```
 MMM
 MMMM       MMM
    MM     M MM
   M      M
  M      M     MMMMMMMMM
 MM     MM    MMMM    MMM
MMM     MM      MM    MMM
MMM    MMM      MM   MMM
MMMMMMMMMM       MMMMMMM          MM
 MMMMMMM MMMM    MMM   MM       MMMMM
          MMM    MM    MMM     M   MM
                 M     MMM    M    M
                M  MM  MMM   MM
            MMMM   MMMMMM   MMM
            MMM             MMM
                            MMM
                             MMM   M
                              MMMMM
```

```
**************************************
*                                    *
*          The Design of a           *
*  High-Level, Language-Independent  *
*     Symbolic Debugging System      *
*                                    *
**************************************
```

by

Mark Scott Johnson

Technical Report 76-10

December 1976

Department of Computer Science
University of British Columbia
Vancouver, B. C.

Abstract

The design of a language-independent, interactive system to facilitate the analysis and symbolic debugging of computer programs written in high-level languages is presented. The principal features of the system are: (1) host source language independence is supported by the abstraction of language entities and constructs with a language interfacer providing the system with language-dependent details, (2) translators can cooperate with the system at varying levels of detail, (3) the user interacts with the system and an executing object program thru an extendable debugging language, and (4) debugging primitive actions are kept to a minimum and nonprimitive actions are provided by user-supplied and library debugging procedures. The design criteria of such a system are presented, and a realization of such a system is illustrated by examples of debugging commands and procedures encoded in a debugging language.

Keywords and Phrases: symbolic debugging, debugging systems, interactive debugging, error detection, automated programming aids, program analysis, dynamic debugging, problem-oriented languages

CR Categories: 4.42, 4.22, 4.30

Table of Contents

1. Introduction 1

2. The Debugging System 2
   2.1. Design Criteria 3
   2.2. Basic Concepts 4
   2.3. The Debugging Language 7
   2.4. Examples 9

3 Conclusion 12

4 Acknowledgements 13

5. References 13

6 Appendix: DiSpeL Syntax Chart 15

1. Introduction

The computer program development cycle consists primarily of six phases: specification, functional decomposition, coding, testing, debugging, and evaluation. In recent years software engineers have focused much attention on the second and third of these phases. This is exemplified by the attention paid to language design and the advocacy of structured programming. Altho these two points are certainly important components in the production of quality software, their emphasis has distracted from progress in the other phases. Emphasizing the design of languages which make the inclusion of logical errors difficult and programming methodologies which encourage error-free programming is important. Nevertheless, it is still necessary for all programs to enter the testing and debugging phases.

This paper presents the design of a language-independent, interactive system to facilitate the analysis and symbolic debugging of computer programs written in high-level languages. The primary purpose of the debugging system is to provide an environment in which the user can detect the presence of errors and trace their cause and, thereby, to reduce the total time spent on the debugging phase of the program development cycle. The need for such a system is apparent from the proliferation of language processing systems which provide inadequate run-time debugging aids.

The concept of an interactive, run-time debugging system is not new. Before the invention of batch processing, debugging was carried out directly at the operator's console using the console switches and lights to provide clues as to why a program behaved incorrectly. After the development of batch processing, this activity was automated to produce memory and register dumps. In the early days of interactive computing, the value of an interactive debugging system became apparent and was first applied to the detection of errors in machine-language programs. Virtually every interactive computing environment provides some such aid [Bern 68, Evan 66, Gain 69, Sali 73].

The trend away from machine-language programming has emphasized the need to develop debugging tools which provide the high-level language user with the ability to monitor the execution of a program using source-level names and notations. One common way in which such aid has been provided is thru language processor supplied symbolic postmortem dumps (e.g., ALCOR ALGOL 60 [Baye 67] and ALGOL-W [Satt 72]).

A significant advance in high-level language debugging has been afforded by the implementation of special diagnostic translators. These translators sacrifice efficiency to provide extensive run-time checks to aid in the detection of logical program errors (e.g., subscripts out of range). Examples of such translators include PLUTO [Boul 72] and PL/C for PL/I programs, DITRAN [Moul 67] and WATFIV for FORTRAN programs, WATBOL for COBOL programs, and ALGOL-W [Site 71].

Another common way in which the high-level language user has been given access to debugging tools is thru extension to the host source language itself [Bair 75, Conw 73, Wolm 72].

Systems have also been developed to provide high-level language users with essentially the same facilities available to the machine-language user via low-level debugging systems. High-level interactive debugging systems have almost invariably been developed around one particular source language (e.g., MANTIS [Ashb 73] and FORTRAN, EXDAMS [Balz 69] and PL/I, the INTERLISP system [Bobr 72], IBM's PL/I Checkout Compiler [Cuff 72], and PL/I under Multics [Wolm 72]). Altho some of these systems have actually involved integrating debugging capabilities directly into an interpretive environment (e.g., INTERLISP and the PL/I Checkout Compiler), others have been designed explicitly as run-time systems manipulating translated code (e.g., EXDAMS and MANTIS). Nevertheless, in both cases a debugging environment has been established which is applicable to a single high-level language.

In view of the current state of interactive debugging, the advantages of a single system which is capable of dealing with programs written in various source languages should be obvious. A language-independent debugging system minimizes the duplication of effort needed in providing a debugging environment with the introduction of new programming languages. It also minimizes the user's overhead in learning a new debugging system for each new language. A language-independent environment allows collections of programs written in more than one source language to be debugged in a uniform manner.

2. The Debugging System

/* As a famous philosopher once almost said,
"Give me a suitable debugging environment and a
tool-building facility powerful (and simple) enough,
and I will debug the world."
Robert M. Balzer - [Balz 69], p. 567 */

The debugging system described in this paper is called RAIDE (Run-time Analysis and Interactive Debugging Environment), named after another product successfully employed to eliminate bugs from the user's environment.

2.1. Design Criteria

Many criteria have been taken into account in the design of RAIDE [Gain 69, Gris 71, Mann 73]. The most important of these is that the system should be language-independent over a large class of source languages. This virtually dictates that the system run using translated code since to provide a system which can interpret a broad class of source languages is currently infeasible.

While language-independence is an advance over previous debugging systems, this approach does have several disadvantages. Foremost among these is that run-time changes to correct the source program are virtually impossible in a noninterpretive environment. Also, using one or more of the host source languages to specify debugging actions may encourage confusion. It is thus desirable to provide a separate debugging language which the user must learn. A third disadvantage, which is inherent in any language-independent system, is that it may not always be possible to cater to the peculiarities of particular source languages, either existing or future.

Altho the debugging system should be language-independent, it should appear language-dependent from the user's point of view. For example, if an array bound is exceeded during program execution, RAIDE should respond with a message couched in the terminology of the source language. Thus, for ALGOL 68 the message "INDEX EXCEEDS THE BOUNDS SPECIFIED IN THE DEFINING OCCURRENCE OF THE MULTIPLE VALUE" might be produced.

Another goal is that RAIDE should be usable on multilingual collections of programs. Thus, the user can debug a set of programs written in more than one source language.

Another major design criterion is that the system should be oriented toward interactive processing, but it ought also to be usable in a batch processing environment. Obviously, many of the interactive features will be of marginal value from the batch stream. Nevertheless, there still exists a kernel of debugging facilities which are applicable to both environments.

Another major requirement is that all debugging should be done within terms of the source language(s). Knowledge of the underlying machine environment should be unnecessary.

One consequence of the preceding criterion is that language translators will need to supply the debugging system with substantial amounts of information concerning the source program. Data such as the identifier table, the type table, and even the source code itself will need to be provided. Nevertheless, it should be possible for translators to provide information in increasing layers of completeness. This will enable RAIDE to be used when the translators are not completely cooperating. If the user makes a request to which the system is unable to respond because of lack of translator-supplied information, RAIDE should return a message to that effect and allow the user to continue.

Another design criterion is that RAIDE should be capable of supplying extensive information concerning the state of program execution. Altho some analysis features (e.g., execution profiles) border on the domain of program testing, as opposed to program debugging, such facilities are needed in a powerful program debugging environment. Nevertheless, the debugging information supplied should never be overwhelming. That is, the user should only see what is relevant, with detailed information being provided upon a more precise request.

The kernel of the system should be minimal, yet sufficient. That is, the system must include a set of primitives sufficient to carry out all of the desired debugging actions; but this set should not contain primitives which can easily be simulated using a combination of the others. Some overlap may be necessary, however, in order to produce a set of easily usable primitives. This design criterion should result in minimizing the effort required to implement and transport the system.

Also, to minimize the implementation effort, the system should avoid duplicating resources providable by the host operating system. It is assumed the user will be familiar with the operating system so that alternative facilities built into the debugging system will merely be a source of confusion.

One final design criterion is that the user should not be required to make modifications to the source program in order to carry out debugging. It should be possible to specify at run-time everything the user may need to facilitate the debugging of a program. This should not, however, preclude the programmer from designing some debugging aids into the program since doing so is a desirable implementation strategy [Ledg 75].

In summary, the design criteria discussed above are enumerated in Table 1.

2.2. Basic Concepts

The user interfaces with RAIDE solely thru the debugging system language. Before it is possible to describe such a language, however, it is necessary to define some terms and to explain some basic RAIDE concepts which are reflected in the debugging system language itself.

A **program** is a collection of procedures which interact to perform one primary task. In other words, RAIDE is designed to debug a single program during one interactive session. This program may, however, consist of a main procedure and any number of subprocedures.

1. The system should be source language independent.
2. The user interface should be language-dependent.
3. The system should be usable on multilingual collections of programs.
4. The system should be interactively oriented, but usable in batch.
5. Debugging should be done symbolically in source language terms.
6. Translators should be allowed to supply information in successive layers of completeness.
7. The system should provide extensive analytic information at run-time.
8. Information supplied by the system should be concise and pertinent to the user's request.
9. A small, usable, and sufficient set of primitive actions should be supplied.
10. Operating system resources should not be duplicated.
11. No translation-time modifications to the source program should be necessary to carry out debugging.

Table 1. Debugging System Design Criteria

The concept which is basic to a proper understanding of RAIDE is the distinction between a specific and a generic. A **specific** is a reference to a particular entity in the user's program. For example, X might be one particular variable, 10 might be one particular constant, and X := 10 might be one particular statement. Thus, X, 10, and X := 10 are specifics. On the other hand, a **generic** refers to a set of entities within the user's program all of one homogeneous variety For example, VARIABLE is a reference to a class of entities of which X is one particular member. Similarly, CONSTANT and STATEMENT are examples of generics. Furthermore, it is convenient to divide generics into two classes. A **segment-generic** is a generic which refers to some executable segment of a user's program. For a block-structured language, typical segment-generics are PROCESS, PROCEDURE, BLOCK, and STATEMENT. A **data-generic** refers to a particular class of data which the user's program can manipulate. Thus, for a block-structured language, VARIABLE, RESULT, PARAMETER, and CONSTANT are examples of data-generics

Generics are host language dependent. The only presupposition which RAIDE makes concerning them is that there are two classes: segment and data. For each language to be interfaced to RAIDE, it is necessary to supply a set of generics. Table 2 contains examples of generics which might be defined for several well-known programming languages.

<u>language</u>	<u>segment-generics</u>	<u>data-generics</u>
ALGOL 68	PROCESS	VARIABLE
	PROGRAM	IDENTITY
	ROUTINE	DENOTATION
	CLAUSE	YIELD
	UNIT	
FORTRAN	PROGRAM	VARIABLE
	SUBROUTINE	ARGUMENT
	FUNCTION	ARRAY
	STATEMENT	CONSTANT
LISP 1.5	FORM	ATOM
	FUNCTION	ARGUMENT
		PROPERTY-LIST
SNOBOL4	FUNCTION	VARIABLE
	PREDICATE	ARGUMENT
	OPERATOR	LITERAL
	STATEMENT	KEYWORD

Table 2. Examples of Possible Generics for Various Languages

Related to the concept of a generic is that of an incident. An **incident** is some activity which is associated with a generic. The system defines both segment-incidents and data-incidents to correspond to segment- and data-generics. The activities which can be associated with the segment-generics are ENTRY and EXIT. Thus, it is possible to speak of PROCEDURE ENTRY or STATEMENT EXIT. Similarly, the activities associated with the data-generics are ACCESS and UPDATE for referencing and changing the value of an item of data. Unlike generics, incidents are fixed within RAIDE and are not specified by a language interfacer.

Another system concept is that of an event. An **event** is any occurrence which can cause the user's program to stop execution leaving RAIDE in the interactive request mode. Examples of possible events are pressing the attention interrupt key on the keyboard and changing the value of some variable. An event which is defined independently of a source program (e.g., ATTENTION-INTERRUPT and OVERFLOW) is called an **exception.** Other events are described by expressions (e.g., "_before_ foo UPDATE" to represent the event occurring immediately preceding an assignment to the variable 'foo'). A number of exceptions are predefined, but language-dependent exceptions can also be defined. For example, SNOBOL4 might define STATEMENT-FAILURE.

When the user is interacting directly with RAIDE, many possible actions can be requested. An **action** is any primitive operation which the system can perform. For example, _display_ is an action which causes information to be displayed to the user. A **deferred action** is any action which does not occur immediately upon its specification. Deferred actions allow the user to set traps within the program. Whenever the event associated with a deferred action occurs, the action itself is initiated by the system. The system maintains all deferred actions on the **deferred action list.**

Altho the basic actions of the system are fixed, system extendability is provided by debugging procedures. A **debugging procedure** is a subroutine written using the primitive actions of RAIDE as available thru the debugging system language. Thus, it is possible to provide the user with a library of debugging procedures which perform any of the standard debugging operations which are not provided as primitive actions in RAIDE. For example, there is no primitive RAIDE action to cause all procedure invocations to be traced altho it is possible to write a debugging procedure to accomplish this.

Once the basic concepts described above are understood, the RAIDE user should be capable of learning the system debugging language to effectively aid in the detection of programming errors.

2.3. _The Debugging Language_

In order for the user to communicate with RAIDE, it is necessary to provide a language embodying the primitive system actions and supporting future extensions of the debugging environment. The debugging language of RAIDE is called DiSpeL (_De_buggi_n_g _Spe_cification _L_anguage). Since it is not possible to present a complete description of DiSpeL here, only enough of the language is described to enable an understanding of the examples presented subsequently. The complete syntax of DiSpeL is outlined in the appendix using a syntax chart in the style of [Watt 74], and a detailed presentation of the language can be found in [John 76].

In the ensuing discussion, an outline of the syntax of DiSpeL is presented using a variation of the syntactic metalanguage Backus-Naur Form (BNF). Upper-case identifiers are used to represent language-dependent entities (e.g., generic names), incidents, exception names, and the names of system functions. Lower-case identifiers are used to represent program-dependent entities (e.g., specific names) and the names of debugging variables and procedures.

The fundamental syntactic entity of DiSpeL is an <utterance>.

```
<utterance> ::= <declaration> . |
                <definition> . |
                <command> .
```

The <utterance> is the basic unit of interactive input. Until the full-stop symbol (.) is encountered, the <utterance> is only checked for syntactic correctness; the full-stop initiates semantic interpretation.

A <declaration> specifies debugging variables, as opposed to user program variables.

```
<declaration> ::= integer <id-list> |
                  specific <id-list>
```

Integer variables are especially useful as counters and as flags. The declaration of specific variables can be used as a shorthand for identifying specifics.

A <definition> identifies a debugging procedure.

```
<definition> ::= define <procedure-id>
                    [ ( <declaration-list> ) ]
                    as <command>
```

The most important <utterance> of DiSpeL is the <command>.

```
<command> ::= [<when-clause>] <action>
<when-clause> ::= [<label-id> :] <when>
<when> ::= on <exception-list> |
           before <specific-list> |
           after <specific-list>
```

The <when-clause> of a <command> causes the associated <action> to become a deferred action. The <label-id> of the <when-clause> is used to remove the action from the deferred action list. For a deferred action, whenever the specified event occurs, the associated <action> is initiated.

There are three principal forms of the <when-clause>. The on <exception-list> form specifies that the associated <action> is to be initiated whenever one of a set of possible <excep-

tion>s occurs. For example, "on ATTENTION-INTERRUPT break" will cause the interactive request mode is to be entered (i.e., a break to occur) whenever the attention interrupt key is pressed. The before and after forms of the <when-clause> cause the associated <action> to be initiated before or after some particular incident occurs. For example,

after each STATEMENT in foo EXIT break.

sets a trap after each executable statement in the procedure 'foo'. The phrase "each STATEMENT in foo EXIT" is a <specific>; it pinpoints the setting of a trap.

The basic actions of RAIDE are specified in DiSpeL as follows.

```
<action> ::= <compound-action> |      <quit-action> |
             <break-action> |         <reference-action> |
             <call-action> |          <restore-action> |
             <cancel-action> |        <save-action> |
             <display-action> |       <set-action> |
             <execute-action> |       <skip-action> |
             <for-action> |           <system-action> |
             <if-action> |            <unexecute-action> |
             <input-action> |         <while-action>
```

Rather than describing the <action>s in succession, each will be explained as needed in the examples of the next section.

The omission of a trace primitive action, present in virtually all previous debugging systems, should be noted. In DiSpeL a trace primitive is unnecessary since it is implied by the syntax of a <command>. Tracing can be implemented using the <when-clause> and the <display-action>.

2.4. Examples

The examples presented here are designed to show the extent and power of RAIDE as well as to demonstrate DiSpeL. It is assumed that the host source language is block-structured and contains the segment-generics PROCEDURE, BLOCK, and STATEMENT and the data-generics VARIABLE and PARAMETER. Furthermore, the existence of three segment-generic related system functions is assumed. These functions are CURRENT_PROCEDURE, CURRENT_BLOCK, and CURRENT_STATEMENT; they yield specifics indicating the procedure, block, or statement most recently active when the function is invoked. Like the generics, these segment-generic related functions are language-dependent and must be supplied for each language interfaced to the system. The language-independent RAIDE system functions are described as needed below.

The examples are presented in the form: a description of the debugging request, the DiSpeL code corresponding to the request, and an explanation and comments concerning the code.

1. Change the value of the variable 'var' in the procedure 'foo' to the value of 'n'.

```
set var in foo to n.
```

This command is entered when execution of the program has been suspended and RAIDE is in the interactive request mode. The set action changes the value of some user program or debugging variable.

2. List the names and current values (if any) of all variables accessible to the currently executing procedure which have not been accessed more than 'n' times.

```
for each VARIABLE in CURRENT_PROCEDURE -> var do
  if #ACCESSES(var) <= n
  then display SKIP, var, " = ", VALUE(var).
```

The for action is a control structure allowing repetitive initiation of some <action>. #ACCESSES is a system function yielding the number of times which the indicated data-specific has been accessed during total program execution. SKIP is a system function causing the items following to be displayed starting on a new line. Notice that displaying the specific variable 'var' causes the source-level name of the variable indicated by the specific to be printed. The current value of some specific is obtained via the VALUE system function.

3. Write a debugging procedure to trace all subroutine calls of a procedure indicating the location of the call, the name of the subroutine called, and the names and values of all of its formal parameters.

```
define trace_proc (specific proc) as
  for each PROCEDURE in proc -> subr do
  begin
     before_subr_entry_trace:
     before subr ENTRY
        display SKIP, "trace at statement ", CURRENT_STATEMENT,
            " in ", CURRENT_PROCEDURE ;
     after_subr_entry_trace:
     after subr ENTRY
     begin
        display SKIP, CURRENT_PROCEDURE,
           " entered with the following parameters:", SKIP ;
        for each PARAMETER in CURRENT_PROCEDURE -> parm do
           display parm, " = ", VALUE(parm)
     end
  end.
```

This example demonstrates a procedure of sufficient utility to merit inclusion within a debugging library. The body of the for action establishes two deferred actions, one which is initiated in the environment of the calling procedure (before subr ENTRY) and one which is initiated in the environment of the called subroutine (after subr ENTRY). Notice how the procedure is capable of setting many traps, all of which are identified by only two labels.

4. Write a debugging procedure to produce the ALGOL-W postmortem dump [Site 71 (pp. 125-127)].

```
define postmortem as
begin
  specific segment, caller ;
  display SKIP(3), "=> postmortem dump of active segments" ;
  set segment to CURRENT_BLOCK ;

  while DEFINED(segment) do
  begin
     display SKIP(2), "=> segment name: ", segment, SKIP(2),
        " value of local variables:", SKIP ;
     for each PARAMETER in segment -> parm do
        call print_parameter_value(parm) ;
     for each VARIABLE in segment -> var do
        call print_variable_value(var) ;
     set caller to CALLER(segment) ;
     if DEFINED(caller)
     then display SKIP(2), segment, " was activated from ",
        caller, ", near coordinate ",
        CURRENT_STATEMENT in caller ;
     set segment to caller
  end ;

  display SKIP(2), "=> end of postmortem dump", SKIP
end.
```

The while action is another control structure allowing repetitive initiation of some <action>. Using it and appropriate system functions, it is possible to trace back thru execution of the program. The system function DEFINED yields a true value if the variable indicated has a value. CALLER is a system function which accepts a segment-specific as an argument and yields a specific indicating the segment which called the argument specific. The procedure above assumes two debugging procedures 'print_parameter_value' and 'print_variable_value' have been defined elsewhere.

The examples above should give a feeling for DiSpeL and for the depth and breadth of RAIDE itself.

3. Conclusion

The principal attributes of RAIDE which distinguish it from previous debugging systems are summarized here.

1. The kernel of RAIDE contains a small set of primitives sufficient to implement all the traditional debugging actions. Unlike previous debugging systems which have been designed rather haphazardly, RAIDE's design is based upon the concept of minimum sufficiency.

2. The traditional debugging primitives (e.g., traces, dumps, and traps), have been generalized in RAIDE. An example of this generalization is the lack of a primitive trace action. All traditional debugging aids are available to the user thru debugging procedures. This generalization of debugging concepts should allow for the easy inclusion of future debugging aids.

3. RAIDE is one of the few debugging system to have language independence as a primary design criterion Virtually all previous system have been language-dependent by design or implementation.

4. The RAIDE debugging language is more extensive and orthogonal than that of any previous debugging system. DiSpeL represents a compromise between an interactive command language and a special-purpose programming language.

5. RAIDE potentially provides more run-time and analysis debugging information than any previous system. It has been designed to filter, not mask, this information so that the user can obtain maximum benefit from the debugging environment.

6. RAIDE is one of the few systems which can be used to debug multilingual collections of programs. Several preceding systems have provided an interface to machine-language subroutines; RAIDE enables subroutines to be written in any high-level language for which an interface has been provided.

A subset of RAIDE is currently being implemented at the University of British Columbia by the author. A more detailed description of the system and its implementation can be found in [John 76].

4. Acknowledgements

The author acknowledges with gratitude the comments, criticisms, and encouragements of his supervisor Dr. Harvey Abramson and of his fellow graduate students Greg Wilbur, Ted Venema, and Bill Appelbe.

5. References

[Ashb 73] Ashby, G., Salmonson, L., and Heilman, R. Design of an interactive debugger for FORTRAN: MANTIS. Software - Practice and Experience, 3:1 (1973 Jan.-March), 65-74.

[Bair 75] Baird, G.N. Program debugging using COBOL '74. Proc. AFIPS NCC, vol. 44 (1975), 313-318.

[Balz 69] Balzer, R.M. EXDAMS -- EXtendable Debugging and Monitoring System. Proc. AFIPS SJCC, vol. 34 (1969), 567-580.

[Baye 67] Bayer, R., Gries, D., Paul, M., and Wiehle, H.R. The ALCOR Illinois 7090/7094 post mortem dump. Comm. ACM, 10:12 (1967 Dec.), 804-808.

[Bern 68] Bernstein, W.A., and Owens, J.T. Debugging in a time-sharing environment. Proc. AFIPS FJCC, vol. 33, pt. 1 (1968), 7-14.

[Bobr 72] Bobrow, D.G. Requirements for advanced programming systems for list processing. Comm. ACM, 15:7 (1972 July), 618-627.

[Boul 72] Boulton, P.I.P., and Jeanes, D.L. The structure and performance of PLUTO, a teaching oriented PL/I compiler system. INFOR, 10:2 (1972 June), 140-153.

[Conw 73] Conway, R.W., and Wilcox, T.R. Design and implementation of a diagnostic compiler for PL/I. Comm. ACM, 16:3 (1973 March), 169-179.

[Cuff 72] Cuff, R.N. A conversational compiler for full PL/I. Computer J., 15:2 (1972 May), 99-104.

[Evan 66] Evans, T.G., and Darley, D.L. On-line debugging techniques: a survey. Proc. AFIPS FJCC, vol. 29 (1966), 37-50.

[Gain 69] Gaines, R.S. The Debugging of Computer Programs. Ph.D. Th., Dept. of Elec. Engr., Princeton U., 1969 Aug. 170pp.

[Gris 71] Grishman, R. Criteria for a debugging language. In [Rust 71], pp. 57-75.

[John 76] Johnson, M.S. "The Design and Implementation of a Run-Time Analysis and Interactive Debugging Environment (RAIDE)". Ph.D. Th. Draft, Dept of Comp. Sci., U. Of British Columbia, 1976 Nov. 80pp.

[Ledg 75] Ledgard, H.F. Programming Proverbs. Hayden Book Co., 1975. 134pp.

[Mann 73] Mann, G.A. A survey of debug systems. Honeywell Computer J., 7:3 (1973), 182-198.

[Moul 67] Moulton, P.G., and Muller, M.E. DITRAN -- a compiler emphasizing diagnostics. Comm. ACM, 10:1 (1967 Jan.), 45-52.

[Rust 71] Rustin, R. (editor) Debugging Techniques in Large Systems. Prentice-Hall, 1971. 148pp

[Sali 73] Salisbury, R. "The Symbolic Debugging System". Computing Centre, U. of British Columbia, 1973 Jan. 53pp.

[Satt 72] Satterthwaite, E.H. Debugging tools for high level languages. Software - Practice and Experience, 2:3 (1972 July-Sept.), 197-217.

[Site 71] Sites, R.L. "ALGOL-W Reference Manual". Tech. Rep. STAN-CS-71-230, Comp. Sci. Dept., Stanford U., 1971 Aug. 169pp

[Watt 74] Watt, J.M., Peck, J.E.L., and Sintzoff, M. Revised ALGOL 68 syntax chart. SIGPLAN Notices, 9:7 (1974 July), 39.

[Wolm 72] Wolman, B.I Debugging PL/I programs in the Multics environment. Proc. AFIPS FJCC, vol. 41, pt. 1 (1972), 507-514.

6. Appendix: DiSpeL Syntax Chart

```
┌─────────┐
|utterance|
└─┬───────┘
  |
  |  ┌───────────┐
  ├─>|explanation|  explain  keyphrase  .
  |  └───────────┘
  |
  |  ┌───────┐
  ├─>|inquiry|  inquire  sentence  .
  |  └───────┘
  |
  |  ┌───────────┐
  ├─>|declaration|
  |  └─┬─────────┘  ┌        ┐
  |    |            |integer |
  |    └───────────>|generic |  id|,  .
  |                 |specific|
  |                 └        ┘
  |
  |  ┌──────────┐
  ├─>|definition|  define  id  (  declaration|;  )  as ─┐
  |  └──────────┘               +-----------------+     |
  |      ┌──────────────────────────────────────────────┘
  |      |  ┌                ┐
  |      |  | command        |
  |      └─>| ┌──────────────┐|  .
  |         | |procedure-body||
  |         | └─┬────────────┘|
  |         └   |             ┘
  |             └─> begin  declaration|;  ;  command|;  end
  |                        +--------------+
  |
  |  ┌───────┐            ┌────┐
  └─>|command|  id  :     |when|  action  .
     └───────┘  +---+     └─┬──┘
                +-----------+--+  ┌                    ┐
                            └────>|when  condition     |
                                  |on  exception|,     |
                                  |before  specific|,  |
                                  |after  specific|,   |
                                  └                    ┘
```

```
┌──────┐
|action|
└─┬────┘
  |
  ├─> begin  command|;  end
  |
  ├─> break  message
  |          +-----+
  |
  ├─> call  id  ( expression|, )
  |             +---------------+
  |
  |                ┌          ┐
  |                |id|,      |
  ├─> cancel       |integer|, |
  |                |everything|
  |                └          ┘
  |                +----------+
  |
  |                 ┌────┐
  ├─> display     {|item|  as  type}|,  on  file-name
  |                └─┬──┘  +------+     +-----------+
  |                  |   ┌          ┐
  |                  |   |specific  |
  |                  └─> |message   |
  |                      |expression|
  |                      |debug-item|
  |                      └          ┘
  |
  |                 ┌                          ┐
  |                 |expression  segment-generic|
  ├─> execute       |            +-------------+|
  |                 |until  condition           |
  |                 |while  condition           |
  |                 └                          ┘
  |                 +--------------------------+
  |
  ├─> for  specific|,  ->  id  do  action
  |
  ├─> if  condition  then  action  else  action
  |                                +----------+
  |
  ├─> input  file-name
  |          +-------+
  |
  ├─> quit  message
  |         +-----+
  |
  ├─> reference  segment-designation
  |              +-----------------+
  |
  ├─> restore  id  saving  id
  |                +--------+
  ▼
```

```
┌-------------------┐
|action (continued)|
└┬------------------┘
 |
 ├--> save  id
 |
 |       ┌                              ┐
 |       |variable  to  expression      |
 ├--> set |id  to  segment-designation|,|
 |       |id  to  specific-expression   |
 |       └                              ┘
 |
 ├--> skip  segment-generic
 |          +-------------+
 |
 ├--> system  system-command
 |            +------------+
 |
 |                 ┌                                ┐
 |                 |expression   segment-generic|
 ├--> unexecute    |             +-------------+|
 |                 |until  condition              |
 |                 |while  condition              |
 |                 └                                ┘
 |                 +------------------------------+
 |
 └--> while  condition  do  action


┌---------┐
|specific|
└┬--------┘
 |
 └--> each  variable   ┌----------------┐
      +--+             |generic-incident|
                       └┬---------------┘
                       +-+---------------+
                         |  ┌      ┐
                         |  |entry |
                         └->|exit  |
                            |access|
                            |update|
                            └      ┘
```

```
┌──────────┐
|variable|
└─┬────────┘
  |
  └─> generic  :  ┌────────────────────┐  ┌─────────────────┐
      +--------+  |unqualified-variable|  |segment-qualifier|
                  └────────────────────┘  └─┬───────────────┘
                     |                     +-+---------------+
                     |                  ┌────┘
                     |                  |
                     |                  └─> in  seqment-designation
                     |
                     |
                     └─> {id  (  expression|,  )}|
                              +-----------------+


┌─────────────────────┐
|segment-designation|
└─┬───────────────────┘
  |                 ┌                     ┐
  |                 |unqualified-variable|
  └─> generic  :    |┌─────────────┐      |  segment-qualifier
      +--------+    ||segment-range|      |  +---------------+
                    |└─┬───────────┘      |
                    └  |                  ┘
                       |
                       |  ┌        ┐       ┌        ┐
                       └─>|integer|  ..   |integer|
                          |id      |       |id      |
                          └        ┘       └        ┘
                                    +-----------+
```

Syntax chart notation:

```
┌──────────────────────────────┬──────────────────────────────┐
|                              |                              |
| ┌─┐                          | ┌ ┐                          |
| |A|  B   A is defined as     | |A|     choose one of A, B,  |
| └┬┘      either B, C, or D   | |B|     or C                 |
|  ├─> C                       | |C|                          |
|  └─> D                       | └ ┘                          |
|                              |                              |
├──────────────────────────────┼──────────────────────────────┤
|                              |                              |
| A|B      A, ABA, ABABA, ...  | abc     abc is optional      |
|                              | +-+                          |
|                              |                              |
├──────────────────────────────┴──────────────────────────────┤
|                                                             |
| {A B}    treat A and B as one construct                     |
|                                                             |
└─────────────────────────────────────────────────────────────┘
```