MASTERSCOPE

Volume 1 , Number 1 February 1985

EDITORS' MESSAGE

Welcome to the inaugural issue of Masterscope, the newsletter for the Xerox Interlisp Users' Group. The objective of this newsletter is to provide a forum for you, our users, to discuss problems, solutions, and the projects that you are involved in.

This is your newsletter. The content will depend almost totally on articles and information supplied by the users of Interlisp-D. Masterscope will also contain announcements of, and reports about, national and local Users' Group meetings. We envision that Masterscope will become an externally (non-Xerox) supported and published newsletter, with Xerox acting in an advisory capacity. However, during the startup period, the Xerox AIS BU will act as editor and will cover the expense of reproduction and distribution.

We are actively soliciting submissions for future issues. Submit articles and other items for publication, as well as requests to be added to the mailing list, as follows:

INTERLISP ARTICLES:

AINewsletter↑.pasa@Xerox

OR: US Mail

AINEWSLETTER
ms 1232
Xerox Special Information Systems
250 North Halstead Street
Pasadena, California 91109

LOOPS ARTICLES:

Hausladen.PA@Xerox

OR: US Mail

Mary A. Hausladen
Xerox AI Systems
3333 Coyote Hill Road
Palo Alto, California 94304

The deadline for submission for the next issue of Masterscope is April 1. We welcome any contributions, especially short descriptions of your projects or packages you may want to provide to the community. It would make our lives much easier if your submissions were in TEdit format, from the Harmony release or later. We do reserve the right to make editoral changes for the purposes of improved clarity and presentation.

We are all looking forward to an active users' group.

The Editors

INTERLISP-D

Articles

GRAPHCALLS: a new Lispusers package

by
Christopher Lane
Stanford University

GRAPHCALLS is an extended graphical interface to the Interlisp CALLS function. It is to CALLS what BROWSER is to SHOW PATHS in MASTERSCOPE. It allows fast graphing of the calling hierarchy of both interpreted and compiled code, which allows examination of both user and system functions. In addition, the functions do not have to be analyzed by MASTERSCOPE first.

Buttoning a function on the graph will bring up a menu of operation that can be performed with respect to the function, such as editing, inspecting and further graphing. Functions which call no other functions or are undefined are printed in a bold version of the graph's font indicating that they cannot be graphed further. It is also possible to exclude specific functions or classes of functions. This is referred to as filtering. Interesting filters are WHEREIS, SYSLOADed files and EXPRP. WHEREIS, limits the tree to functions the user has loaded and prunes out Interlisp functions. SYSLOADed files and EXPRP limit the tree to interpreted functions. This is useful for graphing functions in the development stage.

GRAPHCALLS uses LAYOUTGRAPH in GRAPHER and can be any format specification. In the forest format, multiple instances of a function appear on the graph after every calling function, and a boxed node indicates the function appears elsewhere on the graph; possibly graphed further. In the lattice format, each function is placed on the graph only once and boxed nodes indicate recursive functions calls.

GRAPHCALLS also does dynamic graphing. GRAPHCALLS advises all of the functions on the graph (in the context of their parent) to invert their corresponding node on the graph (as well as delay some to allow it to be seen) and/or follow each function name by a count of the number of times it has been executed. In invert mode, a node remains inverted as long as control is inside its corresponding function. It returns to normal when the function is exited. The lattice format is best when using the invert feature. Closing the graph window UNADVISEs the functions on the graph.

You can, at some risk, interactively BREAK and EDIT a function on the graph while the code is executing. Also, creating subgraphs of advised graphs will show the generated advice functions not the original functions called, as will creating new graphs of functions in advised graphs. You can create advised graphs of functions already graphed normally on the screen.

GRAPHCALLS is being distributed by Xerox.

Bugs, Workarounds And Helpful Hints

- **Global, Special, and Local Variable Declaration Use**

Interlisp provides facilities for declaring the type of references that will be made to variables. Variables may be declared to be global, special or local by using the declarations SPECVARS, LOCALVARS, and GLOBALVARS.

SPECVARS vs **LOCALVARS** is a distinction that applies to the binding of a variable, e.g., in an argument list for a **PROG**, **LET** or **LAMBDA**. It says whether the variable is visible to free variable lookup. The default is that all variables are visible, e.g., are **SPECVARS**. (Now, why are the "special" variables the default ones? Well, because that's how Lisp 1.5 named it.)

On the other hand, **GLOBALVARS** affects the compilation of a reference to a free variable. Normally, if you just refer to variable "**FOOMUMBLE**" in a program that doesn't bind **FOOMUMBLE**, the instruction that references **FOOMUMBLE** will look up the stack looking for some binding of **FOOMUMBLE** (e.g., as a **SPECVAR**), and, when it gets all the way to the top, will stop and use the **GLOBAL VALUE**. If you declare a variable a **GLOBALVAR**, you are asserting that it is ***never*** bound, and thus all that lookup is unnecessary.

There's not much advantage in Interlisp-D to declaring things **LOCALVARS** instead of **SPECVARS**, although there were in Interlisp-10. There *is* a lot of advantage in declaring things as **GLOBALVARS** if they really are.

From within a function, access to the variables of the argument list or internal **PROG** or **LAMBDA** variables takes the same amount of time, whether or not the variable is declared **LOCALVAR** or **SPECVAR**.

For example, in

```
(LAMBDA (X Y Z)
 (PROG (A B C)
   ...computation involvin  X Y Z A B C ...]
```

The time to access **X Y Z** or **A B C** is the same whether they are declared **LOCALVARS** or **SPECVARS**: access uses the **IVAR** instruction for arguments and the **PVAR** instruction for prog variables, independent of the declarations. **IVAR** and **PVAR** each take 1.644 microseconds on an 1108.

The only minor help from declaring variables as **LOCALVARS** comes from the following factors:

a) if variables are declared **LOCALVARS**, then the free variable lookup for ***other*** variables doesn't have to examine their binding.

b) likewise, at least for internal **PROG** variables, declaring the variables **LOCALVARS** means that the compiler doesn't need to save and the loader doesn't need to load the name of the variable. It makes the compiled code a little smaller, a little faster to load, and uses fewer symbols.

c) in some situations, the compiler can optimize away a variable that is declared a **LOCALVAR** where it can't optimize it away if it is a **SPECVAR**.

These three advantages are relatively small: we've not seen any program where they make more than a 10-20% difference in space or speed.

On the other hand, we have seen programs where supplying a **GLOBALVARS** declaration has made it run 10 times faster. The best thing, of course, is to avoid global variables, since they often represent bad programming practice.

• MASTERSCOPE CALLS Semantics

When using the MASTERSCOPE relation **CALLS** one will see the expression **(RETFROM 'MUMBLE)** as a call to **MUMBLE**. This is because the verb **CALL** searches for all occurances where the given expression is used as a function name. Consequently **(FUNCTION MUMBLE)** also will be a call to the function **MUMBLE**. If you want only the usual notion of a function call you should use the MASTERSCOPE phrase **CALL DIRECTLY**.

• FILECOMS Execution and Compiled Files

When a compiled file is created, all of the compiled function definitions are collected and placed at the beginning of the file. In certain cases the file package commands to make a file may mix function definitions and other commands in a ordered way such as:

```
[(P (PRINTOUT T "Defining X"))
 (FNS X)]
```

In this case when the source file with the above commands is loaded Defining X will be printed first, then the function X will be defined. However, when the compiled file is loaded, X will be defined before the message is printed.

In order to always print the message first the file package form **DECLARE**: must be used with the tag **FIRST**. Thus:

```
[(DECLARE: FIRST (P (PRINTOUT T "Defining X")))
 (FNS X)]
```

will always print the message before the function definition.

• ZEROP Function

The function ZEROP checks to see if its argument is EQ to the integer 0. Thus the following are equivalent:

(ZEROP n) and (EQ n 0)

Note that if n is a floating point number, **ZEROP** will return **NIL** instead of T in all cases (even if the **n** is 0). The solution for this problem is to use **(EQP n 0)** instead of **(ZEROP n)**.

[**NOTE**: In the coming Intermetzo release of Interlisp-D the semantics of **ZEROP** will be changed so that **(ZEROP n)** gives the same result as **(EQP n 0)**]

• Generic vs. Specific Numeric Functions

In Interlisp there is no speed advantage in using specific numeric functions over generic numeric functions. Thus, in writing code there is no advantage to using IPLUS or FPLUS over just plain PLUS in execution speed. This is because the microcode checks all of the cases involved in parallel.

• Numeric Operation Overflow and Division by Zero

When Interlisp-D first comes up, the user is not protected from numeric operation overflow (one case of which is division by zero). Thus:

```
(QUOTIENT 5 0)            ==>  0
(QUOTIENT 5.5 0)          ==>  3.402823E38
(QUOTIENT 5E-10 5E40)     ==>  0
```

When floating point numbers underflow they go to 0. When floating point numbers overflow they go to **MAX.FLOAT**. When integers overflow they go to 0. No errors are generated.

By calling the function **OVERFLOW**, the error condition handling of the system can be modified. Executing **(OVERFLOW T)** will turn on error checking in all of the above cases. While it is not documented in the Interlisp Reference Manual. **OVERFLOW** works for both integer and floating operations.

- **CAR and CDR of non-lists**

The value of the litatom CAR/CDRERR controls what action lisp takes when CAR or CDR attempt to operate on non-list objects.

(Also, (SETQ CAR/CDRERR 'CDR) will catch infinite FMEMB loops... there are a few places in the window break package that break when it is set to anything other than N/L, however.)

- **Fonts and Special Characters**

Fonts generally provide printed representations for characters up to about 1778. It is possible to create representations for characters up to 2778. These characters may be then used just as one uses any of the normal character set. Tools for creation of your own font are found in the Lispusers package **EDITFONT**.

Once defined, these characters may be accessed from the keyboard in a number of ways. Executing **(METASHIFT T)** turns the **STOP** key (bottom blank key on machines other than the 1108) into a **META** key. Holding this key down adds 2008 to any key struck while it is down. A second option is to modify the keyaction table by calling the function **KEYACTION**. Using **KEYACTION** you can get any key on the keyboard to generate any character code. (see Interlisp Reference Manual page 18.8). Another possibility is to use the **TEDIT** abbreviation facility to insert your special characters (see Interlisp Reference Manual, page 20.31).

LOOPS

Articles

LOOPS Use At Ohio State

The AI Group at Ohio State University (OSU) is currently making heavy use of LOOPS in a number of projects. From the top level, there are two important reasons we have found LOOPS to be most useful in our system building efforts.

First, the object oriented programming style that LOOPS supports is a very good match to the theoretical paradigm being followed at Ohio State. We tend to factor the world into active agents that are then allowed to interact with one another in well constrained ways. Of course, this is only the barest starting point in our efforts, but that starting point leads to natural implementation expression in the object-oriented style.

Second, and more importantly, the programming environment supported in INTERLISP/LOOPS, with its heavy emphasis on graphic support, allows for effective management of our "world of objects".

Described below are several of OSU/AI's current interests. For general information about our group activities contact Prof. B. Chandrasekaran 614/422-0923 or CHANDRASEKARAN@RUTGERS. For further information about how our group is making use of LOOPS, Mr. Jon Sticklen 614/422-1413 or STICKLEN@RUTGERS) should be contacted.

CSRL: A Language for Designing Diagnostic Expert Systems

by
Tom Bylander, B. Chandrasekaran,
Sanjay Mittal, and Jack W. Smith
Ohio State University

Through our exploration of the medical domain, we have noted on many occasions the need to have expressive high level languages which will support the Ohio State University (OSU) paradigm. To that end we have developed one high level language/environment that supports our view of the diagnostic enterprise. CSRL, as described below, is to the OSU paradigm for diagnostic problem solving what the EMYCIN language is for the MYCIN problem solving systems.

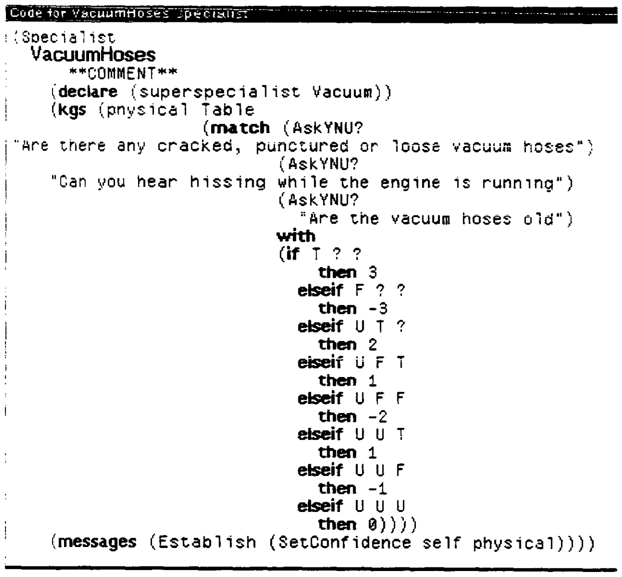
Many kinds of problem solving for expert systems have been proposed within the AI community. Whatever the approach, there is a need to acquire the knowledge in a given domain and implement it in the spirit of the problem solving paradigm. Reducing the time to implement a system usually involves the creation of a high level language which reflects the intended method of problem solving. For example, EMYCIN was created for building systems based on MYCIN-like problem solving. Such languages are also intended to speed up the knowledge acquisition process by allowing domain experts to input knowledge in a form close to their conceptual level. Another goal is to make it easier to enforce consistency between the expert's knowledge and its implementation.

CSRL (Conceptual Structures Representation Language) is a language for implementing expert diagnostic systems that are based on approach to diagnostic problem solving which has been developed by our AI group. In this approach, diagnostic reasoning is one of several generic tasks, each of which calls for a particular organizational and problem solving structure. This approach is an outgrowth of our group's experience with MDX, a medical diagnostic program, and with applying MDX-like problem solving to other medical and non-medical domains.

A diagnostic structure is composed of a collection of specialists, each of which corresponds to a potential hypothesis about the current case. They are organized as a classification or diagnostic hierarchy, e.g., a classification of diseases. A top-down strategy called establish-refine is used in which, either a specialist establishes and then refines itself, or the specialist rejects itself, pruning the hierarchy that it heads.

CSRL facilitates the development of diagnostic systems by supporting constructs which represent diagnostic knowledge at appropriate levels of abstraction. Message procedures describe the specialist's behavior in response to messages from other specialists. Knowledge groups determine how data relate to features of the hypothesis. Rule-like knowledge is contained within knowledge groups.

We have used CSRL in the implementation of two expert systems, both of which are discussed later in this report. Auto-Mech is an expert system which diagnoses fuel problems in automobile engines. It consists of 34 specialists in a hierarchy which varies from 4 to 6 levels deep. Red is an expert system whose domain is red blood cell antibody identification. CSRL is used to implement specialists corresponding to each antibody that Red knows about (around 30 of the most common ones) and to each antibody subtype.



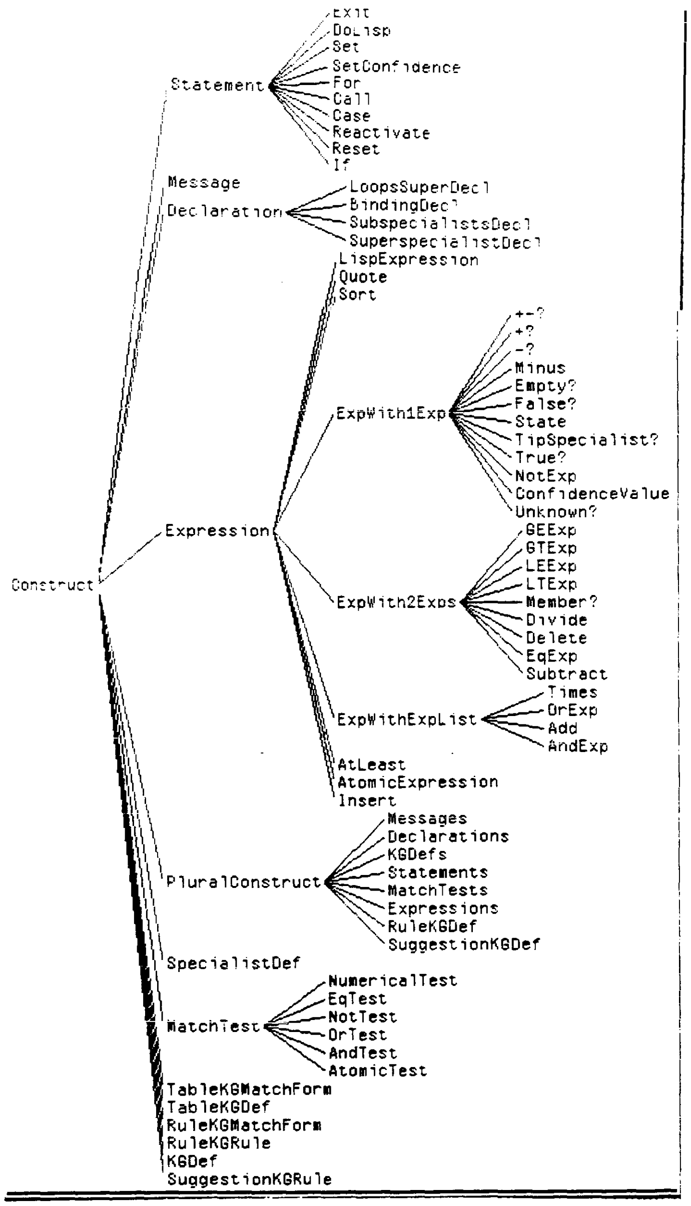
Code for VacuumHoses Specialist

```
(Specialist
  VacuumHoses
      **COMMENT**
    (declare (superspecialist Vacuum))
    (kgs (pnysical Table
                       (match (AskYNU?
"Are there any cracked, punctured or loose vacuum hoses")
                              (AskYNU?
    "Can you hear hissing while the engine is running")
                              (AskYNU?
                                "Are the vacuum hoses old")
                              with
                              (if T ? ?
                                 then 3
                                elseif F ? ?
                                 then -3
                                elseif U T ?
                                 then 2
                                elseif U F T
                                 then 1
                                elseif U F F
                                 then -2
                                elseif U U T
                                 then 1
                                elseif U U F
                                 then -1
                                elseif U U U
                                 then 0))))
    (messages (Establish (SetConfidence self physical))))
```

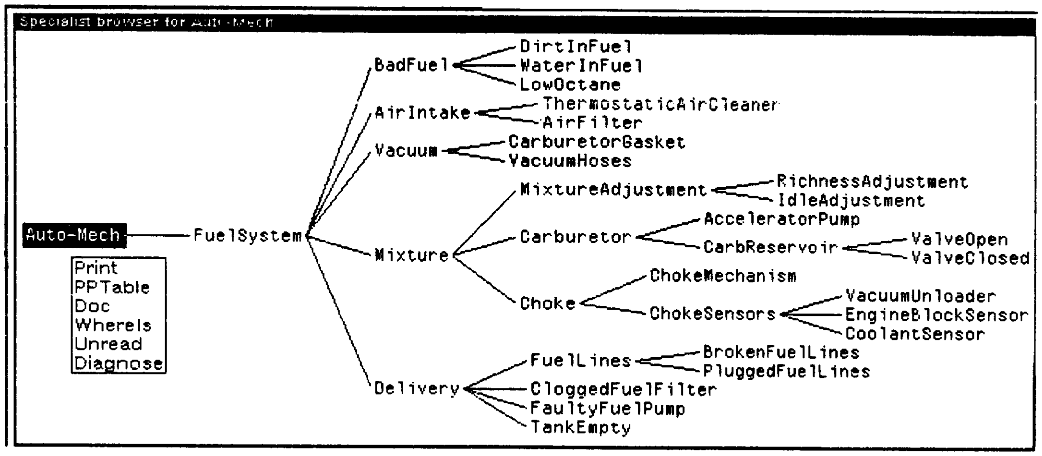
A CSRL system consists of definition of specialists. An example of which is illustrated above.

Each construct in the specialist definition is implemented by a LOOPS class, which is expected to adhere to a specific protocol. The class hierarchy of constructs is illustrated at right.

The tip level classes (on the right) are expected to respond to a Parse message and either a Preprocess or a Run message. Intermediate classes such as ExpWith1Exp contain generalized Parse methods, while others such as Expression contain a Decipher method, which is able to disambiguate between different types of expressions. The Construct class contains general routines for breaking up s-expressions into their constituents according to a pattern specified by a subclass, as well as the "engine" which drives the parsing process.

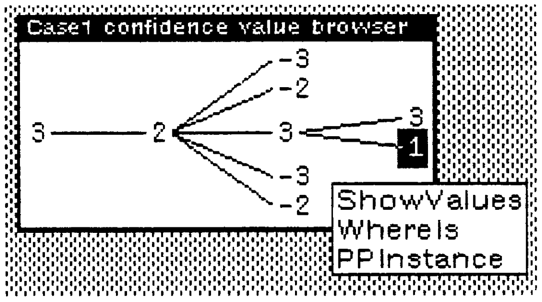


The CSRL programmer can examine his system using a specially built browser to display, edit, and run the system. The browser is a subclass of the ClassBrowser class.

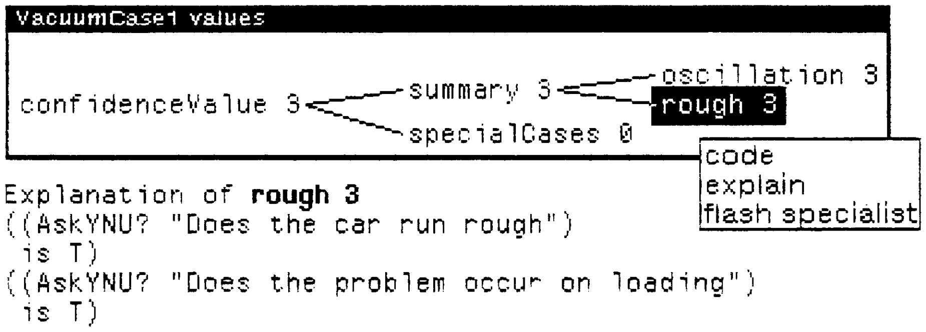


The hierarchy displays the specialist-subspecialist relationships in the system. In this illustration, the Auto-Mech specialist has been left-buttoned, bringing up a menu of commands. The Diagnose command is used to run the hierarchy from this point on a selected case.

The result of running a case is available on another browser which displays the confidence values of each specialist that was run. The range of confidence values is from -3 (the hypothesis was rejected) to +3 (the hypothesis was established).



The menu for the nodes in this browser allows the implementor to flash the corresponding node in the main browser (the WhereIs command), and also to bring up an explanation window for this node's confidence values.



```
Explanation of rough 3
((AskYNU? "Does the car run rough")
 is T)
((AskYNU? "Does the problem occur on loading")
 is T)
```

This is an explanation of Vacuum's confidence value in the current case. It shows how the confidence value was derived from other chunks of knowledge which we call knowledge groups. A knowledge group may combine the values of other knowledge groups, or may derive a value from a list of pattern-action rules. The specialist illustrated earlier has a single knowledge group of the latter type. Buttoning a node ("rough 3" in the above example), and selecting the explain command will print an explanation of the how the value was derived. In this case, the value of the rough knowledge group is 3 because the car runs rough, and the problem occurs on loading.

*[Sanjay Mittal is currently at Knowledge Systems Area, PARC.]

LOOPS at Battelle, Columbus

The KBS (Knowledge Based Systems) group at Battelle has found the LOOPS/INTERLISP environment to be highly supportive of fast prototype development and incremental refinement of a knowledge based system. A Natural Language front end to a DBMS and an expert system for interpreting radiographs of welds are two of the several application projects being developed at Battelle in Columbus, Ohio.

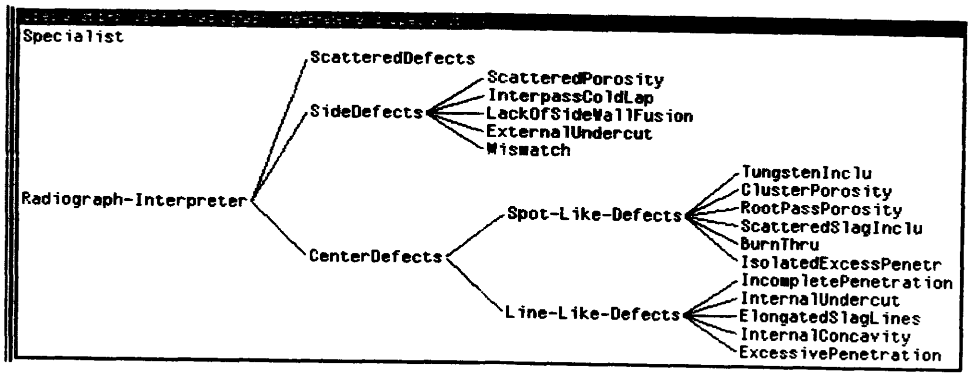
Battelle is in close collaboration with the AI group at Ohio State Universtiy and has found the taxonomy of Problem Solving types to be a useful principle in matching techniques to tasks [B.Chandrasekaran, "Expert Systems: Matching Techniques to tasks", NYU symposium 1982]. Below we describe an expert system built using CSRL, a high level language developed at OSU for classificatory problem solving. For more information on how the LOOPS environment is used at Battelle, contact Dr. Jim Brink at 614/424-4087.

WELDEX: An expert system for interpreting radiographs of welds

by
D.D. Sharma, Sriram Mahalingam
AI Group
Ohio State University

We have developed a prototype system for using expert system technology for Non-Destructive testing of welded joints. Briefly, the image appearing on a radiograph is the projection of the macrostructure of the radiographed piece and of the defects present in the welded section. Features extracted from the image is used as data for WELDEX. Using knowledge of the welding process, the geometry of the welded piece and the radiograph technique used, WELDEX classifies (or interprets) these features as being indicative of specific defects.

On the following page is a feature classification hierarchy. The tip nodes represent the specific kinds of defects WELDEX can deal with. The organization of knowledge and the problem solving strategy follow the MDX paradigm of classificatory problem solving. Each node has knowledge about how well it fits the description of the problem, as well as control knowledge about what nodes to consider next.



This system is now being expanded in terms of knowledge and also being refined in terms of industry specific standards to decide the suitability of welded pieces for their intended application.

Finally, as it currently stands, a human user has to extract features and answer questions asked by the system. We are also designing a knowledge based vision system to extract the features and interface with WELDEX.

Programming tutors with LOOPS

by
Mary Ann Quayle
University of Pittsburgh
Learning Research and Develoment Center

At LRDC we have been using primarily the object-oriented part of LOOPS for implementation of tutors. We presently use LOOPS for our Electricity Tutor, a tutor for learning Ohm's Law, and our Subtraction Tutor, a tutor to address children's subtraction bugs. The programming tutor we refer to as Bridge, is presently in the design stage set for implementation in LOOPS. LOOPS provided a framework for organizing the programming tutor user interface. It is basically a data driven implementation, but is very straight forward and allows alot of specifications to be met without special casing.

As part of our research responsibilities, we provide workshops, a primer, and some LOOPS exercises to sites supported by our project. Our purpose is to give enough information to get started with the system. The intention is that once users have enough expertise to do the basics, they can move to more advanced programming by reading the manual and working with the machine. Participants seem more at ease if they already know some dialect of Lisp, not necessarily Interlisp.

We find ourselves eagerly waiting for incorporation of LOOPS Objects in Interlisp-D, better masterscope facilities for files of LOOPS objects and methods, a better rule system, and better system handling of LOOPS methods and their associated functions.

ANNOUNCEMENTS

For those with access to the ARPA network there are two users' distribution lists for Interlisp programmers: info-1100@sumex-aim for general information, and bug-1100@sumex-aim for discussion of bugs by users.

For Xerox software support, messages can be sent to 1100Support.pasa@Xerox. Our toll free numbers are:

800/228-5325 --	continental US, including Hawaii and Alaska
800/824-6449 --	within California

For those at universities participating in the Xerox University Grants program, Interlisp-D is an additional environment that is available for a moderate software license fee along with the Xerox Development Environment (XDE).

Harmony, the new release, will be arriving soon (if it hasn't already). Harmony has over 600 bug fixes resulting in increased reliability. The most substantial changes are in NS filing, NS printing, TEDIT and the procedure for processing Fonts.

Xerox Artificial Intelligence Sysems Business Unit offers Interlisp, Loops and Expert Systems classes and one and two day seminars and workshops. Call the Training Coordinator on extension 2676 at the above toll free numbers for more information.

QUESTIONS

The Xerox AIS BU development staff is very interested in obtaining written feedback on your requests for new features or wish list items for forthcoming releases, and complaints and comments about the software system. Also, general questions with answers that will be of interest to a general audience will be printed in this section. Comments on the organization and content of the newsletter are welcome.