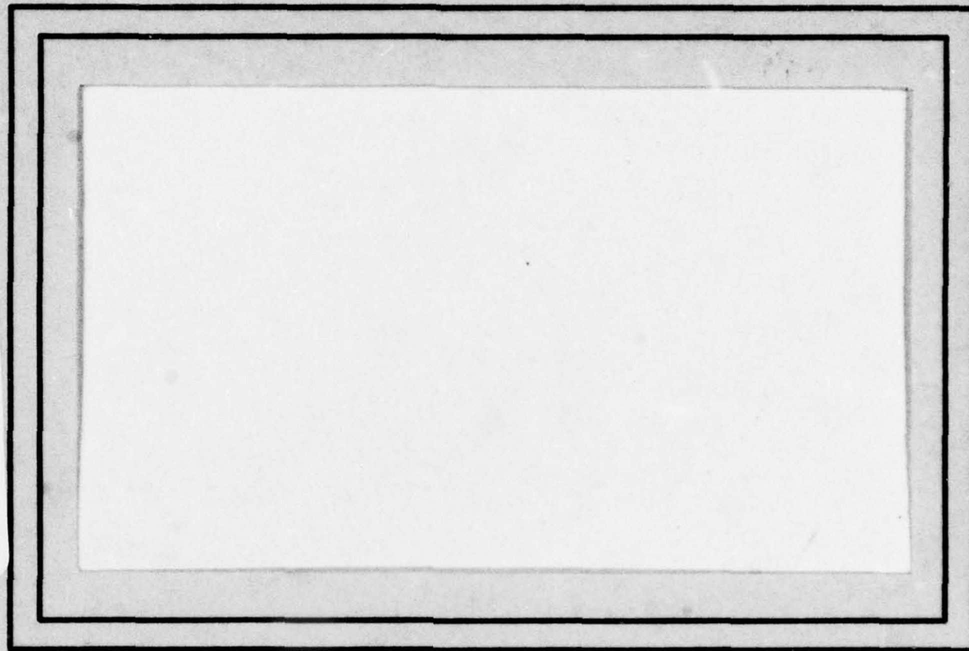
LEVEL II

AD A071603



COMPUTER SCIENCE TECHNICAL REPORT SERIES

DDC FILE COPY

UNIVERSITY OF MARYLAND
COLLEGE PARK, MARYLAND
20742

DDC
RECEIVED
JUL 24 1979
D

DISTRIBUTION STATEMENT A
Approved for public release;
Distribution Unlimited

79 07 23 196

LEVEL II

Technical rept.

TR-646
DAAG-53-76C-0138

March 1978

26p.

A SYSTEM FOR CONTROL STRUCTURE
IMPLEMENTATION FOR IMAGE UNDERSTANDING

Martin Herman

Accession For	
NTIS GRA&I	X
DDC TAB	
Unannounced	
Justification	
By	
Distribution/	
Availability Codes	
Dist	Avail and/or special
A	

ABSTRACT

A data base system with inferencing capabilities is described. This system, developed by C. Rieger, is investigated as a tool in support of control structure implementation for image understanding.

DDC RECEIVED JUL 24 1979 D

The support of the U. S. Army Night Vision Laboratory under Contract DAAG-53-76C-0138, ARPA Order-3206 is gratefully acknowledged.

DISTRIBUTION STATEMENT A
Approved for public release;
Distribution Unlimited

1. Introduction

INFDB is a data base system which supports an inferencing capability. This document describes what is available in the system and how to use it in the context of image understanding and scene analysis. The intent here is to use this software as the basis for a higher level control structure for an image understanding system. The basic data base system (described in Section 2) offers the capability of storing, fetching, and erasing LISP n-tuples. The inferencing capability is provided by the spontaneous computations which are described starting in Section 3. This spontaneous computation system was initially developed and implemented by C. Rieger. The present document describes the user-level capabilities and functions of the system. To obtain a more extensive description of the system, together with details of implementation and theoretical issues, see reference [1].*

The data base and spontaneous computation system was initially implemented to run on the UNIVAC 1108 and UNIVAC 1100/40 machines. R. Haar [2] has modified the system to run in ULISP [3] on a PDP 11/45 under the UNIX operating system.

---

*Certain portions of this text have been copied verbatim from reference [1] with the permission of the author.

2. The basic data base

Each item in the data base consists of a well-formed nested n-tuple, i.e., a LISP list where the list elements can be other lists. (Cyclic references are prohibited.) A nested n-tuple <nn> is defined as:

```
<nn>  := <constant>|<variable>|(<nn>...<nn>)
<constant>  := <LISP atom>
<variable>  := -<LISP atom>
```

There are three user-level data base functions which serve to manipulate items:

```
($STORE <x>), ($FETCH <p>), ($ERASE <p>).
```

$STORE accepts an arbitrary LISP nested n-tuple <x> and inserts it into the data base. $FETCH accepts a pattern, <p>, which possibly contains variables and locates all items in the data base that match <p> by returning pointers to these items. A side effect of locating a matching item is a binding list specifying how each variable in the pattern must be assigned in order for the match to exist. $ERASE accepts patterns similar to those accepted by $FETCH, and erases (deletes from the data base) any items that match the erase pattern.

Examples of these three functions are:

```
($STORE '(TANK TEMPERATURE HOT))
($STORE '(NEAR TREE (TANK TEMPERATURE HOT))
```

```
($STORE '(A (B C) D (E))))
($FETCH '(TANK TEMPERATURE HOT))
($FETCH '(A (-X C) -Y ((E))))
($ERASE '(TANK -X -Y))
($ERASE '(NEAR -X (TANK TEMPERATURE -Y))
```

Note that a variable is distinguished from a constant by having a hyphen prefix (i.e., the first character of the atom is a hyphen). Variables with the same name must be bound to the same object when a match is attempted. Thus

```
($FETCH '(TANK -X -Y))
```

will bind -X to TEMPERATURE and -Y to HOT, while

```
($FETCH '(TANK -X -X))
```

will return NIL since both of the objects to be bound would have to be the same in order for the FETCH to succeed.

3. Spontaneous Computation

The data base just described is used as the groundwork for the demon system. A demon in this system is called a Spontaneous Computation (SC).

An SC has two parts, an activation pattern and a body. The activation pattern is a description of the situation or class of situations to which the SC will react, i.e., spontaneously run itself. The body is the computation it performs. The activation pattern is also called the trigger pattern, since the satisfaction of this pattern by a given situation will trigger the computation in the body of the SC. The trigger pattern consists of nested n-tuples. As an example, consider a trigger pattern which would react to patterns of the form: "(Activate me when) a tank of any temperature is near anything." This could be expressed as

(NEAR -X (TANK TEMPERATURE -Y)).

The following pattern

(NEAR TREE (TANK TEMPERATURE HOT))

would match the trigger pattern, setting -X to TREE and -Y to HOT.

The general SC trigger pattern, <tp>, is defined as follows:

```
<tp>  := <assoc>|<computable>|<complex>
<assoc>  := (+ <effort> <nn>)
```

<computable> := <LISP-S-expression>

<complex> := (AND <tp>...<tp>)

|(OR <tp>...<tp>)

|(ANY <tp>...<tp>)

where <nn> is a nested n-tuple as defined previously.

The associative part, <assoc>, of a complex trigger pattern will react to activity (i.e., patterns) in an arena of events. It is marked with a "+" as in:

(+ 1 (TANK TEMPERATURE -X)).

Any portions of the trigger pattern not so marked will be interpreted as LISP computables. The computable will generally consist of any LISP function call. It will serve to place a restriction on the variables in the associative patterns.

A complex trigger pattern consists of associative and computable parts built from the relations AND, OR, and ANY. An AND condition must have <u>all</u> of its parts bound in a consistent way in order for the AND relation to be true. In an OR relation, at least one of the OR components must be true, and any variables which become bound will reflect the bindings of the first component of the OR found to be true. In an ANY relation, all of the ANY components are tested for truth, and as many bindings as possible are made.

The (+ <effort> <nn>) forms will interface with the system's deductive and data base components (as will be

explained later). The <effort> field, an S-expression that evaluates to an integer, denotes the maximum acceptable level of effort to be expended if the form were to be regarded as a data base query (i.e., deduction). "Effort" is defined as the number of raw data base fetches made.

The following form for a deductive query also exists:

(- <effort> <nn>).

These are identical to "+" forms in all respects except that they cannot initiate the running of the SC directly, i.e., trigger it. They merely express conditions that must be true before the SC's computation can be activated.

Let us apply the complex trigger syntax to express the situation that "the tank whose temperature is the same as that of the truck is near something which is green:"

```
(AND(+ 1 (NEAR -X (TANK TEMPERATURE -Y)))
     (+ 1 (-X COLOR GREEN))
     (+ 1 (TRUCK TEMPERATURE -Y))).
```

All of the parts of this trigger pattern are associative. Whenever any one of its components is seen in the arena of activity, an attempt will be made to verify each other part of the pattern, but only one unit of effort can be expended for each additional part, i.e., the deductive component can use only one data base fetch apiece.

If we wanted the trigger to react only to the "primary" idea, i.e., "the tank is near something," we would make the

last two components non-associative:

```
(AND (+ 1 (NEAR -X (TANK TEMPERATURE -Y)))
     (- 1 (-X COLOR GREEN))
     (- 1 (TRUCK TEMPERATURE -Y)))
```

To illustrate a pattern which includes computables, let us create a pattern which expresses that "something big is near enough to a bush to be less than 3 units from it:"

```
(AND (+ 1 (NEAR -X BUSH))
     (+ 1 (SIZE -X BIG))
     (LESSP  (DIST -X BUSH) 3))
```

where DIST is a function which retrieves the coordinates of the positions of its two arguments and calculates the distance. Here anything which is near a bush or which is big causes the distance between the object and the bush to be compared to the number 3, succeeding if the distance is less than 3.

4. Trigger trees

Generally there will be a large number of SCs in a given application. The SCs can be grouped together into structures called trigger trees. In this way, each population of SCs will inhabit its own trigger tree. An SC can be included as part of a given trigger tree (i.e., "planted" into the trigger tree) by the function $PLANT:

```
($PLANT <tp> <SC-body> <tt>),
```

i.e., "plant in trigger tree <tt> an SC whose trigger pattern is <tp> and whose body is <SC-body>. If the trigger tree <tt> does not exist, it will be created by $PLANT.

The <SC-body> is a LAMBDA expression of one argument. This argument will receive a list of dotted pairs representing the bindings which have caused the SC to be invoked, e.g., ((-X.TREE)(-Y.HOT)). The invocation will cause the LAMBDA expression to be evaluated (executed).

To illustrate, suppose that every time we discover that an object is on a road and its temperature is hot, we want to assert that the object is likely to be moving. We can create the SC to do this by

```
($PLANT '(AND (+ 1(ON -X ROAD))
              (+ 1(-X TEMPERATURE HOT)))
         (LAMBDA (Z)
              ($STORE '(LIKELY MOVING (CAR Z)))
          TT')
```

and the SC will become part of trigger tree TT.

A trigger tree is caused to react to a stimulus pattern in two ways. The first is by calling the function $ACTIVATE:

($ACTIVATE <stimulus> <trigger-tree>).

The second way is by "applying" the trigger tree to the stimulus:

(<trigger tree> <stimulus>),

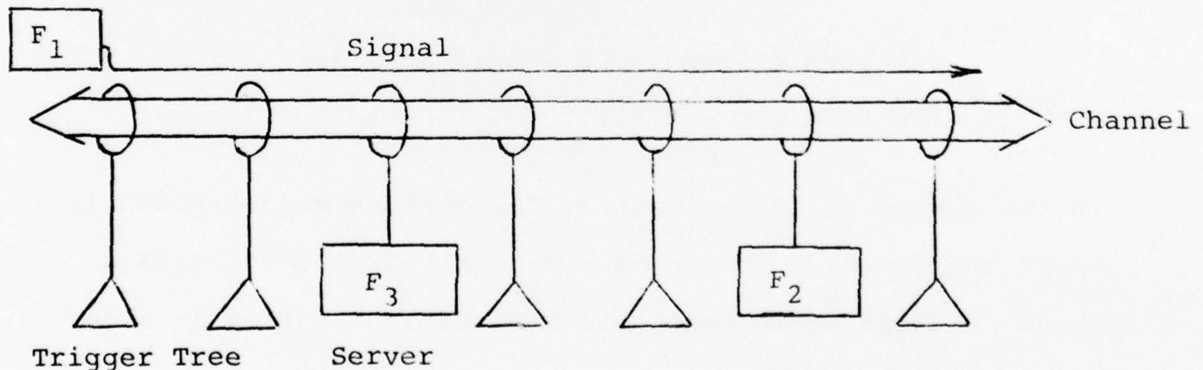
since as a trigger tree is created, it is simultaneously defined as a LISP function which can "apply itself" to a stimulus. For instance, the following two calls will cause the trigger tree TT (as defined above) to react to the stimulus (ON TRUCK ROAD):

```
($ACTIVATE '(ON TRUCK ROAD) 'TT), and
(TT '(ON TRUCK ROAD)).
```

In the system as it now exists, $ACTIVATE does not actually cause the invoked SCs to be run; instead it constructs a queue of calls where each call must be LISP EVALed in order for each SC to be executed.

5. Activity control

The higher level control of SCs is implemented via a construct called a channel. Normally in LISP, when one function calls another, the arguments are sent directly to the called function, and a returned value is sent directly to the calling function. The channel provides an alternative medium whereby one function can call another by posting a request on the channel and sending arguments through the channel. Similarly, a value returned by the called function is sent to the calling function through the channel. This causes the ordinarily private calling protocol between functions to be "made public." Trigger trees may be attached to the channel and react to signals passing along. A channel may be visualized as follows:



Function F_1 must call function F_2 via the channel. One or more trigger trees, as well as other functions such as F_3 in the display, may be attached to the channel, either as "transparent" watchers which can't alter the signals, or as "modifying" watchers which can alter and even terminate the signals.

A channel has the following characteristics:

1. It has a one-dimensional "extent," with directionality.
2. Other constructs (such as functions or trigger trees) can be attached to it at _tap points_; there is no limit to the number of tap points.
3. The left-right ordering of tap points is significant.
4. Each tap point is either a _watcher_ (a trigger tree) or a server (a function), and has mode either _transparent_ (non-altering of signals) or _modifying_ (altering).
5. Signals (either requests to a server or a response from a server) may be injected on a channel at arbitrary starting points and may propagate either left or right.

In this manner, any given trigger tree may be attached to any number of channels at any number of tap points.

There are three primary channel-related functions. They are:

```
($CONNECT <object> <channel> <mode> <type>
        <in-relation-to> <other-point>)
($DISCONNECT <object> <channel>)
($INJECT <signal> <server> <channel> <in-relation-to>
        <other-point> <prop-direction>)
```

The arguments to these functions are defined as follows:

<object> := <watcher>|<server>

<watcher> := <trigger tree name>|<LISP function name>

<server> := <LISP function name>

<channel> := <LISP atom>

<mode> := TRANSPARENT|MODIFYING

<type> := WATCHER|SERVER|RESPONSE-WATCHER

<in-relation-to> := BEFORE|AFTER|AT

<other-point> := <watcher>|<server>
RIGHT-END|LEFT-END

<signal> := <LISP S-expression>

<prop-direction> := LEFT|RIGHT

$CONNECT attaches a trigger tree or LISP function to any point on the channel, i.e., "in-relation-to" some "other-point." The value of <other-point> must be either some existing tap point on the channel or one of the ends of the channel. If <in-relation-to> is BEFORE or AFTER, then the new attachment point is (respectively) to the left or right of the existing tap point.

Imagine the signal on a channel propagating from some starting point in some direction with finite speed. As it passes by a tree of watchers, any relevant watchers in the tree will be triggered and will either (1) allow the signal to continue unaltered, (2) modify the signal and then allow it to proceed, or (3) block the signal altogether. If the signal reaches the requested server, the server will be run

unconditionally on the (possibly modified) signal. Its response will be held while the signal is allowed to propagate to the end of the channel, or until it is blocked. At that time, the server's response will be injected on the channel, starting at the server's tap point. The response consists of the LISP value the server returns. It will propagate from the server's tap point back to the initial starting point. On its way back, the response may pass over a set of "response watchers" which, similarly, may alter or block the signal.

To illustrate, consider the following channel configuration which allows a population of SCs in trigger tree TT to react to items entered into the data base:

```
($CONNECT '$STORE 'CH 'TRANSPARENT 'SERVER
          'AT 'RIGHT-END)
($CONNECT 'TT 'CH 'TRANSPARENT 'WATCHER
          'AFTER '$STORE)
```

i.e., attach the function $STORE (the data base storing function) to channel CH (creating this channel if it does not already exist) as a transparent server at the right end of the channel; then attach trigger tree TT to CH as a transparent watcher after (to the right of) $STORE. Whenever $STORE would have been called directly as in

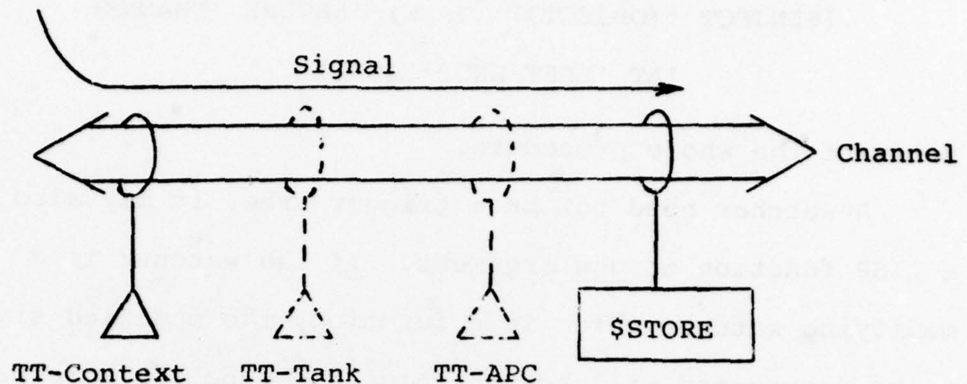
```
($STORE '(ON TRUCK ROAD)),
```

items will now be stored in the data base by placing them as signals to $STORE on CH,

```
($INJECT '(ON TRUCK ROAD) '$STORE 'CH
                 'AT 'LEFT-END 'RIGHT),
```

i.e., inject the signal (ON TRUCK ROAD) to the $STORE server on channel CH, starting at the left end of the channel and propagating right.

Note that we have here the capability to implement a context mechanism, for we can "turn off" or "turn on" whole trigger trees of SCs by means of the $DISCONNECT and $CONNECT functions. In this way, we can create SCs whose only purpose is to turn on or off other trigger trees. For example, suppose we create two trigger trees, TT-APC and TT-TANK, both of which contain SCs which react to signals related to properties of tanks and APCs, such as shape of the base, temperature, overall shape, speed of motion, etc. Suppose it is determined that an object is a tank (or APC) and the system is then directed to look more closely at this object in order to make inferences. It would be very convenient to "turn on" a tank (or APC) context and "turn off" an APC (or tank) context as properties of the object are examined. This could be accomplished with the following configuration:



TT-CONTEXT is a trigger tree containing SCs which turn on or off the TT-TANK and TT-APC trigger trees. $STORE is the data base storing function. This configuration can be implemented as follows:

```
($CONNECT '$STORE 'CHANNEL 'TRANSPARENT
          'SERVER 'AT 'RIGHT-END)
($CONNECT 'TT-CONTEXT 'CHANNEL 'TRANSPARENT
          'WATCHER 'BEFORE '$STORE)
```

This will attach TT-CONTEXT and $STORE to the channel. TT-CONTEXT will contain SCs with the following type of code in their bodies:

```
($CONNECT 'TT-TANK 'CHANNEL 'TRANSPARENT
          'WATCHER 'AFTER 'TT-CONTEXT), and
($DISCONNECT 'TT-APC 'CHANNEL).
```

It is then merely required to inject a signal such as the following:

```
($INJECT '(OBJECT37  TANK) '$STORE 'CHANNEL
          'AT 'LEFT-END 'RIGHT)
```

to start the whole procedure.

A watcher need not be a trigger tree, it may also be a LISP function of one argument. If the watcher is a modifying watcher which is a function, the modified signal to be propagated will be the value the function returns. A trigger tree, however, may contain many computations, several of which might be activated by a signal. The signal to be propagated is determined in the following manner. Each trigger tree has an associated signal buffer. As a signal passes into a trigger tree, its signal buffer is initialized to this original signal. Any SC may alter the signal by replacing the contents of its tree's signal buffer with a new value. The value in the buffer after all SCs have been run is the signal to be propagated. When a watcher (either a trigger tree or a function) modifies the signal to NIL, the signal has been blocked and its propagation ceases.

The method of modifying the contents of a signal buffer of a trigger tree is as follows:

```
(PUT <trigger tree name> 'RESPONSE-BUFFER
      <modified signal>).
```

$PLANT, the function used to create SCs and plant them in some trigger tree, will accept three optional arguments in addition to the three mandatory ones (i.e., trigger

pattern, SC body, and trigger tree name). The optional arguments are the following:

1. A <u>reference name</u>, specified by the form '(N . <LISP-atom>). This will cause the SC to be named, as in:

```
($PLANT '(LYING-ON ROAD TREE)
        <some body>
        'TT3
        '(N . FALLENTREE))
```

2. A <u>run-queue priority</u>, either FRONT or REAR, specified by the form '(P . {FRONT, REAR}). This will indicate to $ACTIVATE how to queue up the SC for subsequent running; either place it at the FRONT of the queue (the default) or at the REAR.

3. A <u>run condition</u>, specified by the form '(R · <condition-builder>), where <condition-builder> is an S-expression which will be EVALed immediately prior to queueing an activated SC for running. The SC will not be permitted to run until the condition is true; it will simply be placed at the end of the queue. It is slightly different for an SC in a trigger tree attached to a channel. In this case, the SC will be run after a successful invocation only if its run condition EVALs non-NIL <u>at the time the tree is activated</u> by a passing signal; SCs whose run conditions

are not satisfied at the time are discarded. If the run condition is omitted in the $PLANT function, the default value is TRUE.

6. Representing numerical quantities

There is a special problem which arises with data base entries containing numbers. When the data base system stores an item (i.e., a nested n-tuple), it creates a back-pointer to the item on the property list of each atom appearing in the item. However, numeric atoms in LISP do not have property lists. This problem has been solved on the systems existing on the UNIVAC 1100 machines by prefixing all numbers with a #, thus making them non-numeric. A parallel set of arithmetic functions have been created for these forms.

The data base system on the PDP 11/45 has a different solution. If an item containing numbers is to be stored, there simply is no back-pointer to the item created for the numeric atoms. Therefore an item cannot be retrieved by specifying only numbers in the item. For example, suppose we call

```
($STORE '(TANK TEMPERATURE 100)).
```

If we then call

```
($FETCH '(-X -Y 100)),
```

there will be no response, since there is no pointer from the numeric atom, 100, to the item. If however, we call

```
($FETCH '(-X TEMPERATURE 100)), or
($FETCH '(TANK TEMPERATURE -X)),
```

the appropriate item will be found in the data base.

Floating point numbers are handled by LISP in exactly the same manner. However, there is a slight inconvenience when typing in a floating point number under the ULISP interpreter on the PDP 11/45. The user must first load the "floating point package" in order to obtain access to a floating point READ macro. Every floating point number which is then typed in must be prefixed with a %.

7. Accessing images

The ULISP system on the PDP 11/45 has facilities for handling arrays. Since arrays offer the most efficient way to store images, the basic image accessing capability is already built into ULISP. Upon loading the arithmetic package, a full image handling capability is provided. It may be more appropriate to use only one-dimensional byte arrays with indices computed by a function rather than two-dimensional arrays. The 2-D arrays create extraneous nodes for pointers and thus use more memory space than 1-D arrays.

While it is desirable from an image processing point of view that ULISP interact with other languages on the PDP 11/45 under UNIX, there is little hope for interaction with FORTRAN. However, communication with the programming language C is feasible.

8. Conclusion

The data base plus inferencing system described here provides a foundation on which to build an image understanding system. Although the system has not been designed for efficient utilization of either memory storage or CPU time, it will prove very useful for constructing and experimenting with various control structures for image understanding.

References

1. C. Rieger, Spontaneous Computation in Cognitive Models, TR-459, Dept. of Computer Science, University of Maryland, College Park, Maryland, July 1976.

2. R. L. Haar, A Fuzzy Relational Data Base System, TR-586, Computer Science Center, University of Maryland, College Park, Maryland, September 1977.

3. R. L. Kirby, ULISP for PDP-11s with Memory Management, TR-546, Computer Science Center, University of Maryland, College Park, Maryland, June 1977.

UNCLASSIFIED

SECURITY CLASSIFICATION OF THIS PAGE (*When Data Entered*)

REPORT DOCUMENTATION PAGE		READ INSTRUCTIONS BEFORE COMPLETING FORM
1. REPORT NUMBER	2. GOVT ACCESSION NO.	3. RECIPIENT'S CATALOG NUMBER
4. TITLE (*and Subtitle*) A SYSTEM FOR CONTROL STRUCTURE IMPLEMENTATION FOR IMAGE UNDERSTANDING		5. TYPE OF REPORT & PERIOD COVERED Technical
		6. PERFORMING ORG. REPORT NUMBER TR-646
7. AUTHOR(s) Martin Herman		8. CONTRACT OR GRANT NUMBER(s) DAAG53-76C 0138
9. PERFORMING ORGANIZATION NAME AND ADDRESS Computer Science Ctr. Univ. of Maryland College Pk., MD 20742		10. PROGRAM ELEMENT, PROJECT, TASK AREA & WORK UNIT NUMBERS
11. CONTROLLING OFFICE NAME AND ADDRESS U. S. Army Night Vision Lab. Ft. Belvoir, VA 22060		12. REPORT DATE March 1978
		13. NUMBER OF PAGES
14. MONITORING AGENCY NAME & ADDRESS(*if different from Controlling Office*)		15. SECURITY CLASS. (*of this report*) Unclassified
		15a. DECLASSIFICATION/DOWNGRADING SCHEDULE

16. DISTRIBUTION STATEMENT (*of this Report*)

Approved for public release; distribution unlimited.

17. DISTRIBUTION STATEMENT (*of the abstract entered in Block 20, if different from Report*)

18. SUPPLEMENTARY NOTES

19. KEY WORDS (*Continue on reverse side if necessary and identify by block number*)

Image understanding
Control structures
Data bases
Inferencing

20. ABSTRACT (*Continue on reverse side if necessary and identify by block number*)

A data base system with inferencing capabilities is described. This system, developed by C. Rieger, is investigated as a tool in support of control structure implementation for image understanding.

DD FORM 1 JAN 73 1473 EDITION OF 1 NOV 65 IS OBSOLETE

UNCLASSIFIED

SECURITY CLASSIFICATION OF THIS PAGE (*When Data Entered*)