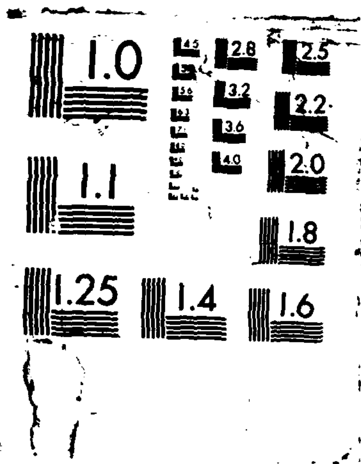
AD-A189 594 IMPOSING CONTROL ON OPS83(U) NAVAL OCEAN SYSTEMS CENTER 1/1
SAN DIEGO CA L E GADBOIS AUG 87 NOSC-TD-1156

UNCLASSIFIED F/G 12/9 NL

END
DATE
FILMED
8



NOSC TD 1156
AD-A189 594
DTIC FILE COPY
(4)
NOSC TD 1156

NOSC
NAVAL OCEAN SYSTEMS CENTER San Diego, California 92152-5000

Technical Document 1156
August 1987

Imposing Control on OPS83

L. E. Gadbois

DTIC
ELECTE
JAN 05 1988
S D
H

Approved for public release; distribution is unlimited.

87

NAVAL OCEAN SYSTEMS CENTER
San Diego, California 92152-5000

E. G. SCHWEIZER, CAPT, USN
Commander

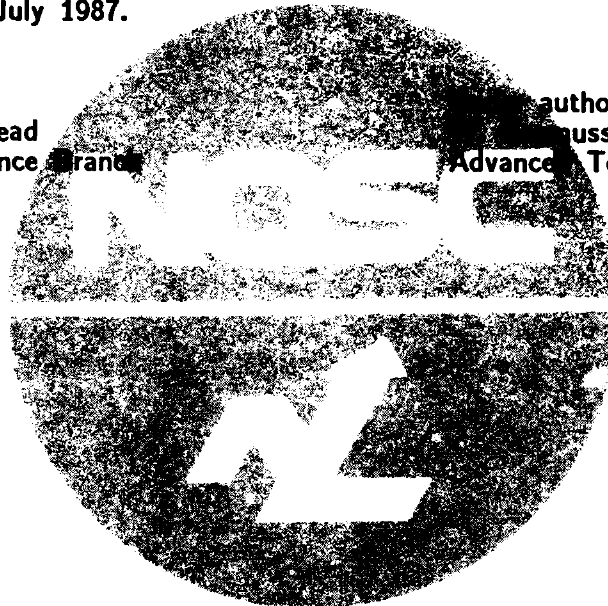
R. M. HILLYER
Technical Director

ADMINISTRATIVE INFORMATION

This report was sponsored by the Naval Ocean Systems Center Independent Research and Exploratory Development Program. The work was conducted over the period June-July 1987.

Approved by
D. Eddington, Head
Artificial Intelligence [illegible]ranch

[illegible] authority of
[illegible]ussen, Head
Advance[illegible] Technologies Division



AS

UNCLASSIFIED
SECURITY CLASSIFICATION OF THIS PAGE

REPORT DOCUMENTATION PAGE

1a REPORT SECURITY CLASSIFICATION: UNCLASSIFIED	1b RESTRICTIVE MARKINGS
2a SECURITY CLASSIFICATION AUTHORITY	3 DISTRIBUTION AVAILABILITY OF REPORT: Approved for public release; distribution is unlimited.
2b DECLASSIFICATION DOWNGRADING SCHEDULE	
4 PERFORMING ORGANIZATION REPORT NUMBER(S): NOSC TD 1156	5 MONITORING ORGANIZATION REPORT NUMBER(S)
6a NAME OF PERFORMING ORGANIZATION: Naval Ocean Systems Center; 6b OFFICE SYMBOL (if applicable): Code 444	7a NAME OF MONITORING ORGANIZATION
6c ADDRESS (City, State and ZIP Code): San Diego, CA 92152-5000	7b ADDRESS (City, State and ZIP Code)
8a NAME OF FUNDING SPONSORING ORGANIZATION: NOSC IR/IED Program; 8b OFFICE SYMBOL (if applicable)	9 PROCUREMENT INSTRUMENT IDENTIFICATION NUMBER
8c ADDRESS (City, State and ZIP Code)	10 SOURCE OF FUNDING NUMBERS

PROGRAM ELEMENT NO	PROJECT NO	TASK NO	AGENCY ACCESSION NO
61152N	RR0000101	444-ZT70	DN307 404

11 TITLE (include Security Classification): Imposing Control on OPS83

12 PERSONAL AUTHOR(S): L.E. Gadbois

13a TYPE OF REPORT	13b TIME COVERED	14 DATE OF REPORT (Year, Month, Day)	15 PAGE COUNT
Final	FROM Jun 1987 TO Jul 1987	August 1987	23

16 SUPPLEMENTARY NOTATION

17 COSATI CODES			18 SUBJECT TERMS (Continue on reverse if necessary and identify by block number)
FIELD	GROUP	SUB GROUP	Artificial intelligence Algorithmic control OPS83

19 ABSTRACT (Continue on reverse if necessary and identify by block number)

The inference engine in the artificial intelligence language OPS83 applies knowledge resident in rules to data in working memory to identify and solve problems. Solving a task may require unconstrained application of rule knowledge to the problem. Often, however, the solution may require applying knowledge to data elements in a algorithmic manner, using a control method.

Two methods of imposing control are contrasted: (1) Inclusion of negated conditions in the antecedent, and (2) Making contexts to instantiate relevant rule sets in sequential order. Several means of imposing algorithmic control to cause the inference engine to step through the sequence can be used. Managing these control schemes imposes a burden on the inference engine. The effect on total run time of that burden is more than compensated for by efficiency in the conflict-resolution scheme resulting from addressing sequential pieces of the task rather than the whole task at once.

Storage of information in global variables is orders of magnitude faster than storage in working memory. Storage in working memory is dictated, however, when: (1) The number of instantiations of a class of information is variable; or (2) The information is needed in the antecedent of a rule. These cases excepted, global variables should be used.

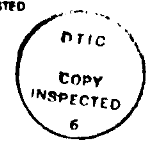
When many working memory elements are needed, accessing them from the right-hand side (the consequent) of rules by means of primitive OPS83 routines can be much faster than firing a sequence of rules to perform the same task. Other features of these primitive OPS83 routines are: (1) Tasks which do not require further knowledge from other rules can be performed quickly by not escaping to the shell each time additional information from working memory is needed; and (2) Sequential tasks vulnerable to the action of other rules can be completed within a single rule.

20 DISTRIBUTION AVAILABILITY OF ABSTRACT: ☐ UNCLASSIFIED UNLIMITED ☒ SAME AS RPT ☐ DTIC USERS	21 ABSTRACT SECURITY CLASSIFICATION: UNCLASSIFIED
22a NAME OF RESPONSIBLE INDIVIDUAL: Laurence E. Gadbois	22b TELEPHONE (include Area Code): (619) 225-7367; 22c OFFICE SYMBOL: Code 444

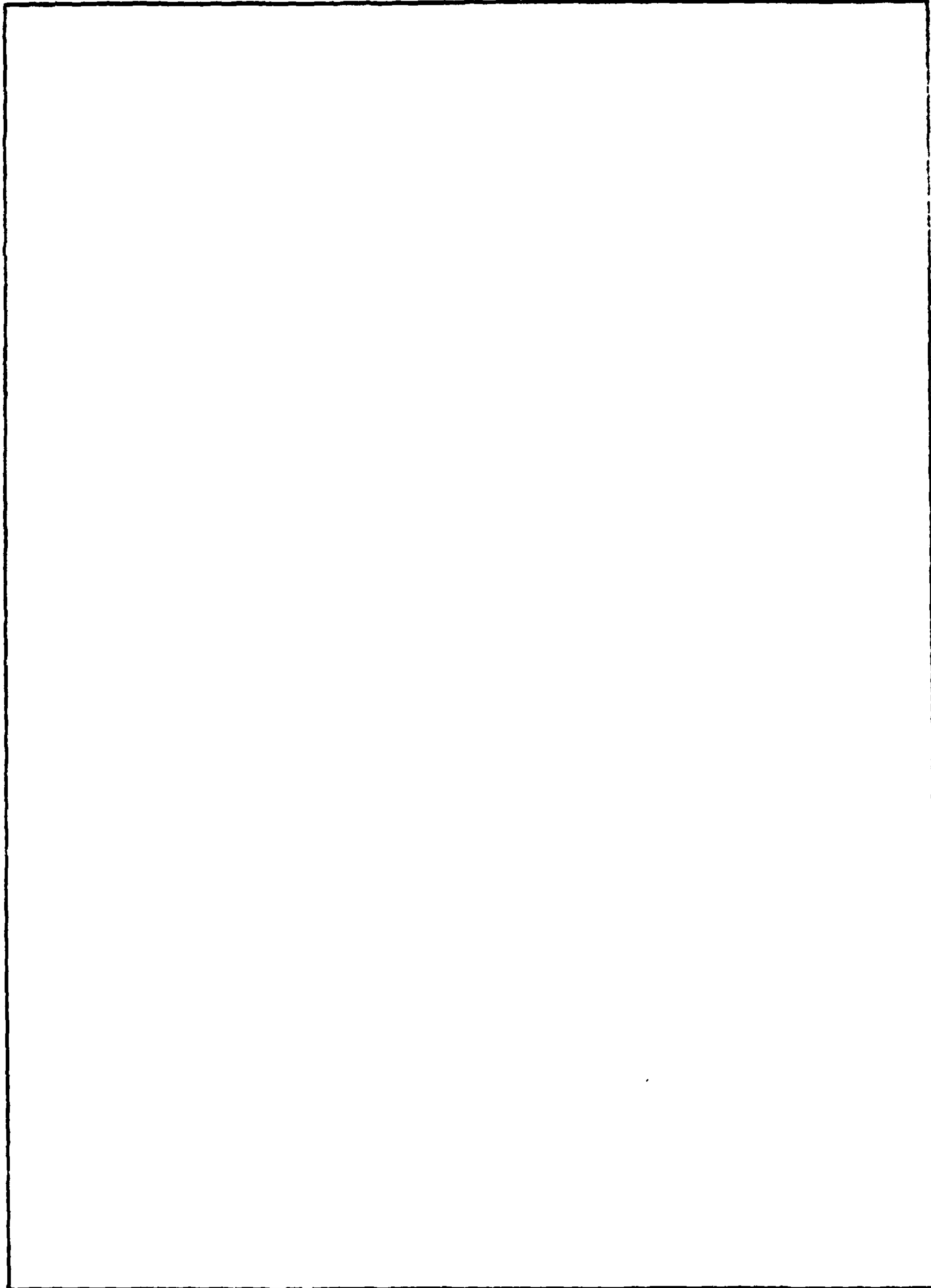
DD FORM 1473, 84 JAN

83 APR EDITION MAY BE USED UNTIL EXHAUSTED
ALL OTHER EDITIONS ARE OBSOLETE

UNCLASSIFIED
SECURITY CLASSIFICATION OF THIS PAGE



Codes
and/or
Dist Special
A-1



EXECUTIVE SUMMARY

OBJECTIVE

The objective of the work described in this report was to assess the impact on run time of imposing control on the execution flow in OPS83 programs.

METHOD

Artificial intelligence (AI) programs generally require that the inference engine search through a problem space for a solution. The OPS83 AI language uses an inference engine to apply knowledge residing in rules to data elements, in order to provide a solution. The subtasks performed during the search may require unconstrained application of rule knowledge to the data set. When the need to perform a specific subtask is identified, it is often possible to apply knowledge to data elements in a sequential, algorithmic manner. Several means of imposing algorithmic control to make the inference engine step through the sequence can be used. Managing these control schemes imposes a burden on the inference engine. The effect of that burden is more than compensated for by the efficiency in the conflict-resolution scheme which results from presenting the inference engine with sequential pieces of the task rather than overwhelmling it with the whole task at once. Dramatic overall savings in cpu time result.

TERMINOLOGY USED IN THE RESULTS

Rules contain a left-hand-side (LHS), or antecedent, which specifies patterns of information which must be in data (working) memory. These patterns can contain positive conditions, which must match a working memory element (WME). The patterns can also contain negated conditions, which specify that there are no WMEs which match this condition.

RESULTS

- Negated conditions and contexts on rule LHSs can produce severalfold increases in execution speed by providing guidance for algorithmic tasks.

- Storage of information in global variables can be an order of magnitude or more faster than storage in working memory. Storage in working memory is dictated, however, when the number of instantiations of a class of information is variable or the information is needed on the LHS of rules. These cases excepted, global variables should be used.

- Calling up WMEs from the right-hand-side (RHS) of rules by means of primitive OPS83 routines can be much faster than firing a sequence of rules to perform the same task. Other features of these primitive OPS83 routines are: (1) Tasks which do not require further knowledge from other rules can be performed quickly by not having to escape to the shell each time additional information from working memory is needed, and (2) Sequential tasks vulnerable to the action of other rules can be completed within a single rule without exiting to the shell, thereby avoiding the action of other rules and any slowness from executing the other shell procedures.

CONTENTS

INTRODUCTION 1
Problem 1
Background 1

METHODS 3
Approach 3
Software 4
Hardware 4

RESULTS 4
Measurements 4
Information storage in working memory versus global variables 5
Effects of no control 7
Control via negation 7
Control via contexts 7
Searching working memory within the right-hand side 8
Summary of results 8

CONCLUSIONS/RECOMMENDATIONS 9

APPENDIX: SOURCE CODE FOR THE CONTROL REGIMES 11

INTRODUCTION

PROBLEM

The data and knowledge representation forms used in production systems are designed for nonalgorithmic search through a problem space. This is conducted via an inference engine which applies the knowledge resident in rules to data resident in working memory elements (WMEs). An intolerably slow process results when many data elements and rules are examined concurrently.

BACKGROUND

Artificial intelligence (AI) is ideally suited for solving nonalgorithmic problems through the application of domain or other "advanced" knowledge. AI is used to assess the overall scope of the problem and determine the approaches most likely to lead to a correct conclusion. This is done by applying knowledge (represented in structures called rules) to facts and data (stored in structures called WMEs). After the initial nonalgorithmic steps, the subsequent steps are often algorithmic.

When an algorithmic phase of the task is reached, a number of approaches can be pursued to solve the problem. These tasks may include: (1) database calls; (2) direct linear programming techniques applied within the RHS (right-hand side, or consequent) of a single rule without the need to extract more information than was bound on the LHS (left-hand side, or antecedent); (3) linear techniques which need more information (usually from working memory) than was bound on the LHS; or (4) searches which apply the knowledge resident in multiple rules to data in working memory.

This report addresses various approaches to the types of tasks outlined in 3 and 4 above. For clarity in subsequent discussions, the term "multidata tasks" will be used to refer to the types of tasks in number 3 above. The term "multidata/knowledge tasks" will be used to refer to the types of tasks in number 4 above.

A multidata task involves searching through information in working memory which was not bound to variables on the LHS of the rule. OPS83 provides low-level routines for just such applications, which, as will be seen later, are a fast alternative to firing rules which search working memory. These routines provide a very viable means for searching working memory. They can also be used by the shell or other non-rule portions of the program. They are valuable for use on the RHS for such reasons as: (1) There are more WMEs needed than the maximum number which can be bound on the LHS; and (2) The number of WMEs needed may be unknown or variable (meaning a generic rule with a set LHS would not encompass all cases). In summary, these primitive routines allow complete access ("read", modify, or remove) to all the information in working memory from any point in OPS83 code.

A multidata/knowledge task involves the following sequence: (1) Identify that a task needs to be performed; (2) Encode the knowledge needed to perform that task in a set of rules; (3) Apply the production kncwledge to the data (generally working memory); and (4) Derive a best solution to the task. Because a solution may be derivable from multiple sets of input (both in content and form), the rule sets must be diverse enough to encompass the different input forms. Diversity in the rule set is also needed to accommodate the distillation of multiple solutions when more than one is valid and may need to be considered.

To summarize, the main thrust of the multidata/knowledge tasks is to recognize that performance of a task is needed and to bring to bear the knowledge in a set of rules and the data in a set of working memory towards that task. An approach to this task presented herein involves placing a context to perform a certain goal in working memory and having this context be a toggle on the LHS of rules applicable to this task.

At this point a sphere of control comes into play. The set of rules can be "turned loose" on the problem to fire in an unguided order, drawing numerous conclusions which are not necessary for the current task. This task must be concluded by sorting through the multitude of conclusions, keeping those which are appropriate and discarding the chaff. This "turning loose" approach has such advantages as: (1) No time is spent making and managing the control regimes; (2)

It finds many solutions when needed; and (3) It is applicable when the path to take to the solution is not well known in advance. The disadvantages include: (1) So many rule instantiations to the conflict-resolution algorithm are presented that a bottleneck is created; and (2) The free-form generation of so many conclusions, many of which are unnecessary, results in the performance of excess work.

Two alternatives to "turning loose" the rule set are compared. The first is to make a context which contains a stack of the steps necessary to perform the task. These are incrementally stepped through by the subset of rules applicable to this step. As each step is performed, it is popped off the stack. This approach allows fairly rigid control, appropriate for strongly algorithmic tasks.

The second alternative evaluated in this report is the use of negation conditions on the LHS of rules. A negated condition is satisfied when nothing in working memory matches the negated condition. This can be used for control in LHSs by specifying that the current step in the task is not completed (the negated condition) while the previous step is completed (a positive condition).

METHODS

APPROACH

Short programs were written which differed in the following features: (1) Storage of data was via WMEs or global variables; (2) Control on execution flow was either unrestrained, controlled with contexts, or controlled with negations; and (3) Searching working memory within the RHS of a single rule was measured and compared with firing numerous rules, each of which identified an element in working memory.

SOFTWARE

The UNIX version 3.0 "prof" program was used to measure the total run times (see a UNIX manual for details on the prof command).

Version 2.1(B) of the OPS83 compiler was used for all programs.

HARDWARE

Measurements were made on a SUN 3/160 workstation with 8 Mbytes of main memory (Motorola 68020 cpu). A 385-Mbyte formatted hard disk was used, with 150 Mbytes of swap space.

RESULTS

MEASUREMENTS

Run time will be affected differently by inclusion of LHS control conditions, depending on many production and working memory parameters. Thus actual magnitudes and ratios presented in table 1 are an artifact of the production and data memory configurations used for these experiments. The general trends alluded to by the values in table 1 are sufficient to draw comparisons among: (1) information storage in working memory verses global variables, (2) effects of no control, (3) effects of control via negation, (4) effects of control via contexts, and (5) efficiency of searching working memory within the RHS.

INFORMATION STORAGE IN WORKING MEMORY VERSUS GLOBAL VARIABLES

Information learned at run time can be stored internal to the program as WMEs or in global variables (external storage, such as files, is an option not addressed in this report). The decision of which approach to use is dictated by two factors: how the information will be used, and whether the number of instantiations of that class of information is fixed. If the information is needed on the LHS of rules to regulate which rules will fire, it needs to be in working memory. If this information is one of a variable number of instantiations of a class of information, then it is most convenient to represent it as a WME. Table 1 presents a comparison of global variables versus WMEs.

Table 1. Use of global variables versus WMEs.

Global variables	WMEs
• Value can be accessed and changed easily from the RHS of rules.	• Normally must be bound on the LHS of the rule to access and remove it from the RHS.
• Cannot be used on the LHS of rules, thus it cannot be used in the decision of which rule to fire.	• Can be used on the LHS, allowing it to be used in the selection of which rules to fire.
• Value can be modified often and quickly.	• Modification can be very slow depending on its involvement in and size of working memory. Slowness results because each change must be worked through the Rete network.

In the results shown in table 2, the use of global variables to store the result was up to nine times faster than a WME. Thus a savings of time can be realized by using global variables when appropriate to the other constraints on the information representation.

Table 2. Measurements for all the control regimes run in duplicate.

Table 2A. First set of results.

Control structure	Number of WMEs. 10	50	100	250
1	0.78	4.28	15.20	96.54
2	0.74	1.56	4.58	36.34
3	0.68	2.66	8.02	42.90
4	0.66	1.18	1.94	4.64
5	0.50	0.58	0.70	0.94
6	0.66	1.32	1.92	4.42

Table 2B. Second set of results.

Control structure	Number of WMEs. 10	50	100	250
1	0.70	4.28	15.08	100.44
2	0.72	1.62	4.52	36.96
3	0.76	2.70	7.72	43.06
4	0.68	1.08	1.72	4.52
5	0.52	0.62	0.62	0.90
6	0.66	1.30	2.16	5.06

Values in the body of the table are milliseconds of cpu time to run the program. Key to control structures:

1) No control used. The result was put in *working memory* (see section A.1).

2) LHS negations used. The result was placed in working memory (see section A.2).

3) No control used. The result was put in *a global variable* (see section A.3).

4) LHS negations used. The result was put in a global variable (see section A.4).

5) No control. RHS conducted a search through working memory. The result was put in a global variable (see section A.5).

6) Context used for control. The result was put in a global variable (see section A.6).

EFFECTS OF NO CONTROL

Programs which have no control elements or negations expose all the rule LHS conditions to the full working memory. This can result in large conflict sets. The brute force firing of all the rules against all the WMEs can lead to an exponential proliferation of conclusions, only several of which are appropriate to the current task. Table 2 shows a speed up for all methods of imposing control.

CONTROL VIA NEGATION

The impact of negations on run time is multifaceted. Negated LHS condition elements are expensive for the match algorithm. Each time a WME is added or removed, the negations must be examined to see if they are now satisfied. Negations reduce the conflict-resolution time by having in the queue only the rules appropriate for that stage of the problem solving. In the results of table 2, negations markedly reduced the total run time by keeping the conflict set size to a single rule.

CONTROL VIA CONTEXTS

The use of contexts has mixed effects on the match times and positive effects on the conflict-resolution times. The act of making and removing the contexts must be traced through the Rete network. On the other hand, having context conditions early in the LHS of rules relieves the match algorithm of much work since it does not have to do total matching against the remainder of the LHS of the rule. Conflict-resolution time is reduced because only the rules appropriate to the current stage of the task are entered in the conflict set.

SEARCHING WORKING MEMORY WITHIN THE RIGHT-HAND SIDE

For accessing information in many WMEs, searches from the RHS are much faster than firing many rules (see table 2, control structure number 5). This method of search cannot be interrupted by other rules firing, a valuable feature when this level of complete control and uninterrupted sequences are desired.

SUMMARY OF RESULTS

- Negated conditions and contexts on rule LHSs can produce manyfold increases in execution speed by providing guidance for algorithmic tasks.

- Storage of information in global variables is orders of magnitude faster than storage in working memory. Storage in working memory is dictated, however, when the number of instantiations of a class of information is variable or the information is needed on the LHS of rules. These cases excepted, global variables should be used.

- Calling up WMEs from RHS of rules by means of primitive OPS83 routines can be much faster than firing a sequence of rules to perform the same task. Other features of these primitive OPS83 routines are: (1) Tasks that do not require further knowledge from other rules can be performed quickly by not having to escape to the shell each time additional information from working memory is needed; and (2) Sequential tasks vulnerable to the action of other rules can be completed within a single rule without exiting to the shell, thereby avoiding the action of other rules.

CONCLUSIONS/RECOMMENDATIONS

Three conclusions result from this research:

- Global variables can be modified much faster than working memory. Thus global variables should be used whenever appropriate given all other constraints on their use.

- Control conditions (contexts and negations) should be used freely to toggle the pattern matching of rules for identified tasks.

- Searching working memory from a rule's RHS can be a very quick means to perform certain tasks.

APPENDIX: SOURCE CODE FOR THE CONTROL REGIMES.

A.1. NO CONTROL USED WITH THE RESULT IN WORKING MEMORY.

This program generated the results presented in the first row of table 2. The task is to take 10, 50, 100, or 250 integers in separate WMEs (called a "piece" in this example) and add their value to the element "result" which is kept in working memory. Each piece is removed from working memory after its value has been added to the result. When shell is called at the end of procedure main1, "rule number1" fires once for each piece.

```
module setup(main1) {
use o83shl;                                   -- Link to libraries, and the shell
                                              -- which controls conflict resolution
                                              -- and rule firings.
type result = element (value: integer);       -- Declare a WME type.
type piece = element (value: integer);        -- Declare another WME type.
rule number1 {                                -- This is the only rule in the
      &1  (result);                           -- program. It's LHS points to the
      &2  (piece);                            -- result, and to any "piece" in
                                              -- working memory. Thus no control.
-->                                           -- The LHS (beginning here) takes the
modify &1 (value = &1.value + &2.value);--    value in piece and adds it to
remove &2;                                    -- the result, modifying result. Piece
}; -- end rule                                -- is then removed.

procedure main1() {
      local &i: integer;          -- Variables begin with "&"
      make (result);              -- The result will be in working memory.
      for &i = (1 to 10)          -- This number changed from 10, 50, 100, and 250
                                  -- to produce the results shown in table 2.
      make (piece value = &i);    -- Make 10 pieces whose values end up being 1
                                  -- to 10.
      call shell();               -- Invoke the shell for conflict resolution and
                                  -- rule firing.
}; -- end procedure main1
}; -- end module
```

A.2. LHS NEGATIONS USED WITH THE RESULT IN WORKING MEMORY

This program generated the results presented in the second row of table 2. The task is to take 10, 50, 100, or 250 integers in separate WMEs (called a "piece" in this example) and add their value to the element "result" which is kept in working memory. Each piece is removed from working memory after its value has been added to the result. When shell is called at the end of procedure main1, "rule number1" fires once for each piece. The choice of which piece to choose is dictated by a negation on the LHS.

```
module setup(main1) {
use o83shl;                               -- Link to libraries, and the shell
                                          -- which controls conflict resolution
                                          -- and rule firings.
type result = element (value: integer);   -- Declare a WME type.
type piece = element (value: integer);    -- Declare another WME type.
rule number1 {                            -- This is the only rule in the program.
      &1  (result);                       -- points to the result, and
      &2  (piece);                        -- to the "piece" which
      ~   (piece value < &2.value);       -- is the smallest.  Thus control is by
                                          -- negation.
-->                                       -- The RHS (beginning here) takes the
modify &1 (value = &1.value + &2.value);-- value in piece and adds it to
remove &2;                                -- result, modifying result.  Piece is
}; end rule                               -- then removed.

procedure main1() {
      local &i: integer;         -- Variables begin with "&"
      make (result);             -- The result will be in working memory.
      for &i = (1 to 10)         -- This number changed from 10, 50, 100, and 250
                                 -- to produce the results shown in table 2.
      make (piece value = &i);   -- Make 10 pieces whose values end up being 1
                                 -- to 10
      call shell();              -- Invoke the shell for conflict resolution and
                                 -- rule firing.
}; -- end procedure main1
}; -- end module
```

A.3. NO CONTROL WITH THE RESULT IN A GLOBAL VARIABLE

This program generated the results presented in the third row of table 2. The task is to take 10, 50, 100, or 250 integers in separate WMEs (called a "piece" in this example) and add their value to the global variable "result." Each piece is removed from working memory after its value has been added to the result. When shell is called at the end of procedure main1, "rule number1" fires once for each piece.

```
module setup(main1) {
use o83shl;                             -- Link to libraries, and the shell
                                        -- which controls conflict resolution
                                        -- and rule firings.
global &result: integer;                -- Result is a global variable.
type piece = element (value: integer);  -- Declare a WME type.
rule number1 {                          -- This is the only rule in the
     &1  (piece);                       -- program.  It's LHS points to all
                                        -- "pieces".  There is no control used.
-->                                     -- The RHS (beginning here) takes the
&result = &result + &1.value;           -- value in piece and adds it to
                                        -- result.
remove &1;                              -- Piece is then removed.
}; -- end rule

procedure main1() {
     local &i: integer;          -- Variables begin with "&"
     &result = 0;                -- Initialize the global variable.
     for &i = (1 to 10)          -- This number changed from 10, 50, 100, and 250
                                 -- to produce the results shown in table 2.
     make (piece value = &i);    -- Make 10 pieces whose values end up being 1
                                 -- to 10
     call shell();               -- Invoke the shell for conflict resolution and
                                 -- rule firing.
}; -- end procedure main1
}; -- end module
```

A.4. LHS NEGATIONS USED WITH THE RESULT IN A GLOBAL VARIABLE

This program generated the results presented in the fourth row of table 2. The task is to take 10, 50, 100, or 250 integers in separate WMEs (called a "piece" in this example) and add their value to the global variable "result." Each piece is removed from working memory after its value has been added to the result. When shell is called at the end of procedure main1, "rule number1" fires once for each piece.

```
module setup(main1) {
use o83shl;                                -- Link to libraries, and the shell
                                           -- which controls conflict resolution
                                           -- and rule firings.
global &result: integer;                   -- Result is a global variable.
type piece = element (value: integer);     -- Declare a WME type.

rule number1 {                             -- This is the only rule in the
    &2  (piece);                           -- program. The first condition points
                                           -- to all the "pieces", but the
    ~   (piece value < &2.value);          -- negation constrains the LHS to the
                                           -- smallest piece.
-->                                        -- The RHS (beginning here) takes the
&result = &result + &2.value;              -- value in piece and adds it to result
remove &2;                                 -- and then removes the piece.
}; -- end rule

procedure main1() {
    local &i: integer;          -- Variables begin with "&"
    make (result);              -- The result will be in working memory.
    for &i = (1 to 10)          -- This number changed from 10, 50, 100, and 250
                                -- to produce the results shown in table 2.
    make (piece value = &i);    -- Make 10 pieces whose values end up being 1
                                -- to 10.
    call shell();               -- Invoke the shell for conflict resolution and
                                -- rule firing.
}; -- end procedure main1
}; -- end module
```

A.5. NO CONTROL, RHS SEARCH WITH THE RESULT IN A GLOBAL VARIABLE

This program generated the results presented in the fifth row of table 2. The task is to take 10, 50, 100, or 250 integers in separate WMEs (called a "piece" in this example) and add their value to the global variable "result." Each piece is removed from working memory after its value has been added to the result. When shell is called in procedure main1, "rule number1" fires once, searching working memory on the RHS.

```
module setup(main1) {
use o83shl;                                 -- Link.
global &result: integer;                    -- Result is a global variable.
type start_task= element ();                -- Declare a WME type.
type piece = element (value: integer);      -- Declare another WME type.
rule number1 {                              -- This is the only rule in the
    &1 (start_task);                        -- program. It's LHS is activated when
                                            -- a start_task is made but no other
                                            -- WMEs are bound.
-->                                         -- The LHS (beginning here)
local &i: integer, &l: logical, &type: symbol, &value: integer, &size:integer;
&size = wsize();                            -- checks the size of working memory.
for &i = (&size downto 1) {                 -- Starting with the most recent
     &type = wtype(&i);                     -- element the type is checked
     if (&type = piece) {                   -- to see if it is a "piece". If so
        &l = wextract(&value,&i,value);     -- its value is extracted
        &result = &result + &value;         -- and added to result.
        &l = wremove(&i);      };  };       -- The piece is removed as in the other
                                            -- programs.
remove &1;                                  -- Done so remove task starter.
}; -- end rule

procedure main1() {
     local &i: integer;           -- Variables begin with "&".
     &result = 0;                 -- Initialize the global variable.
     make (start_task);           -- Trigger the rule to perform the task.
     for &i = (1 to 10)           -- This number changed from 10, 50, 100, and 250
     make (piece value = &i);     -- Make 10 pieces whose values end up being 1
                                  -- to 10
     call shell();                -- Invoke the shell for conflict resolution and
                                  -- rule firing.
}; -- end procedure main1
}; -- end module
```

A.6. CONTROL VIA CONTEXTS WITH THE RESULT IN A GLOBAL VARIABLE

This program generated the results presented in the sixth row of table 2. The task is to take 10, 50, 100, or 250 integers in separate WMEs (called a "piece" in this example) and add its value to the global variable "result." Each piece is removed from working memory after its value has been added to the result. When shell is called at the end of procedure main1, "rule number1" fires once for each piece.

```
module setup(main1) {
use o83shl;                               -- Link to libraries, and the shell
                                          -- which controls conflict resolution
                                          -- and rule firings.
global &result: integer;                  -- Result is stored as a global
                                          -- variable.
type context= element(value: integer);    -- Declare a WME type. The
                                          -- value field will specify which
                                          -- "piece" should be selected in the
                                          -- next rule firing.
type piece = element (value: integer);    -- Declare another WME type.
rule number1 {                            -- This is the only rule in the
                                          -- program. It's LHS points to a
   &2  (piece);                           -- piece in working memory which is
   &3  (context value = &2.value);        -- specified in the context. Thus
                                          -- strict control.
-->                                       -- The LHS (beginning here) takes the
                                          -- value in the global variable
&result = &result + &2.value;             -- "result" and adds the value in
remove &2;                                -- piece to it. Piece is then removed
modify &3 (value = &3.value +1);          -- and the context value is incremented
}; -- end rule                            -- (simulating starting the next
                                          -- sub-task).

procedure main1() {
     local &i: integer;           -- Variables begin with "&"
     make (result);               -- The result will be in working memory.
     for &i = (1 to 10)           -- This number changed from 10, 50, 100, and 250
                                  -- to produce the results shown in table 2.
     make (piece value = &i);     -- Make 10 pieces whose values end up being 1
                                  -- to 10
     call shell();                -- Invoke the shell for conflict resolution and
                                  -- rule firing.
}; -- end procedure main1
}; -- end module
```

ATE

LMED

8