug Location Agents (VITAL-BLAs). VITAL-BLAs will take the burden from the KE to compute the indirect relationships between potentially distant points within the execution space. We describe VITAL-BLAs in section 8.

© The Open University, 1993

6. THE VITAL APPROACH TO THE SOLUTIONS PROPOSED IN THE STUDIES

As a gentle introduction into the VITAL-BLM we shall first describe how we have addressed the solutions proposed at the end of section 5. We group the proposed solutions into three broad categories which we address in turn. In the following sections we expand on the VITAL implementations of the second and third group of solutions.

6.1 S2 and S3

These two proposals were: S2 "Atomic user-goals mapped onto atomic actions"; and S3 "Allow full functionality at all times". In VITAL we will implement these two proposals by attempting to create the 'right interface' to the debugging tool. We do not know, right now, exactly what form the interface will take. What we do know is that the KE will interact by direct manipulation of the visualizations produced by the execution, both to pose queries (e.g. who inserted this piece of knowledge?) and to add new information (e.g. I know that bug is somewhere in *this* part of the execution?).

6.2 S4, S5 and S8

These three proposals were: S4 "Viewers for any evaluable expression"; S5 "Variety of navigation tools"; and S8 "Displayable states displayed on request". S4, S5 and S8 have had the most profound impact on the VITAL-BLM. The VITAL-BLM will be *visualization based.* By this we mean that we will provide visualizations for *all* the executions within the VITAL workbench (including animations of the conceptual models). Some executions may require several layers of visualization at different levels of abstraction. For example, an implementation can be considered at the domain level, at the level of interacting VITAL-KR Inference Modules (see [Motta *et al* 91]), or at the level of an individual inference module (i.e. a program written in a specific knowledge representation language).

The interface between the knowledge engineer and the debugging tool will be via visualizations of the entities created by a run of the KB. In section 8 we describe a framework and software tool for creating the VITAL visualizations.

6.3 S1=S6=S7

This proposal "Computable relations computed on request" was a solution to the cause/effect chasm, the fact that the symptom can be far removed in space and/or time from the root cause, and making it hard to detect. The mechanism for computing relations will be similar to the SNIFFER system [Shapiro, 1981] debugging agents provided by MRE [Brayshaw, 1991] (see sections 3.2 and 3.6). Each agent will traverse a stored history of an execution searching for a specific relation. As in MRE the expressible relations will be arbitrarily complex, thus satisfying S3.

7. A FRAMEWORK FOR CREATING SOFTWARE VISUALIZATION SYSTEMS

As we mentioned at the start of this part of the document the VITAL-BLM will be visualization based with visualizations provided for all the executable languages within VITAL. As a precursor to this large exercise we needed to a) evaluate existing visualization systems and b) create software support (specifically a software library)

© The Open University, 1993

for the creation of the various visualization systems. In this section, adapted from [Domingue *et al*, 1992; in press], we report on the results of this work.

7.1 What is Software Visualization?

The tools traditionally used by software engineers to help monitor and analyse program execution have been plain ASCII-text based debugging environments which usually allow the user to trace the currently executing code, stop and start execution at arbitrary points, and examine the contents of data structures. Although they can be understood by experts, these tools have a limited pedagogic value and by the early 1980's the work of Baecker [Baecker, 1981] and Brown [Brown, 1988] showed how algorithms could be animated with cartoon-like displays that show a high level abstraction of a program's code and data. Brown referred to this as “algorithm animation” since the emphasis was on meaningful graphical abstractions about the high level algorithm underlying a program. Both of these systems proved successful in computer science teaching, but since algorithm animations tend to be custom built (by hand) for specific algorithms, the overhead is too high for use by software engineers working on large projects.

A concurrent development in the mid-1980's was the appearance of systems which displayed graphical representations that were more tightly coupled with a program's code or data and showed more or less faithful representations of the code as it was executing. Although the displays were not as rich as the custom built algorithm animations, these systems were closer to the tools that software engineers might use. These “program animators” together with the algorithm animators became known as “program visualization” systems. We prefer the more descriptive term “software visualization” [Price, in press] which encompasses both algorithm and program visualization as well as the visualization of multi-program software systems. In this section we will use the term software visualization (SV) to describe systems that use visual (and other) media to enhance one programmer's understanding of another's work (or his own).

7.2 Classifying SV Systems

One of the first taxonomies of SV was that of Myers [Myers, 1986] (updated later as [Myers, 1990]), which served to differentiate SV, visual programming, and programming-by-example. In classifying SV systems, Myers used only two dimensions: static vs. dynamic and code vs. data. The first dimension is based on the style of implementation; static displays show one or more motionless graphics representing the state of the program at a particular point in time while animated displays show an image which changes as the program executes. The second dimension describes the type of data being visualized, be it the program source code or its data structures.

The taxonomies that have been proposed since Myers have also used few dimensions, which seems to ignore that fact that there are many styles for implementation and interaction as well as different machine architectures and ways of utilising them. Price, Small, and Baecker recently proposed a taxonomy [Price, in press] which describes 6 broad categories for SV systems: scope, content, form, method, interaction, and effectiveness. Each of these categories has between three and seven characteristics, for a total of thirty dimensions to describe each system.

© The Open University, 1993

This pragmatic classification system provides a means for comparing the functionality and performance of a wide range of SV systems, but it does not provide a language or framework for implementing new systems. Eisenstadt *et al.* described nine qualitative dimensions of visual computing environments [Eisenstadt *et al*, 1990] which can form the basis of a language for describing SV systems, but these serve only to describe the attributes of systems rather than drive their construction.

7.3 From Taxonomy to Framework and System: "what goes on?"

Taxonomies are useful, but we need more if we are to provide a firm basis upon which to describe SV systems in depth, let alone implement them. A *framework* for describing SV systems could provide extra leverage by being a little more prescriptive, i.e. making a commitment regarding how to approach the design and construction of SV systems. In fact, it is not a very big step from specifying such a framework to designing a system for *building* a SV system (SV system-building system). An important difference is that the former activity is merely a paper exercise, whereas the latter activity is intended to lead towards a working tool. Indeed, the latter activity serves as a useful forcing function: it encourages us to build re-usable libraries of software that we believe encapsulate important generalisations about SV system-building. The proof of the soundness of a design built in this way lies in the ability to use it both to reverse-engineer existing SV systems and construct new systems with ease.

By shifting our focus of attention from taxonomy to working system (and its underlying framework), we force ourselves to face the fundamental question of software visualization: "What goes on during program execution?" There are many different "truths" or "stories" that one can tell about the seemingly unambiguous behaviour of an executing program, as convincingly demonstrated (in the case of Prolog) by Pain and Bundy [Pain, 1987]. The choice of a certain story or metaphor to explain how a particular machine, programming language, or program works depends on the culture of the audience, their level of experience, and which points the author wishes to emphasise. Our problem is not how to work out which story to tell, but rather how to enable diverse story-tellers to carry out their craftwork. As in Propp's analysis of folktales [Propp, 1968], we want to understand the commonalities, the invariances, the essentials of a "story" that differentiate it from random scribblings or random collections of "sentences", and to do this at a level of abstraction which is meaningful to the community of "story-tellers" (SV system builders) we are addressing. This means that the "kernel truths" about memory addresses, registers, and CPU instructions are not necessarily of immediate use to us, because they usually offer too low a level of abstraction.

7.4 Programming Language Visualization vs Algorithm Animation

Several of the noteworthy SV building systems and frameworks focus on algorithm animation (AA), which means that the animations that they produce are custom designed and each new program requires manual annotation to animate it. Programs are animated in BALSA [Brown, 1988] by adding calls to the animation system at "interesting events" in the code. TANGO [Stasko, 1990] also animates programs with interesting event calls and it provides facilitates for smooth

© The Open University, 1993

transitions in animations. While producing aesthetically pleasing results, these techniques are not practical for large software engineering projects, which require a more automatic approach.

In Viz we have focused on supporting the construction of systems which visualize the execution of programming languages. By visualizing a programming language interpreter (or compiler) one also automatically gets a visualization for any program written in that language (thus achieving the *automatic* goal suggested in [Price, in press]). Programming Language Visualization (PLV) and Algorithm Animation (AA) overlap, but there are differences in the approach. AA systems typically show a very high level picture of a program's execution and the images that it generates can be far removed from the data structures contained in the program. The animations cover a narrow set of programs (typically a single algorithm). PLVs on the other hand have to deal with any program which can be realised in the language. Thus PLV displays usually have much simpler images than AA displays since they must be highly generalised whereas AA displays can be custom tuned. The problem for an AA is to show the characteristics (signature) of an algorithm as clearly as possible. The problem for a PLV is to allow arbitrarily large execution spaces to be examined in a comprehensible fashion.

Our approach is to concentrate primarily on PLV, but to provide generalisations which are applicable to AA as well. In the rest of this paper we describe the design of a SV system-building system (and framework) called Viz, which we have implemented as a prototype running in Common Lisp, CLUE and CLX on Sun workstations. Our Viz implementation has already been used to reconstruct three well known PLV systems: an OPS5 visualization system (based on TRI [Domingue *et al*, 1989]), a Prolog visualization system (based on TPM [Eisenstadt *et al*, 1988]), and a Lisp tracer (based on the Symbolics™ tracer). After describing the Viz architecture, we explain how two of these reconstructions were implemented. In order to explore the relationship between Viz's PLV-oriented approach and AA-oriented systems we have also used Viz to implement some of the animations from BALSA (sorting) and TANGO (bin-packing). We conclude with a comparison of the terminology used in BALSA, TANGO, and Viz to describe abstractions and we highlight the advantages of the Viz design.

7.5 Viz Fundamentals

In Viz, we adopt the "story-telling" metaphor by considering program execution to be a series of *history events* happening to (or perpetrated by) *players*. To allow our "story-tellers" (SV system builders) considerable freedom, a player can be any part of a program, such as a function, a data structure, or a line of code. Each player has a name and is in some *state*, which may change when a history event occurs for that player. History events are like Brown's "interesting events" in BALSA—each event corresponds to some code being executed in the program or some data changing its value. These events are recorded in the history module, which allows them to be accessed by the user and "replayed". Events and states are *mapped* into a visual representation which is accessible to the end-user (the programmers who need to use the SV system, not the SV system builder). But the mapping is not just a question of storing pixel patterns to correspond to different events and states—we also need to specify different views, and ways of navigating around them. Before we describe the Viz architecture in detail, let's preview the main ingredients:

© The Open University, 1993

• *Histories*: a record of key events that occur over time as the program runs, with each event belonging to a player; each event is linked to some part of the code and may cause a player to change its state (there is also some pre-history information available before the program begins running, such as the static program source code hierarchy and initial player states).

• *Views:* the style in which a particular set of players, states or events is presented, such as using text, a tree, or a plotted graph; each view uses its own style and emphasises a particular dimension of the data that it is displaying.

• *Mappings:* the encodings used by a player to show its state changes in diagrammatic or textual form on a view using some kind of graphical language, typography, or sound; some of a player's mappings may be for the exclusive use of its *navigators*.

• *Navigators:* the tools or techniques making up the interface that allows the user to traverse a view, move between multiple views, change scale, compress or expand objects, and move forward or backward in time through the histories.

This framework is equally at home dealing with either program code or algorithms, since a player and its history events may represent anything from a low-level (program code) abstraction such as “invoke a function call” to a high level (algorithm) abstraction such as “insert a pointer into a hash table”.

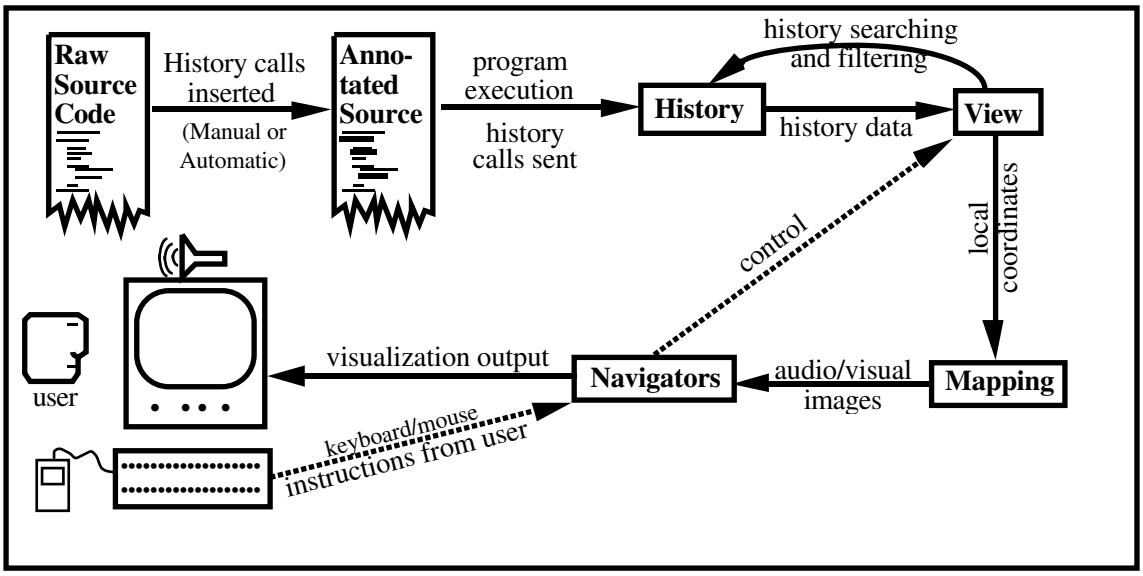


Figure 13. The architecture of Viz.

7.6 The Viz Architecture

Figure 13 shows the general architecture of Viz. The target system source code is annotated to generate history calls. When the system being visualized is a programming language, hooks into the interpreter or compiler are used to generate history events. As the code executes, the inserted calls cause “interesting” events regarding players to be recorded in the history module.

When the user runs the visualization, the view module reads the history data at the request of the navigator. The view module sets the layout of the history events

© The Open University, 1993

and sends local coordinates for each history datum through the mapping module, which draws a graphical or textual representation for each event. The screen images are then transformed and presented on the screen by the navigator. The user interacts with the visualization using the navigator, which sends control signals to the view module to cause all changes in the visualization, such as panning, zooming, local compression and expansion, and moving forward and backward in time through the program execution space.

7.6.1 Histories

Histories are the data that form the basis of *any* visualization and they roughly correspond to what Price, Small, and Baecker called "content" in their taxonomy. Most SV systems require the visualization designer to manually select the events in the program which will characterise it or show each of its continuous or discrete states. This can be done by inserting explicit instructions in the code or by attaching "probes" to various data structures (this requires that the language interpreter be modified). The probe method is necessary when one wishes to visualize a programming language itself. It is also possible to have the events automatically selected by using algorithm recognition techniques, as shown by a recent system which recognises simple programming clichés and produce algorithm animations [Henry, 1990]. This algorithm recognition work has only been successful with a small set of simple programs, however, and thus can not yet scale up to the demands of full-size software engineering visualizations. A simpler approach using a preprocessor to automatically insert event calls into the source code at points related to the language syntax, such as procedure calls and returns, can be used to achieve automatic visualization of programs of any size [Price, 1991], but the visualizations tend to display few abstractions.

We aim to integrate the PLAN diagram formalism into the Viz history mechanism. We will use a PLAN recogniser Lutz [1989, 1992] currently being ported from POP-11 into Common Lisp within our laboratory. We will add parsers from this tool to the VITAL formalisms.

© The Open University, 1993

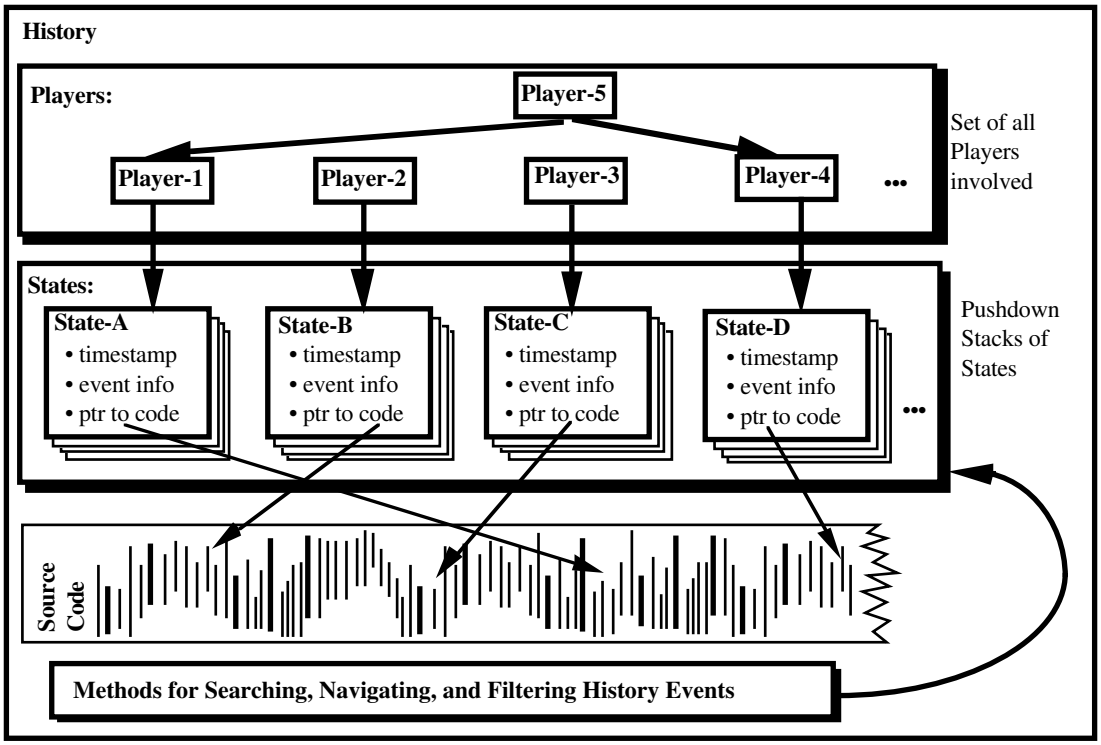


Figure 14. A snapshot of a prototypical history structure.

The first task for the visualization programmer using Viz is to decide what types of events may occur during program execution, which elements in the program will be the players and how the players change state. After defining these, the programmer may insert *create new player* and *note event* calls in the code, which form the interface between a program and its visualization.

Figure 14 shows a prototypical history structure in the history module. This consists of a set of players and a sequence of history states. Each player has a name, a pointer to its current history state and a pushdown stack of previous states. A player may contain other players, as shown by player 5 in figure 14. This feature is useful in navigation. Each state has a timestamp, a pointer to the appropriate segment of source code and an event structure. As a program executes, new players and history states are created, and existing players are "moved" into new states, pushing previous states onto a stack. The various states of the players are caused by the different types of events.

The choice of players and event types together with the judicious placement of *note event* calls in code determine the execution model. Currently, we do not advocate any methodology for creating the execution model, except to point out that events are the "things that happen" in a program causing a player or players to move into a particular state.

7.6.2 Views

A view can be thought of as a perspective or window on some aspect of a program or algorithm, with (possibly) many views making up a visualization. A view is a

© The Open University, 1993

kind of style for laying out the mapped players, so it is possible to have two different views of the same players in order to emphasise different details.

Stasko [Stasko, 1990] states:

> "In designing animations with TANGO, I quickly discovered the need to repeatedly create logical structures of objects such as rows, columns, graphs and trees." (pp. 33-34)

Stasko addressed this by implementing a package of high level macros. Rather than an add-on or macro package, we see these complex entities as a central part of any visualization.

There are some similarities between our views and the animation views, adapted from the Model-View-Controller paradigm, described by [London & Duisberg, 1985]. The main difference is that within the animation views, the layout, handled by views in Viz, and appearance, handled by mappings in Viz, are handled together. Each view in Viz can be thought of as embodying a style of formatting collections of objects.

We have observed that the methods or procedures for drawing and managing many graphical visualizations can be characterised by an inheritance hierarchy. For example, if one has a set of methods for drawing a histogram, then many of these methods could be reused for a "thermometer" display, since a thermometer is just a histogram with one data element. Similarly, if one has a set of methods for managing generalised directed or un-directed graphs then these would form the basis of any tree drawing methods that were required. This has an important implication for the design of a SV system-building system: it means that by implementing the views as a hierarchy with inheritance, new views that are required can be added as new "child nodes.". Since these new nodes inherit the methods of their parents, a great deal of code reusability is achieved.

Figure 15 shows our initial view hierarchy. The first level is based on the number of spatial dimensions required to draw the display (note that we do not address displays of more than three dimensions, which is beyond the scope of this work—see [Feiner, 1990] for a good treatment of this). Three dimensional displays of data are still quite rare and are often achieved by mapping onto two dimensions and relying on the ability of the viewer to see the perspective. High speed graphical workstation technology has now made true stereo displays practical and we expect to see SV designers take advantage of both this and virtual reality technology as it becomes better understood.

As the hierarchy indicates, two dimensional displays are the most common and we have implemented a number of methods for several different two dimensional styles. Graphs are a common style for displaying many kinds of data and they usually appear in two dimensions on a workstation screen, but this is a convenience of media: graphs are actually multi-dimensional and the methods required for their management have little in common with the other two dimensional views.

© The Open University, 1993

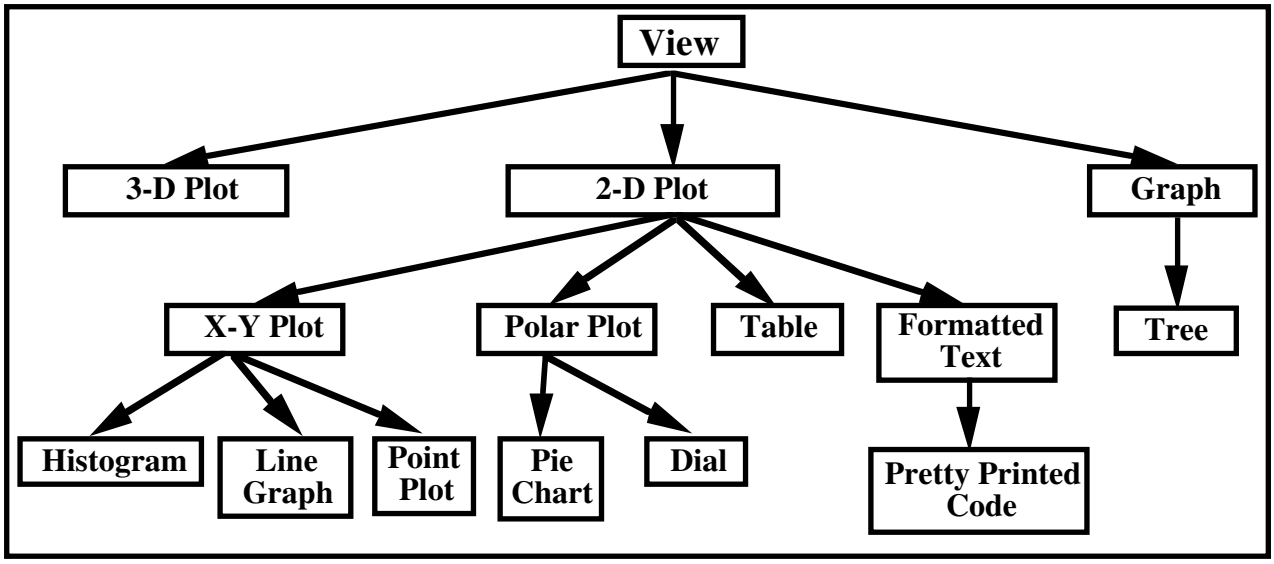


Figure 15. The Inheritance Hierarchy of Views

An individual view is responsible for the layout of the visualization and is controlled by the navigator. The view requests data from the history module and sends it to the mapping module, which decides the appearance. The view module then tells the mapping module where to display the mapped data. This means that the view module is concerned only with the position of the history data item, not its appearance.

A view must be initialised before it can be used to display data. The type of initialisation required depends on the view, with children having an initialisation that builds upon (refines) that inherited from their parents. For example, our prototype implementation of the tree view takes two parameters: (i) the root node; and (ii) a function which returns the children of a given node.

The view module is also responsible for managing the compression (elision) and expansion of elements in the display based on user commands from the navigator. If the user selects the compression of display elements, such as a subtree in a tree hierarchy, then the view module groups the players concerned into a new player which has its own mapping to represent the compressed players. When the navigator tells the view module that an element is to be expanded, the view disbands the new player and displays the individual mappings for the players.

The navigator may also instruct the view module to show a more coarse grained display of the visualization data, perhaps so that long term changes over time are visible or to increase the rate of display. This increase in granularity is achieved by switching to another view which filters the history data.

7.6.3 Mappings

The goal of a mapping is to communicate the maximum amount of information about a player's state while imposing the least possible cognitive load on the user. Some general advice on the subject of communicating information may be found in

© The Open University, 1993

Tufte's books [Tufte, 1983; Tufte, 1990], but the problem of finding an effective SV mapping is far more complex. These mappings must communicate large and varying amounts of information to an audience in a very specialised culture. Software engineers do not have a standard notation for communicating about high level abstractions (compared with, say, chemists, who have several internationally recognised notations). Even if a good notation existed for describing computing abstractions, there is still the problem of algorithm animations which often use abstractions that are completely unrelated to computing.

One attempt at solving this problem is the use of graphic mappings that are based on detailed models of how programmers think about programming (e.g. [Anderson, 1984]). This strategy is promising, but such models focus on programmer's plans rather than on ways of graphically visualizing such plans. With such little guidance, the visualization designer is often left to use intuition along with typographic techniques and graphics to create effective mappings.

In conjecturing a theory of effectiveness of graphical languages, Mackinlay [Mackinlay, 1986] noted Cleveland and McGill's observation that people accomplish the perceptual tasks associated with the interpretation of graphical presentations with different degrees of accuracy. Using psychophysical results, Mackinlay extended Cleveland and McGill's work to show how different graphical techniques ranked in perceptual effectiveness for encoding quantitative, ordinal, and nominal data. He found that the position of the data item in the x-y plane is ranked first for all three types of data, which is why we separate the view layout from the mappings. The other techniques which may be varied to create an effective mapping are (in decreasing order of effectiveness): colour hue, texture, connection, containment, density (brightness), colour saturation, shape, length (size), angle or slope (orientation), and area (or volume).

A mapping in Viz is attached to a particular type of player, event or state, and view. Multi-method inheritance occurs over the class of entity and view, allowing a Viz user to formulate expressions such as "all entities in view-x are to be displayed as a filled triangle", "entity-y is always displayed as a white circle" and "entity-a is displayed as a circle in tree based views but as a square in all other views". Mappings can be inherited, forming an inheritance hierarchy in much the same fashion as views. Mappings can also be aural. Currently, an audio mapping is a sound file in the Sun™ .au format.

7.6.4 Navigators

In a software visualization, one must provide tools for navigating through both the space (two or three dimensions) and time of the execution history. Brayshaw and Eisenstadt [1991] outline various navigation techniques which we have refined to form the basis of the Viz navigators:

• *panning* - moving through the 2-D (or 3-D) space of a view which is larger than the window that it is displayed in;

• *searching* - finding an event in the execution space (selectivity);

• *scale* - changing the magnification factor (zooming) for a view, including non-uniform scale changes (e.g. fisheye views);

© The Open University, 1993

• *granularity and detail control* - changing the way in which history events are filtered, which may have the effect of compressing or expanding time or individual elements;

• *customisable layout* - allowing the user to alter the display in arbitrary ways (e.g. manually adjusting a complex graph layout to minimise edge crossings).

The Viz navigator module encapsulates the interface between the user and the visualization, although the methods for performing the navigation tasks are found in the view module, thus allowing custom navigation interfaces to be built independently of the task. This also means that when new views are developed the SV system builder must provide a minimum level of navigation functionality for the default navigation tools. For example, we could construct a detail control module for tree based views which would be able to collapse subtrees into a single node.

Our prototype provides a replay panel (see screen snapshots, Figures 16 and 18) for searching, which has buttons for moving to the beginning or end of the animation, single-stepping forward or backward, playing forward, and stopping. Stepping in Viz involves notifying the history and view modules of the change of focus. The history module selects the next appropriate event. Currently, the view module can either simply redraw the mappings for the old and new focus points, or smoothly animate between the focus points in a style similar to that found in [Stasko, 1990]. Horizontal and vertical scroll bars are provided for panning while simple zoom in and zoom out buttons provide scaling. The user can select a fine grained view of a data element by clicking on it.

There are many techniques for navigating through large information spaces (e.g. [Mackinlay, 1991]) and since many techniques are application specific, we leave specific implementation choices to the individual visualization designer.

7.7 Examples Defined in Viz

The descriptions of the three systems that we have implemented using Viz are presented in table 8 along with the two examples from BALSA and TANGO animations. The table provides a summary of the players, states, events, mappings, and views used in each visualization. Each row represents a distinct Viz entity type and each column represents one of the visualizations. The players row lists the players which can take part in each example. The states row shows the possible states players can enter. The events row shows the events which cause state changes. For example, the events for the Bin-Packing animation are:

• attempt-fit - attempting to fit an item into a bin,

• succeed-fit - succeeding in placing an item inside a bin, and

• new-item - a new item to try and fit into one of the bins.

The mappings row contains, in order, the icon mapping for each state. The views row lists the names of the possible views in decreasing order of granularity. The connection between a view and the history is also shown in the views row. For example, the view cell for Prolog indicates the following:

© The Open University, 1993

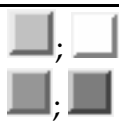
	Prolog	OPS5	Lisp	Sort	Bin-Packing
Player	predicate instantiation (goal)	rule	form	data item	data item bin
States	pending goal; succeeded; failed; failed on backtracking; redo-goal	failing to match working memory; matching working memory; firing	unevaluated; evaluated	location	attempting to fit; succeeded; new
Events	call; exit; fail-1st; fail-nth; redo	matching against a working memory element; choose for firing	call; return	assignment of item to cell	attempt-fit; succeed-fit; new-item
Mappings	 (colour)		-> *italic*; **<- bold**		
Views (in order of decreasing granular-ity)	tree: players, player's current state; formatted text	table: players vs. cycles, player's state @ cycle formatted text	pretty printed code: player's current state	point plot: players, player's value & current state; formatted text	point plot using rectangles: bin-players and current state

Table 8: Viz description of five example systems.

• each node in the tree based view corresponds to a player,

• when displaying a player we use the player's current state, and

• the fine-grained view for a player in this view will use a formatted text type of view.

We shall now explain the first and second columns, which are KBS specific, in detail.

© The Open University, 1993

7.7.1 A Prolog Visualizer

The Prolog visualizer is based on the Transparent Prolog Machine (TPM) [Eisenstadt *et al*, 1988]. TPM uses an AND-OR tree representation where the nodes represent goals which are instantiated Prolog predicates, and the arcs represent conjunctions or disjunctions of subgoals. A similar visualizer was used within the KEATS project [Domingue, 1989; Eisenstadt, 1990] to visualize the backward chaining production system.

The players in the visualization are the instantiated Prolog predicates or goals in the proof tree. The events, which are adapted from the Byrd Box Model [Byrd, 1980], are: call (trying to prove a goal), exit (a goal succeeding), fail-1st (a goal failing the first time attempted), fail-nth (a goal having succeeded earlier, later failing on backtracking), and redo (re-attempting to satisfy a goal). There is a corresponding state and mapping for each event type (shown in respective order so equals call, equals exit, etc.).

The visualization is built on top of the Prolog interpreter provided with the LispWorks™ environment. Hooks into the interpreter have been provided which 'spit out' the required events containing the current goal and event type.

Figure 16 shows a screen snapshot of the proof for the goal ?- party(?x) given the following Prolog database (adapted from [Eisenstadt & Brayshaw, 1988]).

```
party(?x) :- happy(?x), birthday(?x).

party(?x) :- friends(?x, ?y), very-sad-person(?y).

happy(?x) :- hot, humid, cut, swimming(?x).

happy(?x) :- cloudy, watching-tv(?x).

happy(?x) :- cloudy, having-fun(?x).

cloudy.

humid :- had-lots-of-rain.

had-lots-of-rain :- lots-of-clouds, clouds-full-of-water.

lots-of-clouds.

clouds-full-of-water.

hot :- sun-out.

having-fun(tom).
having-fun(sam).

swimming(john).
swimming(sam).

watching-tv(john).

very-sad-person(bill).
very-sad-person(sam).
```

© The Open University, 1993

```
birthday(rae).
birthday(samson).

friends(tom, john).
friends(tom, sam).
```

Bearing in mind that atoms beginning with "?" depict variables in this approximation of Edinburgh-syntax Prolog, the first five clauses define a) the relation that someone has a party if they are happy and its somebody's birthday or if they have a friend who is very sad; and b) the relation that a person is happy if it is hot and humid and they are swimming, or if it is cloudy and they are watching television or if it is cloudy and they are having fun.

Figure 16 below shows the visualization for the execution of the party(?x) goal. The party goal has succeeded because the two subgoals of finding a friend and the friend being a very-sad-person succeeded. The goals hot, humid, had-lots-of-rain, lots-of-clouds and clouds-full-of-water have had the predicate names 'clouded'. This happens to all the left siblings, and their descendants, of a cut goal when the cut goal succeeds. This indicates that these goals are frozen. When a cut goal is backtracked into (as has happened in figure 16) the clouds darken. The small window in figure 16 shows a fine-grained view for the happy goal. The fine-grained view shows the stack of all the events which where executed on the goal (the most recent event is at the bottom).

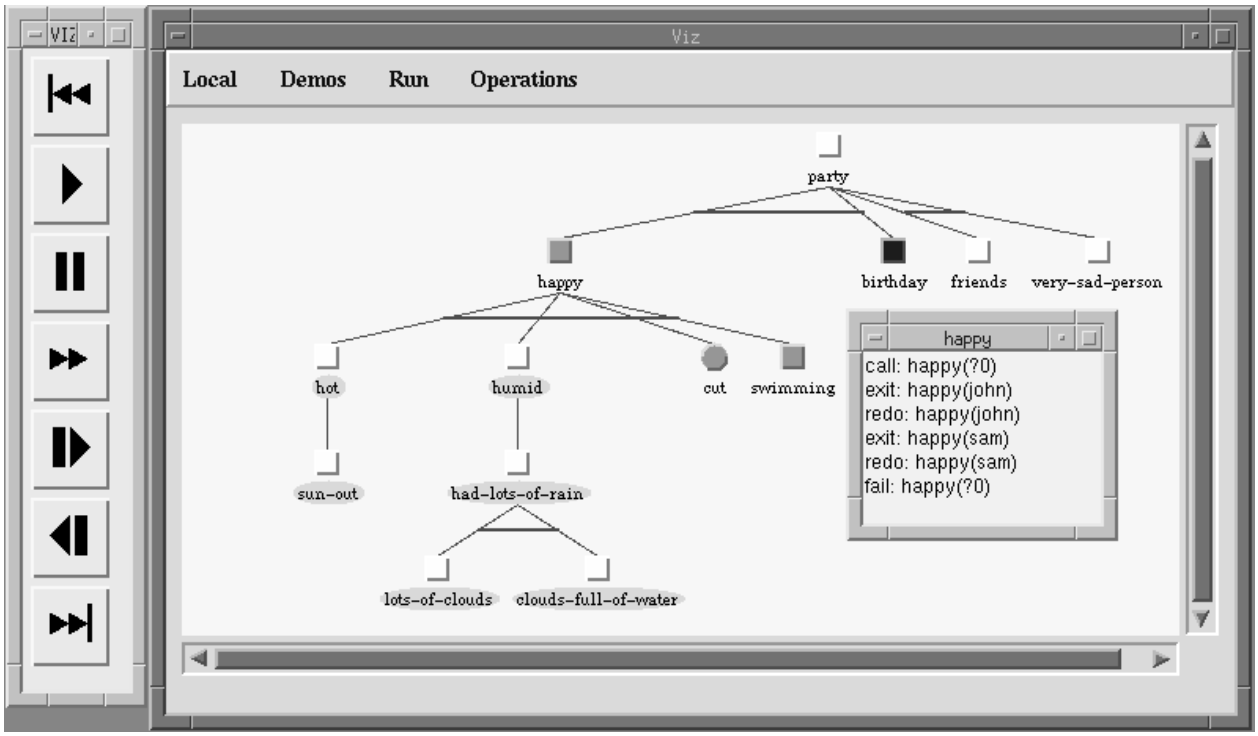


Figure 16: Screen snapshot of prolog visualizer (greyscale from a colour screen).

The collapsing of subtrees as in figure 17 allows Viz to deal with Prolog programs of any size. If required a collapsed subtree can be expanded again using the operations menu.

© The Open University, 1993

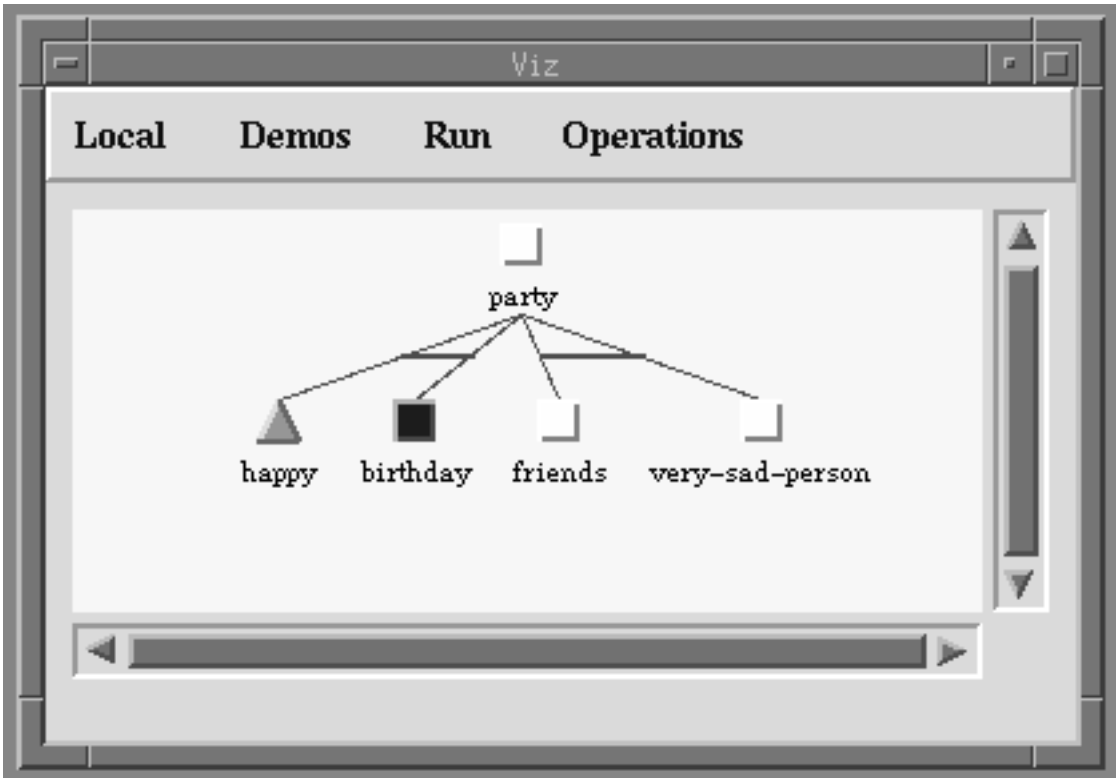


Figure 17. A screen snapshot of a visualization of the Party example with the Happy node collapsed.

7.7.2 An OPS5 Visualizer

We have implemented visualizations of two OPS5 implementations. The first is an OPS5 available with the Generic BlackBoard system [Corkill, 1986] (a re-write of the public domain OPS5 available from Carnegie-Mellon University). The second is a port of the KEATS rule interpreter ported from the Symbolics™ to the Sun. Both visualizations are very similar and share 90% of their code. We shall describe the ported KEATS rule system. The players in the OPS5 visualization are the rules and the events are: a rule firing and invoking the backward chainer, a rule firing, a rule matching working memory but failing to fire and a rule failing to match working memory.

Figure 18 below shows a screen snapshot of the visualization after running the Highway example [Poltreck *et al,* 1986]. These rules find the route between two cities. The initial working memory for the run was ((origin austin) (destination dallas)) and the output was:

AUSTIN --> 60 --> TEMPLE --> 34 --> WACO --> 41 --> HILLSBORO --> 48 --> DALLAS
Total mileage: 183

AUSTIN --> 10 --> SAN-ANTONIO --> 20 --> FREDERICKSBURG --> 50 --> DALLAS
Total mileage: 80

© The Open University, 1993

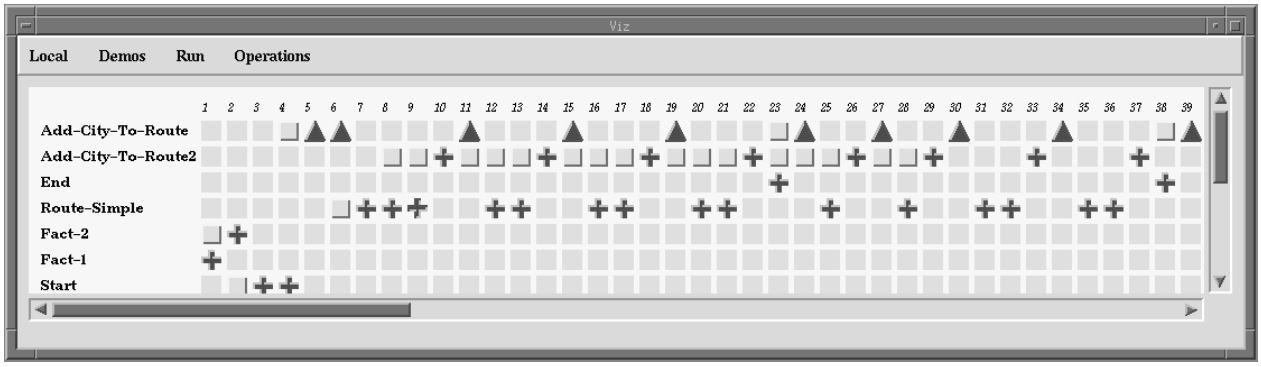


Figure 18. A visualization of an OPS5 rule interpreter. The indicates a rule fired and invoked the backward chainer, the indicates that rule fired, a indicates that a rule matched against working memory creating one or more instantiations but did not fire, and a indicates that the rule failed to match working memory (note the and symbols are more easily distinguishable on a colour screen).

We can see from figure 18 that there is a pattern occurring. After the initialisation rules Fact-1 and Fact-2 fire there is a repeated cycle of Add-City-To-Route, Route-Simple (firing once, twice or three times) then Add-City-To-Route2 firing. Each loop cycle extends the route by one more city.

Fine-grained views of the execution (created by clicking on the appropriate icon) show the rule definition and all the instantiations for a rule at a particular cycle. A fine-grained view for the rule Route-Simple at cycle 9 is shown below.

© The Open University, 1993

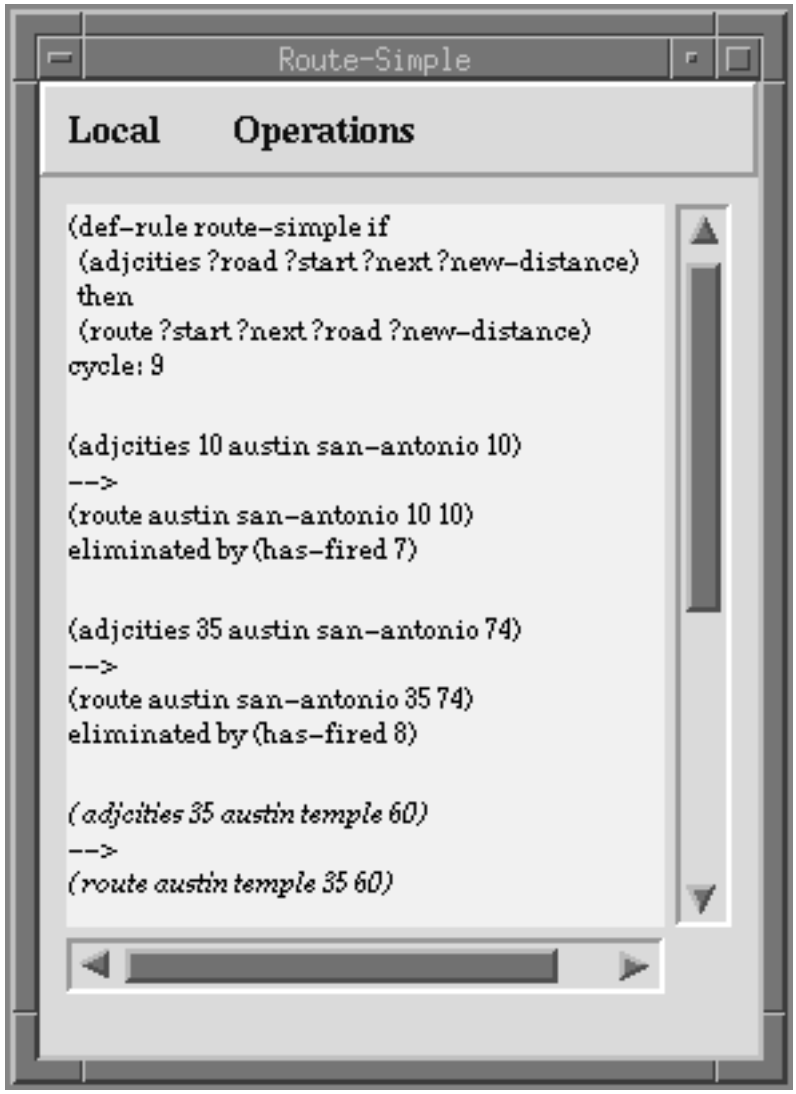


Figure 19. A fine-grained view of the rule Route-Simple at cycle 9. Below the definition of the rule are all the instantiations active for the rule at cycle 9. The first two instantiations were ruled out because they had fired, in cycles 7 and 8 respectively. The last instantiation is in an italic font because it fired in cycle 9.

7.7.3 Additional Systems

To fully exercise our evolving framework across a range of players, events, states, mappings, and views, we have also used Viz to re-implement the textual visualization provided by the Symbolics™ Lisp stepper/tracer which uses layout to summarise the execution history of the Lisp evaluator (the Viz implementation actually improves on this by using colour and typography as well). We have also duplicated some of the well known AA examples from BALSA (sorting) and TANGO (bin packing) to show that the system can be used to easily construct custom algorithm animations as well.

7.8 A Comparison of Viz, BALSA, and TANGO Terminology

Although each of the goals of Viz, BALSA, and TANGO are somewhat different, the importance of the pioneering work of BALSA and TANGO is such that a close

© The Open University, 1993

comparison of terminology is warranted. Viz terminology is designed to allow existing systems to be described as well as implement new ones. Table 9 shows the systems in left-to-right chronological order, mapping the similar terminology across systems where appropriate, and highlighting differences accordingly in the "comment" column.

BALSA	TANGO	Viz	Comments
Interesting (Algorithm) Events	Algorithm Operations	Events and Create Players	The BALSA and TANGO terms are virtually identical while Viz events are more general, can be arranged hierarchically, and are designed to relate to the code rather than the algorithm.
Modellers	Image, Location, Path, and Transition	States and Players	In describing a visualization's internal representation, TANGO adds to the BALSA framework by providing 4 abstract data types (geared towards animation); Viz's states and players are program execution level abstractions.
Renderers	Animation Scenes	Mappings and Views	BALSA provides a general mechanism for each view while TANGO provides reusable libraries of animation scenes; Viz discriminates between the actual images that are mapped to the screen and the style in which they are displayed (the view).
		Navigators	BALSA and TANGO don't specify any kind of user interface interactions within the framework, nor techniques for dealing with arbitrarily large programs.
Adaptor and Update Messages		History	The Viz history is a structure for the collection of events, states, and players generated during program execution. The history module includes various searching and filtering functions.

Table 9. A Comparison of Viz, BALSA and TANGO Terminology

8. BUG LOCATION AGENTS

In this section we describe how we will implement the S1 'Computable relations' solution first discussed in section 4. You may recall that this solution was to alleviate the difficulties caused when the connection between important parts of a bug were indirect or large in terms of space or time.

We shall now briefly outline the architecture of a VITAL Bug Location Agent (VITAL-BLA). Each agent will have:

• a name,

• a trigger function - when this function returns true the agent will be activated.

• a goal to satisfy - the format of the goal will depend on the type of agent, it may for example be a pattern.

• a goal satisfied function - this function determines whether the goal has been satisfied or not. This may be a simple pattern matcher or a full blown KB. Following

© The Open University, 1993

S3 (see section 4.9.2 and 5.5.1) we allow the goal satisfied function to use the full power of the VITAL-KRL.

• a knowledge store - an area local to the agent which stores the agents bug location knowledge.

• a set (possibly empty) of sub agents which the agent can activate if needed.

The trigger function, goal, goal satisfied function, and knowledge stores will be inheritable. In contrast to the agents within MRE, VITAL-BLAs will not be confined to one of three classes but will be able to 'mix and match' various styles of representation as required.

An activated agent will be given its own process and then run. A 'run' will typically traverse an execution history trying to match a portion of the history against some stored pattern.

8.1 Inter-Process Product Bug Location

The space problem (mentioned in sections 4 and 5) is compounded by the fact that the VITAL knowledge engineering methodology is based around the successive refinement of the four process products: user requirements, conceptual model (CM), design models (DM), and executable code (EC). Due to this sequential process, any bug that is introduced in a particular model, will be inherited by successive ones where it may manifest itself. For example, an incorrect OPS5 rule may be the result of an erroneous piece of knowledge elicited from the expert, and then correctly transformed into a conceptual and design model. In order to locate the true source of the bug a link from the buggy rule to appropriate parts of the design and conceptual models and to the elicited piece of knowledge must be present. To enable this inter-process product debugging we will keep 'derived from' links between the process products as shown in the figure below.

© The Open University, 1993

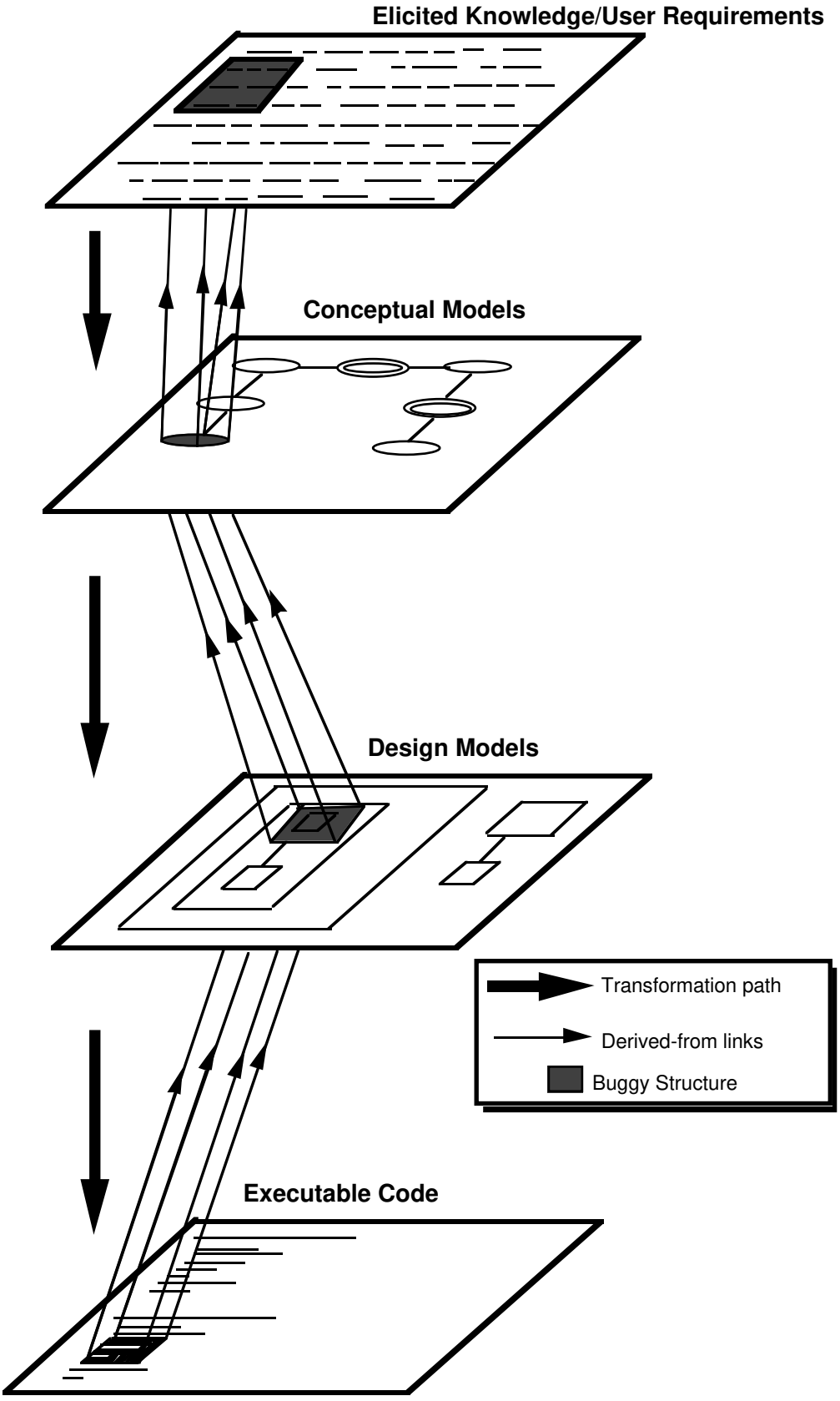


Figure 20. The use of 'derived from' links to aid in inter-process product debugging.

© The Open University, 1993

We shall describe how inter-process product bug location would work using a small example. We shall show how a bug within an Executable Code (EC) Process Product would be traced back to the Design Model (DM) Process Product.

The example is taken from the VITAL implementation [Motta, 1993] [Motta *et al*, in press] of the Sisyphus I problem [Linster, 1992]. Put basically, the problem is to assign members of a research group to a set of rooms. Each person in the group is either a researcher, a manager, the head of the group or a secretary. The constraints on the problem are:

- the head of the group should be in a central office,
- the secretaries should be near the head of the group,
- smokers and non-smokers should not share an office,
- as far as possible researchers sharing an office should be working on different projects.

Within VITAL we can semi-automatically generate an implementation from a design [Motta, 1993] [Motta *et al*, in press]. The design consists of a diagram from which we can generate an executable subset of our informal design language (VITAL-OCML [Koopman, 1991] [Motta, 1993]). The generated VITAL-OCML design is fleshed out by the KE until it becomes an implementation. In our scenario the initial diagram for the design of the VITAL solution (taken from [Motta *et al*, in press]) is:

© The Open University, 1993

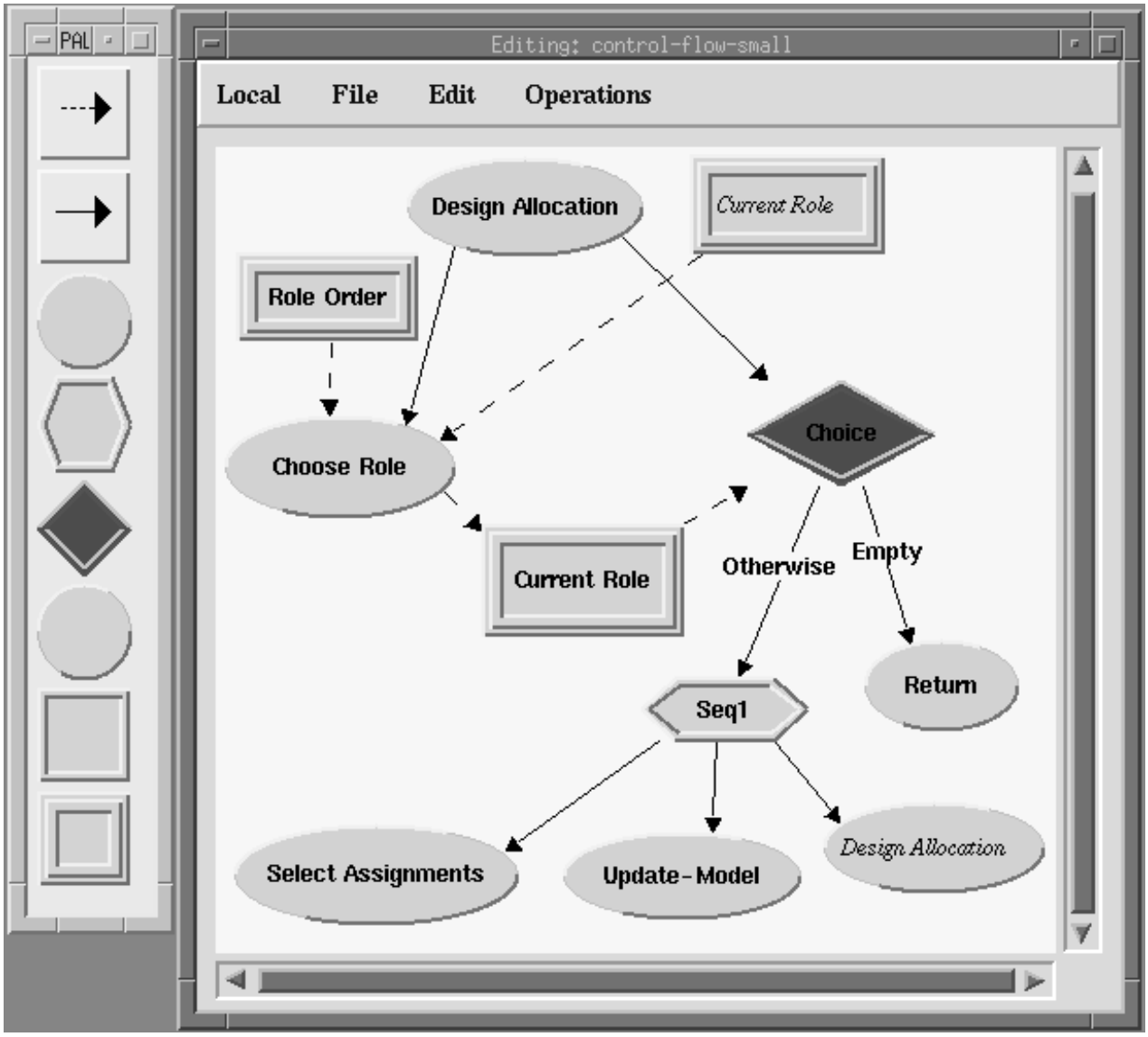


Figure 21. The design level control flow diagram in the VITAL-OCML.

The diagram above is created by dragging icons from the 'wells' within the palette on the left of the figure, and linking them together. The oval nodes are tasks and the rectangular nodes are models. Three special control nodes are also used. These are diamonds which represents a choice point, hexagons which represent a sequence of tasks and return labelled nodes which represent a return. The solid lines represent control flow and the dashed lines represent dataflow.

The knowledge engineer would expand on the subtasks 'Select Assignments' and 'Update Model' within new diagrams. From these diagrams the VITAL design tool automatically generates a textual representation for all the tasks and models. The KE would then flesh out this initial design into an executable solution. A domain level visualization of the execution, created in Viz, is shown in figure 22 below:

© The Open University, 1993

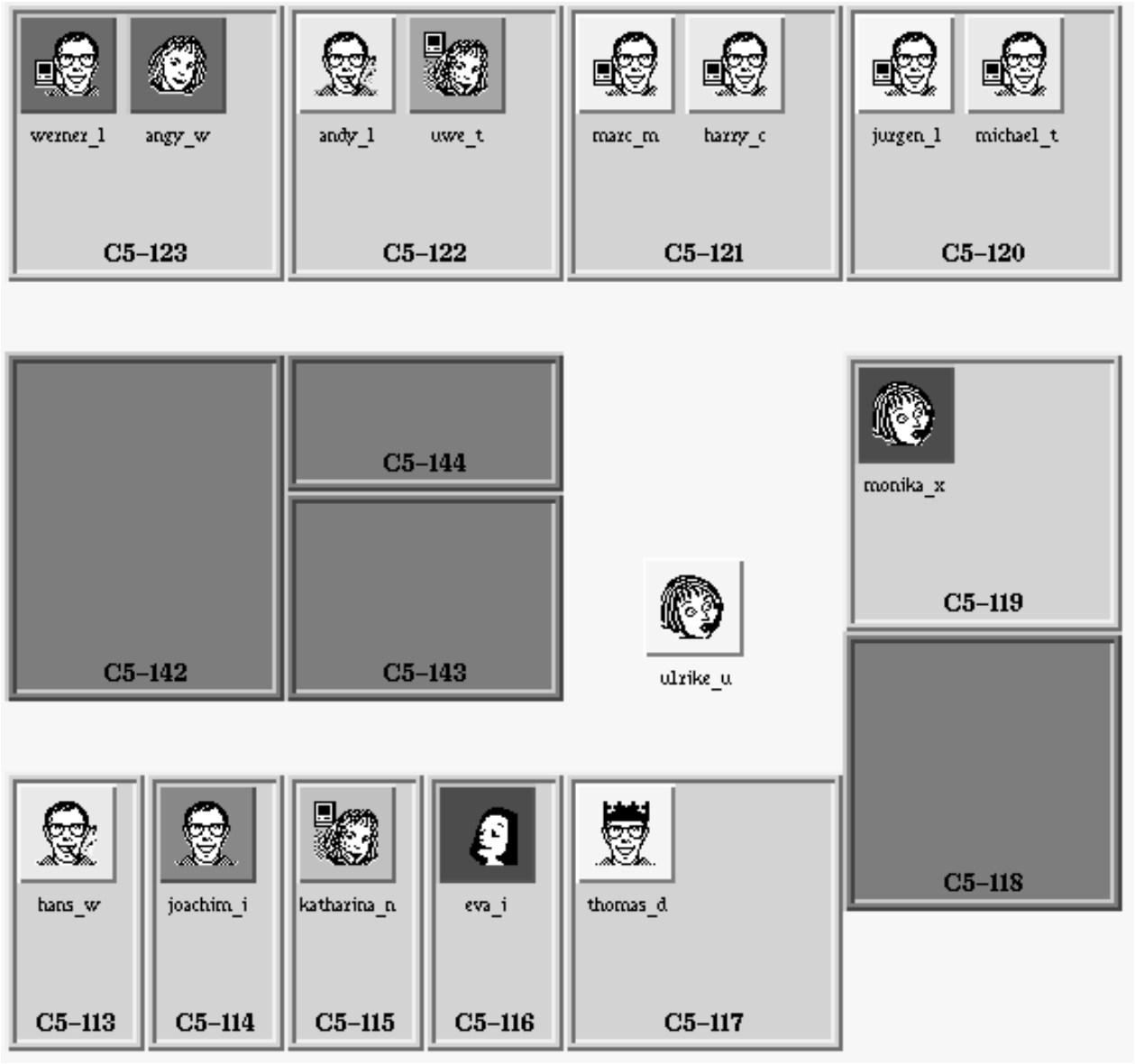


Figure 22. A domain level visualization of a buggy implementation of the Sisyphus problem.

Each icon represents a member of the group. The colour of each icon indicates the current project of the member. The visual appearance of an icon indicates a person's role within the group in the following way:

- or - researcher,
- or - smoker,
- or - hacker,

© The Open University, 1993

• or - hacker and smoker,

• - manager,

• - secretary,

• - head of group.

We can see from figure 22 that one of the members of the group, the secretary ulrike_u has not been allocated a room.

In our scenario the KE would flag the icon for ulrike_u as 'being in the wrong state' (i.e. not in a room). One of the agents activated would be 'violations of the design' an agent which searches for violations of the data flow dependencies expressed within the KB design.

When the agent found a discrepancy it would highlight appropriate parts of the design diagram and the execution visualization. The highlighted execution visualization is shown in the figure below.

© The Open University, 1993

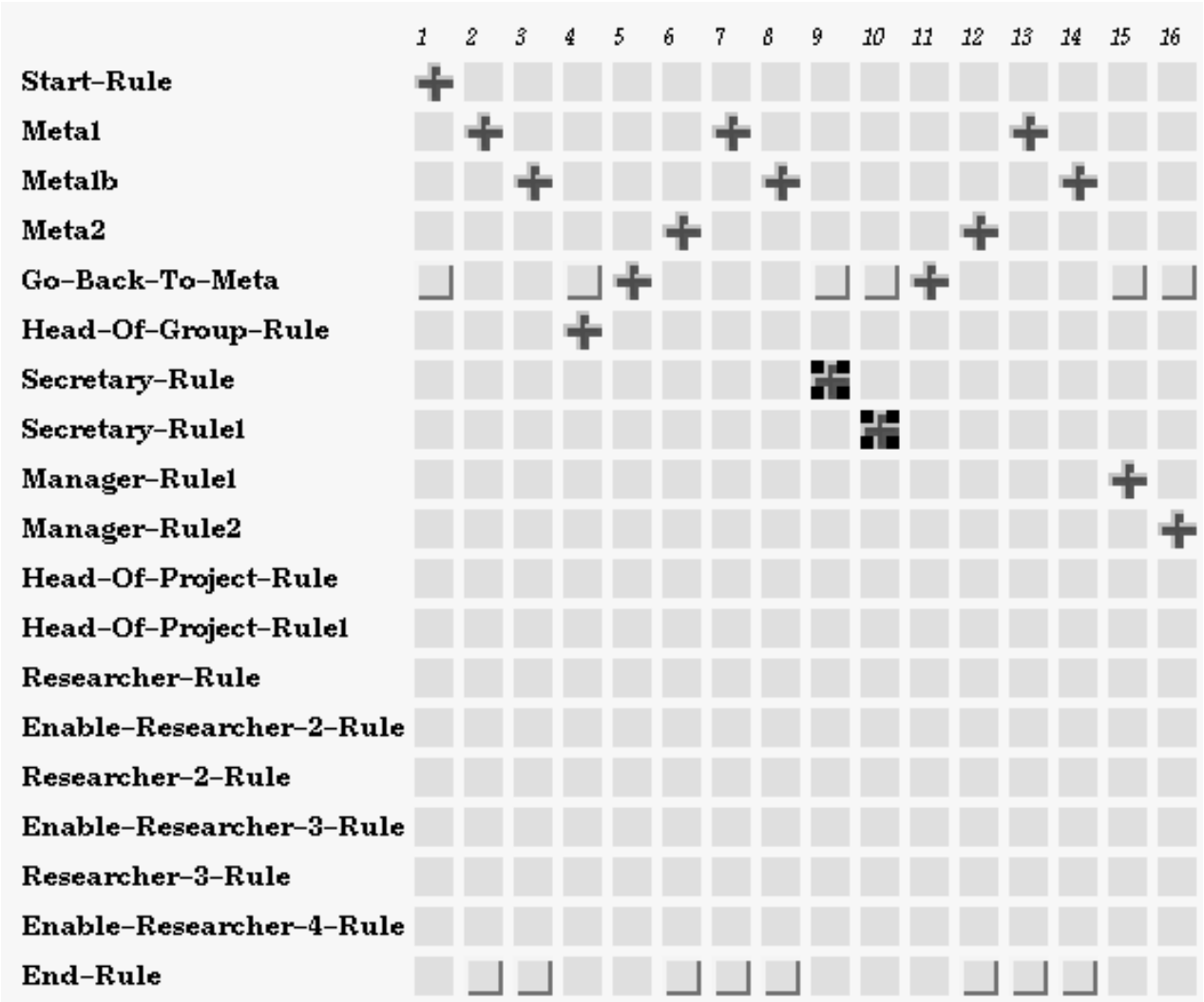


Figure 23. A trace of the Sisyphus rule execution with the suspect rule firings highlighted.

You can see in the figure 23 above (a small portion of the whole execution) that the rules are in two main groups: the control rules Start-Rule, Meta1, Meta1b Meta2 and Go-Back-To-Meta; and the role assignment rules: Head-Of-Group-Rule, Secretary-Rule, Secretary-Rule1 etc. The agent has signalled that the two secretary rules are 'suspect' by highlighting them. The role assignment rules correspond to the choose role, current role, choice segments of the design layer. The corresponding part of the design diagram would be highlighted.

The KE inspects the rule definition for the Secretary-Rule1 (by clicking on the highlighted icon for rule Secretary-Rule1 in figure 23).

© The Open University, 1993

```
(p  Secretary-Rule1
    {(context  secretary)  <context>}
    (close-to  <head-room>  <name>)
    (room  ^size  2  ^kb-name  <name>  ^available?  t  ^allocated?  nil)
    {(tmpsec  <sec1>  <sec2>)  <tmps>}
   -->
    (cl-call  vkr-prove
                ($list update-cm <sec1> <name> secretary))
    (make  rule-fired  ^context  secretary)
    (remove  <tmps>))
```

After checking that certain conditions are met (such as the fact that the chosen room is near to the head of group's room) Secretary-Rule1 updates the current model in the first clause of the right hand side of the rule. There is a bug, however, within this first clause. The second secretary <sec2> is missing. The clause should read:

```
(cl-call  vkr-prove
            ($list  update-cm  ($list  <sec1>  <sec2>)
                    <name>  secretary))
```

The agent was able to spot this inconsistency because the output of this rule (together with Secretary-Rule) was an atomic item <sec1> instead of a list. The definition of a current role (the input to the task Update Model), not shown in the design diagrams, is that it should be a list.

9. CONCLUSIONS

In this document we have presented the VITAL Bug Location Methodology. The methodology is based on the results and recommendations of two studies. The first study took the form of an in-depth self-report. This study revealed eight fundamental problems in the debugging facilities provided by HyperTalk's programming environment, and suggested six solutions. The second study analysed the debugging anecdotes collected from a world-wide email trawl. It confirmed that the results from the first study were representative of debugging in the 'real world'. The conclusions of the these two studies led to the formulation of two principles which underpin the VITAL Bug Location Methodology:

i) bug location is to be based on visualizations of the execution, and
ii) the relations between entities within an execution are to be computed by specialist bug location agents.

We then reported on the work we have carried out instantiating these two principles. We described Viz, a framework and software tool to describe and implement software visualization systems. The potential of the Viz framework has already been proved as it has been used to implement five visualization systems within VITAL.

We then described VITAL Bug Location Agents (VITAL-BLAs). VITAL-BLAs will take the burden from the KE to compute the indirect relationships between potentially distant points within an execution space.

© The Open University, 1993

The approach taken within the VITAL-BLM is *fail safe,* because the knowledge engineer will always get some sort of aid. If a VITAL-BLA detects a bug the knowledge engineer benefits from the new information which can then be acted on. Even if the VITAL-BLAs fail to find any information the knowledge engineer still benefits from the visualization.

© The Open University, 1993

REFERENCES

Adam, A. & Laurent, J. P. (1980) Automatic Diagnostics of Semantic Errors. Proceedings of the AISB-80 Conference on Artificial Intelligence

Adrion, W. R., Branstad, M.A., and Cherniavsky, J.C. (1982) Validation, Verification, and Testing of Computer Software, *ACM Computing Surveys*, **14** (**2**), pp. 159-192.

Anderson, J.R., Farrell, R., and Sauers, R. (1984) Learning to Program in LISP**.** *Cognitive Science*. **8**, pp. 87-129.

Ayel, M. (1988). Protocols for Consistency Checking in Expert System Knowledge Bases. Proceedings of European Conference on Artificial Intelligence, ECAI '88, Munich.

Baecker, R.M., & Sherman, D. (1981) Sorting Out Sorting. narrated colour videotape, 30 minutes, presented at ACM SIGGRAPH '81. Published by Morgan Kaufmann, Los Altos, CA.

du Boulay, J. B. H. (1986). Some Difficulties of Learning to Program. *Journal of Educational Computing Research*, **2** (**1**) pp. 57-73.

Brayshaw, M. & Eisenstadt, M. (1991). A Practical Graphical Tracer for Prolog. *International Journal of Man-Machine Studies*, **35** (5) pp. 597-631.

Brayshaw, M. (1991) An Architecture for Visualising the Execution of Parallel Logic Programming. Proceedings of the Twelth International Joint Conference on Artificial Intelligence (IJCAI-91). Los Angeles: Morgan Kaufmann.

Byrd, L. (1980) Understanding the Control Flow of Prolog Programs. In Proceedings of The 1980 Logic Programming Workshop, pp. 127-138.

Brotsky, D. (1981) Program Understanding through Cliche Recognition. MIT Artificial Intelligence Laboratory. Working Paper 224.

Brown, M.H. (1988) *Algorithm Animation*. ACM Distinguished Dissertations, MIT Press, New York.

Corkill, D.D., Gallagher, K.Q., & Murray, K.E. (1986) GBB: A Generic Blackboard Development System. Proceedings of AAAI'86, Philadelphia, USA.

Domingue, J. (1987). ITSY: An Automated Programming Advisor. *Doctoral Dissertation.* Human Cognition Research Laboratory, The Open University. (Also available as Tech. Rep. No. 22).

Domingue, J., & Eisenstadt, M. (1989) A New Metaphor for the Graphical Explanation of Forward Chaining Rule Execution. In Proceedings of The Twelvth International Joint Conference on Artificial Intelligence (Detroit, MI), pp. 129-134.

Domingue, J. (1992). An Automated Programming Advisor. In *Novice programming environments: explorations in human-computer interaction and artificial intelligence* (Eisenstadt, M., Keane, M. and Rajan, T. Eds.), London: Lawrence Erlbaum Associates. pp. 287-329.

Domingue, J., Price, B. A. & Eisenstadt, M. (1992). A Framework for Describing and Implementing Software Visualization Systems. The proceedings of Graphics Interface'92 Conference, Vancouver, Canada, pp. 53-60.

© The Open University, 1993

Domingue, J., Price, B. A. & Eisenstadt, M. (in press). Viz: A Framework for Describing and Implementing Software Visualization Systems. In *User-Centred Requirements for Software Engineering Environments*. (Gilmore, D. & Winder, R. Eds.).

Eisenstadt, M. (1984) A Powerful Prolog Trace Package. Proceedings of the Sixth European Conference on Artificial Intelligence (ECAI-84). Amsterdam: North-Holland.

Eisenstadt, M., & Brayshaw, M. (1988) The Transparent Prolog Machine (TPM): an execution model and graphical debugger for logic programming**.** *Journal of Logic Programming* **5** (**4**), pp. 1-66.

Eisenstadt, M., Price, B. A., & Domingue, J. (1993). Software Visualization As A Pedagogical Tool. *Instructional Science*, **21** pp. 335-365.

Eisenstadt, M., Domingue, J., Rajan, T., & Motta, E. (1990). Visual Knowledge Engineering. *IEEE Transactions on Software Engineering*, **16** (**10**) pp. 1164-1177.

Eisenstadt, M. (1993). Tales of Debugging from the Front Lines. Technical report No. 101, Human Cognition Research Laboratory, The Open University, Milton Keynes, U.K. (also submitted to Empirical Studies of Programmers V, Palo Alto, CA., December 1993).

Evertsz, (1990). The Role of the Crucial Experiment in Student Modelling. *Doctoral Dissertation,* IET The Open University, U.K.

Evertsz, (1991). The Automated Analysis of Rule-based Systems Based on their Procedural Semantics. Proceedings of teh 12th International Joint Conference on Artificial Intelligence, Sydney.

Evertsz, R., Motta, E. (1991) The Abstract Interpretation of Hybrid Rule/Frame-based Systems. In *Trends in Artificial Intelligence.* Lecture Notes in Artificial Intelligence. (Ardizzone, E., Gaglio, S., and Sorbello, F. Eds.), Springer Verlag.

Feiner, S., & Beshers, C. (1990) Worlds within Worlds: Metaphors for Exploring *n*-Dimensional Virtual Worlds. In Proceedings of The ACM SIGGRAPH Symposium on User Inteface Software and Technology (UIST) (Snowbird, UT, Oct. 3-5). ACM, New York, pp. 76-83.

Fickas, S. (1987). Supporting the Programmer of a Rule Based Language. *Expert Systems,* **4** (**2**), pp. 74-87.

Ginsberg, A. (1988) Knowledge-base reduction: A New Approach to Checking Knowledge Bases for Inconsistency and Redundancy. Proceedings of the 7th National Conference on Artificial Intelligence (AAAI 88), pp. 585-589.

Goldstein I. P. (1975) Summary of MYCROFT: A System for Understanding Simple Picture Programs. *Artificial Intelligence,* **6**.

Hastings, R. & Joyce, B. (1992) Purify: fast detection of memory leaks and access errors. Proceedings of the Winter Usenix Conference.

Henry, R.R., Whaley, K.M., & Forstall, B. (1990) The University of Washington Illustrating Compiler. In Proceedings of The ACM SIGPLAN '90 Conference on Programming Language Design and Implementation (White Plains, NY, June 20-22). ACM, New York, pp. 223-233.

Horwitz, S., Reps, T. & Binkley, D. (1990) Interprocedural Slicing Using Dependence Graphs. *ACM Transactions on Programming Languages and Systems,* **12** (**1**).

© The Open University, 1993

Jacob, R.J.K., & Froscher, J.N. (1988). Facilitating Change in Rule-based Systems. In *Expert Systems: The User Interface*. (Hendler, J. A. Ed.) Ablex Publishing Corp. Norwood, New Jersey.

Johnson, W. L. (1983). An Effective Bug Classification Scheme Must Take the Programmer into Account. In Proceedings of The Workshop on High-Level Debugging. Palo Alto, CA.

Johnson, W. L. & Soloway, E. (1985) PROUST: An automatic debugger for Pascal Programs. *BYTE: The Small Systems Journal.* **10** (**4**), pp. 179-190.

Jonker, W., Kontio, J., & Motta, E, (1991) Definition and Positioning of the VITAL Project. VITAL Project Report ID732.1, PTT Research, Groningen.

Katz, I. R. & Anderson, J. R. (1988) Debugging: An analysis of bug-location strategies. *Human Computer Interaction*, **3** (**4**) pp. 351-399.

Knuth, D. E. (1989) The Errors of TeX. *Software—Practice and Experience*, **19** (**7**) pp. 607-685.

Koopman, M. R. J., Spee, J. W., Jonker, W., Montero, L., O'Hara, K., Mepham, M., & Motta, E. (1991) *VITAL Conceptual Modelling*, VITAL Project Report DD213, PTT Research, Groningen.

Laubsch, J. & Eisenstadt, M. (1982) Using Temporal Abstraction to Understand Recursive Programs Involving Side Effects. Proceedings of the National Conference on Artificial Intelligence.

London R. L., & Duisberg R. A. (1985) Animating Programs using Smalltalk. *IEEE Computer,* **18** (**8**) pp. 67-71.

Linster, M. (1992) Sisyphus '92: Models of Problem Solving. GMD Technical Report.

Lutz, R. (1989) Chart Parsing of Flowgraphs. Proceedings of the 11th International Joint Conference on Artificial Intelligence. pp. 116-121.

Lutz, R. (1992) Plan Diagrams as a Basis for Understanding and Debugging Pascal Programs. In *Novice programming environments: explorations in human-computer interaction and artificial intelligence* (Eisenstadt, M., Keane, M. & Rajan, T.Eds.), London: Lawrence Erlbaum Associates, pp. 243-285.

Mackinlay, J. (1986) Automating the Design of Graphical Presentations of Relational Information. *ACM TOGS*. **5** (**2**), pp. 110-141.

Mackinlay, J.D., Robertson, G.G., & Card, S.K. (1991) The Perspective Wall: Detail and Context Smoothly Integrated. In Proceedings of CHI'91 (New Orleans, Louisiana, May). ACM, New York, pp. 173-179.

Motta, E., Stutt, A., O'Hara, K., Kuusela, J., Toivonen, H., Reichgelt, H., Watt, S., Aitken, S., & Verbeck, F. (1992) VITAL Knowledge Representation Language Specification: Final Deliverable. VITAL Project Technical Report, OU/DD412/W/1.

Motta, E., O'Hara, K., & Shadbolt, N. (in press) Grounding GDMs: A Structured Case Study. *Knowledge Acquisition Journal.*

Motta, E. (1993) Operationalizing the VITAL Conceptual Modelling Language. VITAL Project Technical Report VITAL/221.1/OU/D/2.

Murray, W. R. (1986) Automatic Program Debugging for Intelligent Tutoring Systems. *Doctoral Dissertation* Artificial Intelligence Laboratory, The University of Texas at Austin.

© The Open University, 1993

Myers, B.A. (1986) Visual Programming, Programming by Example, and Program Visualization: A Taxonomy. In Proceedings of CHI '86 Human Factors in Computing Systems (Boston, MA, April 13-17). ACM, New York, pp. 59-66.

Myers, B.A. (1990) Taxonomies of Visual Programming and Program Visualization. *Journal of Visual Languages and Computing* **1** (**1**), pp. 97-123.

Nguyen, T.A., Perkins, W.A., Laffey, T.J. & Pecora, D. (1985). Checking an expert system knowledge base for consistency and completeness. Proceedings of the 9th International Joint Conference on Artificial Intelligence. Los Angeles, CA.

Pain, H., & Bundy, A. (1987) What stories should we tell novice Prolog programmers? In R. Hawley, Ed. *Artificial Intelligence Programming Environments*. Wiley, New York, pp. 119-130.

Pennington, N. (1987). Stimulus structures and mental representations in expert comprehension of computer programs. *Cognitive Psychology*, **19**, pp. 295-341.

Perkins, W.A., Laffey, T.J., Pecora, D. & Nguyen, T.A. (1989). Knowledge base Verification, in *Topics in Expert Systems Design, Methodologies and Tools* (Guida, G., & Tasso, C., Eds.), North Holland, Amsterdam, pp. 353-376.

Poltreck, S.E., Steiner, D. D., & Tarlton, P. N. (1986) Graphic Interfaces for Knowledge-Based System Development. Proceedings ACM Conference on Computer Human Interaction.

Propp, V. (1968) *Morphology of the Folktale*. 2nd ed. (L.A. Wagner ed). University of Texas Press, Austin, TX.

Price, B.A., & Baecker, R.M. (1991) The Automatic Animation of Concurrent Programs. In Proceedings of The First International Workshop on Computer-Human Interfaces (Moscow, 5-9 August). ICSTI, Moscow, USSR, pp. 128-137.

Price, B.A., Baecker, R.M., Small, I. S. (in press) A Principled Taxonomy of Software Visualization. *Journal of Visual Languages and Computing.*

Rich, C., Shrobe, H. E., Waters, R. C., Sussman, G. J. & Hewitt, C. E. (1978) Programming Viewed as an Engineering Activity. A.I. Memo 459 MIT Artificial Intelligence Laboratory.

Rich, C. (1981) Inspection Methods in Programming. MIT Artificial Intelligence Laboratory, Report No. AI-TR-604.

Rousset, M-C. (1988) On the Consistency of Knowledge Bases: The COVADIS System. Proceedings of European Conference on Artificial Intelligence, ECAI '88, Munich.

Reubenstein, H.B. (1985). OPMAN: An OPS5 Rule Base Editing and Maintenance Package. *Master's Thesis*. MIT Department of Electrical Engineering and Computer Science.

Ruth, G. R. (1976) Intelligent Program Analysis. *Artificial Intelligence,* **7**.

Shapiro, D. G. (1981) Sniffer: A System that Understands Bugs. A.I. Memo No. 638 MIT Artificial Intelligence Laboratory.

Soloway, E., Bachant, J., & Jensen, K. (1987). Assessing the Maintainability of XCON-in-RIME: Coping with the Problems of a VERY Large Rule-Base. Proceedings of the Sixth National Conference on Artificial Intelligence, Seattle, Washington.

Soloway, E., & Ehrlich K. (1984) Empirical Studies of Programming Knowledge. *IEEE Transactions on Software Engineering*, **10** (**5**), pp. 595-609.

© The Open University, 1993

Spohrer, J. C., Soloway, E., & Pope, E. (1985). A Goal/Plan Analysis of Buggy Pascal Programs. *Human-Computer Interaction*, **1** (**2**), pp. 163-207.

Stasko, J.T. (1990) The Path-transition Paradigm: a practical methodology for adding animation to Program Interfaces**.** *Journal of Visual Languages and Computing*. **1** (**3**), pp. 213-236.

Tufte, E.R. (1983) *The Visual Display of Quantitative Information*. Graphics Press, Cheshire, CT.

Tufte, E.R. (1990) *Envisioning Information*. Graphics Press, Cheshire, CT.

Vesey, I. (1989). Toward a Theory of Computer Program Bugs: an Empirical Test. *International Journal of Man-Machine Studies*, **30** pp. 123-46.

Waters, R. C. (1978) Automatic Analysis of the Logical Structure of Programs. Technical Report No. TR-492.

Waters, R. C. (1979) A Method for Analysing Loop Programs. *IEEE Transactions on Software Engineering,* **5** (**3**) pp. 237-247.

Waters, R. C. (1982) The Programmer's Apprentice: Knowledge Based Program Editing. *IEEE Transactions on Software Engineering,* **8** (**1**).

Waters, R. C. (1985) KBEmacs: A Step Toward the Programmer's Apprentice. Technical Report 753 MIT Artificial Intelligence Laboratory.

Weiser, M. (1982) Programmers Use Slices When Debugging. *Communications of the ACM.* **25** (**7**), pp. 446-452.

Wertz, H. (1982) Stereotyped Program Debugging: an aid for novice programmers. *International Journal of Man-Machine Studies,* **16**, pp. 379-392.

Wills, L. M. (1990) Automated Program Recognition: A Feasibility Demonstration. *Artificial Intelligence,***45** pp. 113-171.

Winograd, T. (1975). Frame Representations and the Declarative/Procedural Controversy. In *Representation and Understanding, Studies in Cognitive Science*, (Bobrow, D. & Collins, A. Eds.), Academic Press, New York.

Zelinka, L. M. (1986) Automated Program Recognition *Master's Thesis* MIT Electrical Engineering and Computer Science.

© The Open University, 1993