SUPPLEMENT

Table of Contents

* Includes those papers not received in time to appear alphabetically in the Proceedings.

CONTROLLING KNOWLEDGE DEDUCTION IN A DECLARATIVE APPROACH

Herve Gallaire
ENSAE - CERT
2, Avenue Edouard Belin
31055 Toulouse Cedex
France

Claudine Lasserre
ENSAE
10, Avenue Edouard Belin
31055 Toulouse Cedex
France

This paper is concerned with the problem of introducing user-defined heuristic control over the deduction process in a knowledge based system. Control information is seen as being just another piece of data, separated from the knowledge base the use of which it controls and from the theorem prover. A declarative approach to control expression was retained. Control information is embedded in user's metarules taken from the metalanguage presented in this paper. This metalanguage and a theorem prover interpreting it have been implemented. Examples of its use are given and discussed.

1. INTRODUCTION

The process of deducing information from a knowledge base and from deduction rules is central to many fields : proof theory, problem solvers, expert systems in many areas. Whatever the approach followed, be it more declarative or more procedural, the major problem encountered is that of guiding the deduction process, namely the control problem.

Expressing control over the deduction process is but one aspect of metaknowledge, as shown in L3] ; we give here an application-independent formalism for expressing application-dependent control, which is a way of heuristically guiding a general process. The metarules presented here for that purpose have been developed in a declarative context and as such come close to ideas from [2] ; we were influenced by [5] and [8] too.

The paper is organized as follows. Section 2 presents the framework of this study and the PROLOG system we use for knowledge and metaknowledge expression. Section 3 gives an overview . of the possible levels and means for metaknowledge expression while section U presents the metalanguage we have defined and examples of its use.

Study sponsored in part by CNRS and DGRST.

2. THE DERIVATION PROCESS

2.1. A point of view for deduction control

Reasoning strategies have been of main concern in many fields of A.I investigated at CERT [9], [12]. For instance Data Base systems need to be enhanced with deduction of implicit information from the explicit one or with checking of integrity constraints to be satisfied by the stored data ; CAD systems need problem solving techniques for exhibiting some kind of intelligence in the design process [10], etc.

Most of the systems built at CERT rely on a declarative approach [15]. But we do not deny the practical interest of more procedural approaches [15]. In these studies the choice was guided by the interest in <u>separating</u> knowledge expression from the expression of algorithmic information on how to use it, and in <u>studying this relationship</u> ; a declarative approach, together with the use of logic as a formalism,proved to provide a very interesting basis for that. Its benefits in knowledge representation - ease of specification, clarity, adaptability, ease of modification of the knowledge base, power of the deductive or proof mechanism - are well known. Besides, we don't believe the proof mechanism to be more complex, although it is more powerful, than dynamic pattern-directed invocation ; it is reasonably efficient when improvements are made in specific areas such as clause indexing and retrieval.

As the declarative approach makes it necessary to express "how to use what we know" ie meta-knowledge, it gives a good framework for experimenting not only the different possible levels of meta knowledge expression but its different effects on the same data too. Here stands the basic point of this paper, namely that metarules - ie rules on how to use the knowledge base [2] - can actually he expressed in a declarative mannor separately from the knowledge the use of which they control. It is important to note that the necessity *of* expressing control is not an additional work which could be viewed as a default *of* the declarative approach : as argued in [5], the possibility for the user to mix control to his program in the procedural approach will raise a problem of interpreter control instead of a program control problem.

This study on control of the deduction mechanism rests upon the logical system PROLOG as a formalism and inference mechanism both for rules and metarules. Ideas developed here could as well he carried over to other systems, such as rule-based systems, production systems, provided they oiler leatures similar to the second-order logi c ones that PROLOG gives.

2.2. Notations and interpretations

We shall use the First order predicate calculus (FOPC) under clausal form, restricted to Horn clauses [8] and assume the reader to be familiar with theorem proving [I]. We use the following notation :

(1) +A -BI . . -Bn stands for $(B1\wedge..\wedge Bn)\longrightarrow A$
(2) +A stands for A
(J) -HI.. -Bn stands for $\neg(B1\wedge..\wedge Bn)$

Literals are either positive (eg + A) or negative (eg-Bi). A is the head of the clause, the rest of it is its body.

These Horn clauses can be interpreted in many different ways. Their standard interpretation is : if x,y.. are all variables in the clause, then (1) asserts that for all x,y.. A if BI and ... and Bn ; (2) asserts that for all x,y.. A ; and (3) asserts that for no x,y.. BI and. ... Bn (i.e. $\neg\exists x\exists y..\ B1\wedge..\wedge Bn$). Other interpretations are possible, in terms of problem solving ((1) being an operator, (2) an assertion, (3) a problem), or of programming languages, ((1) and (2) being procedures, (3) a procedure call). We shall stick to the logic language as much as possible in the sequel.

As for the -proof procedure we shall only consider derivation or refutation procedures, based on resolution, restricted to linear strategies starting from purely negative clauses. At each step i of the derivation we are given the first (negative) parent clause Ci (FPCL) ; we have to choose the literal —Cij in Ci as the literal to be resolved upon at this step (Cij is called SEL) ; we also have to choose among all clauses those whose head unifies with Cij (making the UNIFCLSET) ; finally we have to choose in UNIFCLSET the second parent clause (SELCL) ; resolution yields resolvent C'i ; Cij is said to be the parent *of* the literals in the body of SELCL. All the above choices will be guided by a strategy.

2.3. Background on PROLOG

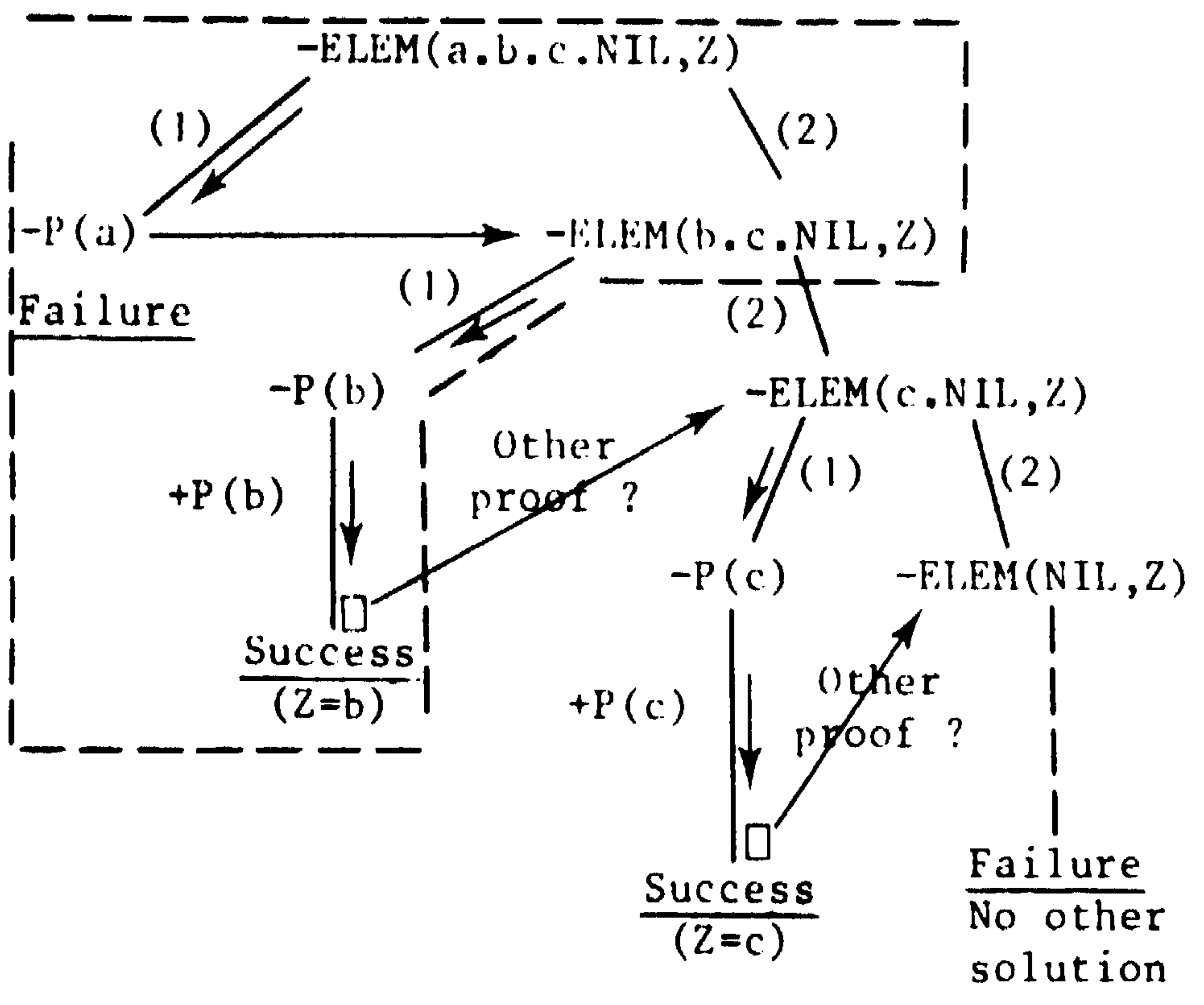
PROLOG [13] is a theorem prover which takes as input a set of Horn clauses of type (1) and (2) above and a theorem to be proved, or actually its negation of type (3). It uses a static strategy complete for Horn clauses, namely LUSH resolution [6]; it is "top-down", starting from the purely negative clause it is given ; it chooses as SEL the leftmost literal in FPCL, working "depth-first", and chooses clauses from UNIFCLSET in the exact order they were input to the theorem prover by the user. Further, the process *is* controlled by an automatic backtracking mechanism activated in case of incorrect clause choice and in order to find all proofs of the alleged theorem. Very important to our work is that, due to LUSH resolution results, any other SEL selection strategy would preserve completeness.

We must note that PROLOG implementation is not complete, due to the unbound depth-first mechanism and to the lack of loop cheeks. But PROLOG exhibits great advantages for metaknowledge expression : it allows using clauses and predicates as terms, i.e as arguments of other predicates ; through built-in predicates it offers access to input-output, to the proof tree (ETAT predicate) and to the various parts of clauses and terms (VAR, UNIV predicates) ; one can also alter the set of axioms (ADD and SUPP predicates) ; it is also possible to alter the standard backtrack step by eliminating return points : any selection of the '/' predicate in a SELCL will suppress all remaining choices for all the already solved literals in SELCL and for their parent.

Example : the following is a set of axioms for proving that Z is an element of list L satisfying predicate P. The '.' is the concatenation symbol and NIL is the empty list.

```
(1)    +ELEM(X.Y,X)    -P(X)
(2)    +ELEM(X.Y,Z)    -ELEM(Y,Z)
```

When used to prove ∃Z ELEM(L,Z) axiom (1) will unify Z and the first element of the given list L ; then if Z satisfies P, the theorem will have been proved (the value of Z being the answer of PROLOG). Due to PROLOG strategy for choosing SELCL, axiom (2) is used in case axiom (1) does not yield the empty clause, i.e when P(X) cannot be proved ; in that case a new theorem is to be proved, namely ∃Z ELEM(Y,Z), where Y is the rest of the list. If +P(b), +P(c) are axioms and the theorem to prove is ∃Z ELEM(a.b.c.NIL,Z), the derivation tree will be :



Recall that PROLOG gives all possible proofs ; hence when success is recorded an alternative is looked for. In this tree the arrows indicate the order in which the nodes of the tree are built. If one wanted only one proof of that theorem, it would be simple to modify (I) and to write it +ELEM(X.Y,X) - P(X) -/ . In that case we would get a tree limited to the one enclosed in the box above.

3. PARAMETERS OF THE DERIVATION PROCESS

A choice strategy rests upon a selection function. Such a function may or may not depend on the object on which it bears ; for instance it may depend on the specific clause from which to extract SEL, or it may not depend on the specific UNIFCLSET from which to extract SELCL. Similarly it may or may not depend on the context of the object on which it bears ; by context we mean the derivation tree or part of it.

The information needed by the selection function may be either implicit or explicitly specified by the user. Autonomous selection functions use implicit information (which may even be void,eg the leftmost-literal-first rule seen in PROLOG) as in the literal-with-the-least-number-of-instantiated-variables-first rule, or in the shortest-clause-first rule, etc. Such functions will not be discussed in this paper, although an interactive choice of such strategies is offered to the user in some systems.

This paper is precisely concerned with the expression of information to be used by selection functions ; we shall only deal with the means of expression, not with designing specific criteria. Important to this work is the notion of context ; to express context information, the user must choose between a <u>local</u> context (part of the proof tree, eg the parent of a literal, its parameters, its descendants) and a more <u>global</u> one (ancestors of a literal, the whole proof tree itself). Although the declarative approach that we follow, especially the logic one, seems to be well suited to that latter choice through the notion of proof, we have not investigated it, thus restricting ourselves to local context expression. Neither shall we discuss the pragmatic control expression that clause writing always conveys [7].

There are two basic ways for expressing control of the deduction process.

a. As is typical of the procedural approach, control information can be mixed with the knowledge base, ie with the application data (clauses) used in the derivation process. For instance the user can modify his set of clauses so that the order of his original clauses in the UNIFCLSET they belong to becomes dependent on SEL. We do not reject such an approach ; its systematic use is reported in [9].

b. More coherently with the declarative approach, one can separate control information from the knowledge base ; at the user level this becomes another piece of data, ie a set of clauses expressing a heuristic control. This is the choice that was made here.

We shall now present our metalanguage for stating the metarules expressing this heuristic control. It uses content-directed invocation [2] heavily- and we feel we have the correct tools for it. Our metalanguage is coherent with the base language [5].

4. A <u>PROPOSAL FOR</u> A <u>METALANGUAGE</u>

There are two types of metarules in this language, expressing control either over the choice of SEL or over the choice of SELCL ; for the user writing them, such metarules are true assertions, in the logical sense, about priorities and orderings as explained below ; they will be used as axioms by the theorem prover.

4.1. Ordering of UNIFCLSET

The METAL metarules to be described here have as primary purpose the expression of assertions on priorities of individual clauses in UNIFCLSET, SELCL being the clause of higher priority. Then let P, R be terms or variables, let L be a term LI,L2...Lr.NIL or a variable (see 2.3 for the '.' notation and for NIL) ; let CI,...Cn be positive literals ; then Metarule

+ METAL(P,L,R,K) -CI ... -Cn

asserts that any user's clause matching (unifying) with clause +P-L1-..-Lr has priority Kin the UNIFCLSET it belongs to, provided R matches the parent of SEL and the existential closure of CIA...ACn can be proved ; in the sequel we talk more simply of the proof of CIA...ACn. We must remark that :

- Li,R,P are terms in METAL but literals in user's clauses (second order featureX

- when any of P, L is a variable, the process of matching will instantiate it ; similarly when R is a variable it is instantiated to the value of SEL ; this feature allows to have metarules applying to many user's clauses, the target clauses, thus avoiding much redundancy, and making it, easier to write metarules by specifying just what is necessary to match the targetclause(s).

- the matching process is constrained not to instantiate any uninstantiated variable of the target clauses or of the parent of SEL at the time matching takes place ; this restriction, enforced by the theorem prover means that a metarule shouldn't be used when it aims at a special case of a target clause. This restriction implies that during the proof of CIA...ACn no variable in P, L,R is instantiated.

- there is a default metarule asserting a MEAN priority value for clauses which aren't target of any metarule : +METAL(P,L,R,MEAN) where P, L,R are variables.

- the set of metarules may be conflicting ie a clause might be the target of two metarules ; as no meta-metarule has been introduced yet, we cannot handle it differently from PROLOG, ie the first metarule that is proved is retained.

- the Ci predicates can be defined by any valid PROLOG set of axioms.

Example : +METAL(P(X,a),L,R(X,Y),0)
-PREDI(X,a) -PRED2(Y)
-PRED3(L,F)

This metarule asserts that priority 0 is the priority of any rule the head of which unifies with P(X,a), the parent of SEL unifying withR(X,Y), provided the conditions PRED are provable. The matching process must comply with the above restrictions - eg as 'a' is constant, X variable, this is not allowed to match P(T,U) where T,U are variables, but P(b,a) would pass the test ; during a proof of PREDI(X,a) it is not allowed to instantiate X more than it is already.

4.2. Choice of SEL

We are to introduce local context evaluation for SEL selection ; there are two selection factors :

- backtracking blocking corresponding to a dynamic '/' introduction

- literal freezing preventing the theorem prover from choosing as SEL a frozen literal ; this feature, the most interesting of all features we present, allows for very dynamic control regimes.

Let P,L,CI,..,Cn be as described for METAL metarules ; let I and F be terms to be described below ; then

+METAX(P,L,I,F) -CI . .. -Cn

asserts that if +P -L) -. . -Lr matches the candidate SELCL, and if CIA...ACn is provable, then :

- if I is the term INH(I1•...Im.NIL) where each Ij matches some Lh, any backtracking on Lh and on its parent will be blocked ; INH (NIL) blocks P only ; if I is NIL no blocking occurs.

- if F is the term FRZ(F1,V1). . . . .FRZ(Fs,Vs).NIL where each Fj matches some Lh, then Lh will be frozen as long as none of its arguments appearing in list Vj is instantiated (ie anything but a pure variable) ; if F is NIL no literal is to be frozen.

We must remark that much of what was noted for METAL metarules applies to METAX ones and is not repeated here, especially for instantiation. Also, there might be cases where all literals are frozen ; we unblock the situation by choosing the leftmost one (default rule). We are now to see examples and to examine more deeply the freezing capability, already investigated but solved differently by Colmerauer [16].

Examples on freezing literals

The following set of PROLOG clauses can be used to prove that two trees X,Y have identical lists of leaves when traversed from left to right :

```
  +IDLF(X,Y)-LF(X,S)-LF(Y,S)
  +LF(X,X.NIL) - ATOM(X)
  +LF(U.V,Z) -LF(U,X) -LF(V,Y) - CONC(X,Y,Z)
  +CONC(NIL,Y,Y)
  +CONC(A.Z,Y,A.V) -CONC(Z,Y,V)
```

If it has been given clauses ATOM(a), ATOM(b), ATOM(c) and if IDLF((a.b).c, a.(b.c)) is a theorem to be proved, PROLOG will succeed ; however the control regime of this proof is quite unsatisfactory for, if it were to fail, much useless work might have been carried out before checking anything. This is a classical example for coroutine need, as a coroutine would allow checking each leaf as soon as it is uncovered in Xor Y. The following metarules, together with the default rule, will do :

```
+METAX(P,L,NIL,FRZ(LF(X,S),S.NIL).NIL)
+METAX(P,L,NIL,FRZ(CONC(U,V,Z),U.Z.NIL).NIL)
```

The first metarule will prevent the theorem prover from choosing LF(X,S) as SEL as long as S is not instantiated ; the second one will do similarly for CONC(U,V,Z) whenever both U and Z are uninstantiated.

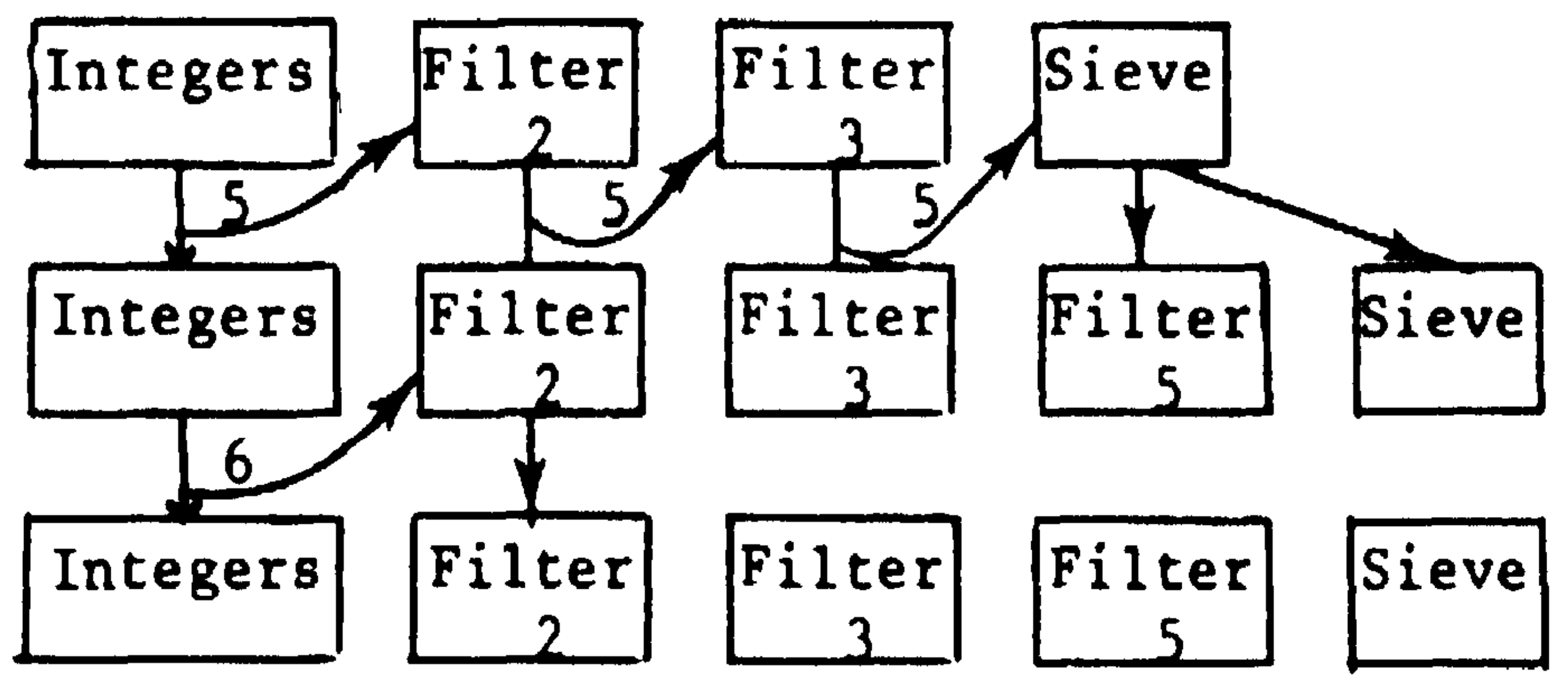
A different and even more interesting example is the classical Sieve of Eratosthenes problem. The following is the list of PROLOG clauses that might be necessary to prove Primes(X) where X is the list of prime numbers

```
+Primes(2.X) -Integers(2.Y) -Sieve(2.Y,X)
+Integers(P.(Q.R)) -Succ(P,Q) -Integers(Q.R)
+Sieve(P.IN,P.PR)-Filter(P,IN,NEW)-Sieve(NEW,PR)
+Filter(P,Q.IN,OUT)-Mult(P,Q)-Filter(P,IN,OUT)
+Filter(P,Q.IN,Q.OUT)-Filter(P,IN,OUT)
```

The Succ(P,Q) predicate is satisfied iff the value of Q is the value of P plus 1 ; the Mult(P,Q) predicate is satisfied iff Q is a multiple of P (definitions not given here for lack of space). As argued in [11] where it is presented, due to PROLOG selection rule this neat example will not work as it is. However it is quite easy to fix with the following :

```
+METAX(X,L,NIL,FRZ(Integers(S.T),T.NIL).NIL)
+METAX(X,L,NIL,FRZ(Sieve(U,V),U.NIL).NIL)
+METAX(X,L,NIL,FRZ(Filter(M,N,K),N.NIL).NIL)
```

Snapshots will explain how the proof proceeds :



4.3 - Implementation principles

There is no major problem in such an implementation, although particular attention is to be paid to allow correct backtracking, fair selection of frozen literals. The theorem prover has been coded in PROLOG mainly for ease of modification and for its unification algorithm. As we piled up levels of interpretation the result is rather inefficient, but our goal is attained. We emphasize how useful both the non first-order features of PROLOG and the possibility to examine its own data structures, i.e. terms and clauses, are. Due to the very simple access to those structures (UNIV, VAR, ...) it is possible to build powerful mechanisms.

Of course this theorem prover is not a major innovation in the field ; one can argue that it behaves with respect to PROLOG in a manner quite similar to CONNIVER^{1}s behaviour with respect to MICROPLANNER. This is basically true from the viewpoint of the theorem prover ; what we gave back to the user is some control over lists of possibilities, over backtrack, over the context mechanism [14]. This similarity is only superficial. We give more to the user because what he can examine in our case is not just a list of possibilities, but it is a list of methods in the CONNIVER sense ; the method body can be scrutinized, modified, reordered whereas it cannot happen with CONNIVER ; this is due to the content-directed invocation feature that we mentioned earlier and of which we saw some important effects in 4.2. We recognize that out context handling mechanisms, which are roughly the SUPP and ADD predicates of PROLOG, are not as powerful as CONNIVER*s. Our goal was different and this remains a possible extension for this work.

5. DISCUSSION

The points we want to stress are the following :

a. User responsibility - Metarules give the user much freedom ; a correct use of metarules goes through an understanding of how the derivation process should go ; then the user will be tempted to use this knowledge and modify his own algorithm so as to take advantage of this understanding directly. This is the reason why we did not give real examples for METAL metarules nor for backtracking use of METAX metarules ; simple examples can easily be fixed without metarules, although this usually introduces much redundancy and awkwardness in the set of axioms. From our studies in plan generation, we are aware of cases where a '/' could be dynamically introduced with much benefit.

b. Extensions - Any time a language is defined, one may propose extensions to it. Among all the extensions we have been thinking of, one is to allow user-defined FRZ predicates ; such a need has been found in another plan generation program where redundant tests could be avoided by properly defined freezing conditions ; such an extension is quite simple to incorporate to our theorem prover.

c. Related topics - Much of the interest of metarules comes from the dynamic ordering of literals (4.2) ; clearly this is related to the many producer-consumer discussions, to the call by opportunity or call by necessity choice ; the Primes problem is discussed in [11] and in a different framework in [4] ; we believe the simple tools given here compare favorably with those given there. We use a data flow mechanism (instantiation) ; more explicit control mechanisms such as message passing have not been looked for. Note that it is possible to define and implement a theorem prover which could take advantage of the connections between variables to select literals ; for instance, in the Prime example the instantiation of a variable unfreezes just one literal, and usually a theorem prover can watch such instantiations. Expressing specific properties of predicates, e.g. that they are true functions, is not covered here although it is an important domain to be investigated. We believe "intelligent" backtracking to be an important topic deserving more attention ; we only note that to be intelligent it must be prepared during the derivation process ; the tools we give are but one step in that direction. Finally interactive control expression has not been covered at all.

ACKNOWLEDGMENTS

We would like to thank A. FAHMY for the many discussions we had with him on this topic and on PROLOG use. P. HAYES comments on this paper are gratefully acknowledged.

REFERENCES

III Chang, C.L. and Lee, R. "Symbolic Logic and Mechanical Theorem Proving" Academic Press, 1973

[2] Davis, R. "Generalized Procedure Calling and Content Directed Invocation." In Proc. Symposium on AI and Programming Languages, Rochester, August 1977, p 45-54

[3] Davis, R. and Buchanan, B.G. "Metalevel knowledge Overview and Applications." In Proc. IJCAI-7j\ MIT, Mass., August 1977, pp 920-927.

[4] Friedman, D. and Wise, D. "Unbounded Computational Structures." Software, Practice and Experience. Vol. 8, pp 407-416 (1978).

[5] Hayes, P. "In Defence of Logic." In Proc. IJCAI-77, pp 559-565.

[6] Hill, R. "Lush Resolution and its Completeness." DCL-Memo 78, University of Edinburgh, Aug. 1974.

[7] Kowalski, R. "Algorithm = Logic ♦ Control." Depart, of Computing and Control. Imperial College, London, 1976.

[8] Kowalski, R. "Logic for Problem Solving" DCL-memo 75, University of Edinburgh, 1974.

[9] Lasserre, C. "Apport de la logique mathematique dans les systemes de decision en robotique." These Universite Paul Sabatier, Toulouse, 1978.

[10] Latombe, J.C. (Ed.) "Artificial Intelligence and Pattern Recognition in Computer Aided Design.' North Holland, 1978.

[11] Mc Cabe, F.G. "Programmer's Guide to IC-PROLOG 0.7." Department of Computing and Control. Imperial College, London, 1979.

[12] Nicolas, J.M. and Gallaire, H. "Data Base-Theory vs Interpretation." In Logic and Data Bases, Plenum Pub. Co, 1978.

[13] Roussel, Ph. "PROLOG : Manuel de Reference et d'utilisation." Groupe d^{1}Intelligence Artificielle. UER de Marseille-Luminy.

[14] Sussman, G. and Mc Dermott, D. "From Planner to Conniver." In Proc. FJCC 1972, pp 1171-1179.

[15] Winograd, T. "Frame Representations and the declarative procedural controversy" In Representation and Understanding. Academic Press, 1975.

[16] Colmerauer, A. Personal Communication.

META-INTERPRETATION OF RECURSIVE LIST-PROCESSING PROGRAMS

Daniel GOOSSENS
Universite PARIS VIII
Route de la Tourelle
75012 PARIS

Presented Is an implemented program understanding system called CAN. CAN relies on a meta-interpretation process which brings to light two new concepts concerning meta-Interpretation of recursive programs: vicious circles and constructive induction. These notions express CAN's ability to extract from program code classes of data for which Interpretation of the code would not stop, and in general, to relate classes of data with abstract values and environments.

keywords: Meta-interpretat Ion, unification, conceptual representat ions, program understanding, propagation, vicious circle, constructive Induction.

1. INTRODUCTION

Works on program verification [10] C133 have brought many useful techniques for proving properties of programs, but all the Implemented systems which derived from them were to be used by expert mathematicians rather than expert programmers. That is, those systems may do nothing with a program on its own. They are intended to be interactively used by programmers who are acquainted with formal specifications manipulations and who understand their programs.
Recent works on program understanding have demonstrated the need of domain dependent knowledge [7] [153 and other types of pragmatic knowledge [19] [203. However, systems in these areas are not equally efficient. More precisely, the systems which support these works may fall on simple tasks which are beyond the scope of their effective specificity. That is, their domain of application is far from fitting with what an expert programmer would consider as "simple", or "visually understandable".
CAN is a powerful tool for such systems since:

- It understands programs independently of traditionnal applications for understanding (verification, correction, Improvement)

- It Is particularly well suited for extracting on its own only what Is "visually understandable" in a program. This prevents CAN from getting stuck on intricate problems. This is a necessity if CAN is expected to be a tool for problem solvers. A part of CAN has been especially developed for the visual understanding of programs which manipulate flat lists.

CAN meta-interprets program bodies in abstract environments [1] [2] [4] [5] [9] [14] [19] [21]. We use conceptual representations [21] in order to represent abstract values of program variables and relations among them. Our conceptual representations are either patterns, or predicate-style assertions which may not be translated into patterns. Our patterns involve element-type and segment-type variables. segment-type variables include Index mathematical notations for flat lists. They also Involve auto-references.
Conceptual representation of meanings Is our bypass of the tradltionnal "set of properties". Conceptual representations may be confronted and compared more easily than assertions, most often with unification procedures, which yield valuable and non-trivial information [6].
CAN involves a sophisticated unification process of conceptual representations.

2. META-INTERPRETATION

When confronted with LISP code, the meta-interpreter most often faces the case where a function is applied on its arguments.
The meta-Interpretation of (F AI A2 ... An) involves the following steps:

first, CAN meta-interprets each of the Ais which should be evaluated on a simple interpretation. this yields a set of possible abstract values and environments for each Ai.
Second, CAN confronts each combination of these values and environments with F's CAN definition. After this confrontation, CAN knows:

- What conditions must hold on the Ais and the environment so that F may apply

- which features of the abstract values and the environment are relevant for the contruction of F's abstract value and side-effects.

Third, CAN propagates this information over the current environment and on the "prototype" abstract value which is part of F's CAN definition.

LISP functions are conceptually defined [21] within CAN. Here are the conceptual definitions of the CAR, CDR and MEMBER functions:

("!" denotes an element variable and "?" a segment variable)

```
(CAR ()) = ()
(CAR (!a ?b)) = !a
(CDR ()) = ()
(CDR (!a ?b)) = (?b)
```

$(\text{MEMBER } !x\ (!a_1 \ldots !a_n\ !x\ ?b)) = (!x\ ?b)$

$(\text{MEMBER } !x\ (!a_1 \ldots !a_n)) = ()$

(where: $!a_i \neq !x$; $i[1,n]$)

That is, the CAR function returns the first element of a list, the CDR function returns the list without its first element. (MEMBER X Y) tests wether the element X is contained in the list Y, at the top-level. If it is the case, MEMBER returns the part of the list which starts from the first occurence of X.

CAN also handles the cases where F is a control structure, a functional variable, an escape label, or an unknown atom. The meta-interpretation of a LISP program is expected to yield a conceptual representation of its meaning.

Example:

Suppose that F is MEMBER, and that the meta-interpretation of its two arguments gave the two abstract values: (on !1 !2) and (?3 !1).The application of MEMBER on "(on !1 !2)" and "(?3 !1)" amounts to checking whether the list (?3 !1) contains the structure (on !1 !2). The confrontation of these values with MEMBER's prototype values is performed by a unification process of conceptual representations [6].

The unification of

$[\ !x\ \ (!a_1 \ldots !a_n\ !x\ ?b)]$

with $[(\text{on } !1\ !2)\ \ (?3\ !1)\ ]$
(where $!a_i \neq !x$) gives only one case:

$!x = (\text{on } !1\ !2)$
$?3 = !a_1 \ldots !a_n\ (\text{on } !1\ !2)\ ?b1$
$?b = ?b1\ !1$

since !1 in (?3 !1) may not contain (on !1 !2).

The propagation of this information on the prototype value (!x ?b) of MEMBER gives the returned abstract value: ((on !1 !2) ?b1 !1). This value is still functionally related to the environment since ?3 is also replaced in the environment by what it is associated to.
The reader is invited to perform the same task for the second rule of MEMBER's definition.

3. UNIFICATION OF CONCEPTUAL REPRESENTATIONS

The problem is, given two conceptual representations, to find the complete set of its unifiers (it's most general unifier [16] [17] [18]).

For theorem proving purposes, The task of unification may be limited to checking wether two structures can fit or not [11]. In that case, it suffices to find one unifier or show that there is none. Rather, CAN needs a conceptual representation of the complete set of unifiers of two conceptual representations, since it needs to know not only if they fit together, but under what exact constraints [5].

In the case where segment variables are restricted to ?-type variables (that is, when index notations are excluded), the problem amounts to solve an equation in a free monoid with a neutral element (the empty sequence). Our implemented unifier exhibits about the same performances as [16] and [17] (the neutral element makes very few differences). It involves too the algorithm for A-C Unification [12] [18] and a splitting algorithm [17] extended for index mathematical notations.

What is new within CAN's unification procedure concerns the unification of patterns which include index notations, auto-references, element-type variables with restricted domain, and negation.
A detailed description of its actual performances is given in [6].

4. META-INTERPRETATION OF ITERATIVE AND RECURSIVE PROGRAMS

At some stage of the process, the following program:

```
(DE MEMBER (X L)
    (COND ((NULL L) NIL)
          ((EQUAL X (CAR L)) L)
          (T (MEMBER X (CDR L)))))
```

is transformed into:

```
1  (MEMBER !x ()) = ()
2  (MEMBER !x (!x ?b)) = (!x ?b)
3  (MEMBER !x (!y ?b)) = (MEMBER !x (?b))
           (where !x ≠ !y)
```

This says among other things that if MEMBER's first argument is different from the CAR of its second argument, then the value is the recursive call: (MEMBER !x (?b)). If CAN were to go on like previously, it would confront the arguments !x and (?b) with each of the three cases. In fact, this is exactly what CAN will do, but more carefully, so as to avoid looping forever.
Application of rule 3 on !x and (?b) needs the following preconditions:

```
!x = !x'
?b = !y ?b'        Ⓐ
```

in which case the value is (MEMBER !x' (?b')). But this is what CAN was meta-interpreting (modulo a change of the variables'name). We consider Ⓐ as a "vicious circle". The meta-interpretation process stops at each vicious circle, and CAN "multiplies" it by a fictive number n of recursions. From Ⓐ, CAN is able to obtain abstract representations of flat lists:

$$!x = !x$$
$$?b = !y_1 \ \ldots \ !y_n \ ?b$$
$$(\text{where } !y_i \neq !x \text{ and } n \neq 0)$$

This says that if the second argument of MEMBER is a list whose n first elements are different from its first argument, then they will be deleted after the nth recursive call. The meta-interpreter will then go on with the two first rules of MEMBER, and obtain the final conceptual representation of MEMBER which we used in the second paragraph.
The concept of vicious circle is important since it does not coincide with the first recursive call in a program.
For the following example:

```
(DE FOO (L)
   (IF (CAR L) (FOO (CONS (CDR L)))
       (CDR L)))
```

the meta-interpreter will never reach a vicious circle, and yield the final conceptual representation:

$$(FOO\ ()) = ()$$
$$(FOO\ (()\ ?b)) = (?b)$$
$$(FOO\ (\underline{!a})) = ()$$
$$(FOO\ (\underline{!a}\ \underline{?b})) = ()$$

(underlined variables may not be assigned the NIL value)

5. VICIOUS CIRCLES

A vicious circle is encountered when the meta-interpretation of some form f in an environment E1 leads to the meta-interpretation of f in an environment E2 such that E1 and E2 "may be identified".
Formally, the two environments will be identifiable if their unification gives a unifier which contains no "transmitter". A transmitter is a couple of variables bindings of the form:

$$v1 = f(v2)$$
$$v2 = H$$

With the restriction that if f(v2) contains v1 and H contains v2, the couple is considered as a vicious circle.

Here are some examples of LISP programs where the first recursive call does not coincide with a vicious circle:

```
(DE FOO (X Y) (FOO Y X))

(DE FOO (L M)
   (IF (NULL L) NIL
       (FOO (CDR M) (CONS (CAR M) L))))

(DE FOO (L M P Q)
   (COND ((NULL L) T)
         ((EQUAL (CAR L) M)
          (FOO (CDDR L) P Q (CADR L)))))
```

6. CONSTRUCTIVE INDUCTION (multiplication by "n")

The notion of vicious circle is motivated by the fact that if a substitution (unifier) is a vicious circle, it may be "multiplied by n" by a simple algorithm (i.e its general form after "n" recursive calls may be constructed).
From a variable binding of the form: v = f(v) the algorithm will obtain: (* v n f(v)) which may be seen as a complex variable whose name is v and f(v) is a conceptual representation with auto-references (everywhere v is). In other words, f(v) restricts v's domain. "n" names the fictive number of successive recursive calls. It relates different complex variables to each other.
In some cases, the induction algorithm may generate mathematical index notations such as those involved in MEMBER's CAN definition, instead of conceptual representations with auto-references. i.e in the case:

$$v = -A-\ v\ -B-$$

the algorithm will obtain:

$$v = -A-_1\ \ldots\ -A-_n\ \ v\ \ -B-_n\ \ldots\ -B-_1$$
$$(\text{where } n \neq 0)$$

(where -A- and -B- are any sequence of conceptual representations. They may involve indexed elements and even sequences whose length depend on the index)

The advantage is that the unification process of conceptual representations is particularly well suited for the unification of such forms.

7. RECURSIVITY

From the program:

```
 (DE VECMAT (L M)
    (IF (NULL L) M
        (DCONS (CAR L) (VECMAT (CDR L)
                               (CONS NIL M)))))
```

where

$$(\text{DCONS } !x \; ((?a_1)\ldots(?a_n))) = ((!x \; ?a_1)\ldots(!x \; ?a_n))$$

The meta-interpreter will get:

1 $(\text{VECMAT } () \; !m) = !m$
2 $(\text{VECMAT } (!1 \; ?2) \; (?3)) = ((!1 \; ?a_1)\ldots(!1 \; ?a_k))$

and $((?a_1)\ldots(?a_k)) = (\text{VECMAT } (?2) \; (() \; ?3))$

Recursivity will be handled by the same processes.
as the unification of $((?a_1)\ldots(?a_k))$

and $((!1 \; ?a'_1)\ldots(!1 \; ?a'_{k'}))$

yields a vicious circle, the induction algorithm is applied. here, from:

$$?a_i = !1 \; ?a_i$$

CAN gets:

$$?a_i = !1_1 \ldots !1_n \; ?a_i$$

By propagation and with the initialisation:

$$((?a_1)\ldots(?a_k)) = (()_1 \ldots ()_n \; ?3)$$

(the right conceptual representation is the abstract value of M when case 1 is encoutered. The initialisation is performed via the unification of the two conceptual representations and propagation of the results.)
CAN finally obtains the conceptual representation of vecmat:

$$(\text{VECMAT } (!1_1 \ldots !1_n) \; ((?b_1)\ldots(?b_m)))$$

$$= ((!1_1 \ldots !1_n)_1 \ldots (!1_1 \ldots !1_n)_n$$

$$(!1_1 \ldots !1_n \; ?b_1)\ldots(!1_1 \ldots !1_n \; ?b_m))$$

8. TERMINATION

The meta-interpretation of a program yields a set of rules which contain recursive calls, as a first approximation. As uas shown, the same meta-interpretation process may be pursued. So, CAN may find which rules data can "go" to, when "coming out" of a recursive rule. That is, find which rules refer to which rules.
If a recursive rule refers only to itself, or if a group of rules which refer to one another refers to no rule "outside", then a class of data is determined for which simple Interpretation of the program never ends. Here are some simple examples:

example 1:

```
(DE FOO (X Y)
   (IF (ATOM X) (FOO Y X) X))
```

CAN yields:

```
(FOO  !list  !b) = !list
(FOO  !atom  !list) = !list
(FOO  !atom1 !atom2) = infinite-loop
```

example 2:

```
(DE FOO (L M)
   (IF (NULL L) M
       (FOO (CDR M) (CONS (CAR M) L))))
```

CAN yields:

$(\text{FOO } () \; !m) = !m$
$(\text{FOO } (\underline{?1}) \; ()) = (() \; \underline{?1})$
$(\text{FOO } (\underline{?1}) \; (!2)) = (!2 \; \underline{?1})$
$(\text{FOO } (\underline{?1}) \; (!2 \; ?3)) = \text{infinite-loop}$

example 3:

```
(DE FOO (X L)
  (IF (EQUAL X (CAR L))
      L
      (FOO X (CDR L))))
```

CAN yields:

$(\text{FOO } () \; (!a_1 \ldots !a_n)) = ()$
with $!a_i \neq ()$
$(\text{FOO } \underline{!b} \; (!a_1 \ldots !a_n)) = \text{infinite-loop}$
with $!a_i \neq \underline{!b}$
$(\text{foo } !c \; (!a_1 \ldots !a_n \; !c \; ?d)) = (!c \; ?d)$
with $!a_i \neq !c$

(underlined variables may not be replaced by the NIL value)

Here is an example of a different kind:

```
        (DE FOO (L M)
           (IF (EQUAL L M) T
                (FOO (CDR L) (CONS (CAR L) M))))
```

CAN yields:

```
    (FOO !1 !1) = T
    (FOO () (?2)) = infinite-computation
    (FOO (!1  ... !1  !1  ... !1  ?2) (?2)) = T
           1       n   n       1
and
(!1  ... !1  !1  ... !1  ?2) ≠ (!1  ... !1  ?2)
   i+1    n   n       1            i     1
i[1,n]
```

For this last example, CAN is not able to explicitly describe the class of data for which there is infinite computation. It is implicitly defined by the set of cases which do not fit the rules above, plus those which fit the second rule.

9. USING CAN: THE INTERACTIVE PROGRAMMER ASSISTANT.

CAN is an interactive program understanding system. When CAN is run, the user may type in LISP functions, or LISP expressions. When an expression is typed in, CAN gives its abstract value. There may be several cases. If the expression is (ATOM X) for instance, CAN will respond:

CASE 1: X=!atoml   VALUE: t

CASE 2: X=!¬atoml VALUE: nil

The user can focus on any of the cases or keep them all.
A practical use of the CAN system, then, is the following: the user first types in a program he has just written. Once the program is understood by CAN, the user may type in any LISP expression involving calls to the program. He will get the corresponding abstract values. Note that CAN is a simplifier. That is, if it is not able to understand an expression E, E's abstract value is E itself (with an implicit meta-evaluation request). In other words, E may not be simplified further.
If the expression is an assertion about the program, it may meta-evaluate to T or NIL, but CAN's performances in this domain still raise the need for a theorem prover.
One common use is to type a call to the program on restricted classes of data. i.e if the program has name FOO and argument X, the user may define a unary LISP predicate P which is true for a restricted class of X's, then focus on the cases where (P X) may evaluate to T, and call (FOO X).

For instance, if FOO is in fact a pattern-matching function MATCH with back-referencing (two occurences of the same variable must be assigned the same value) and the user wants to test wether back-referencing works fine, he may first build several kinds of abstract values for MATCH'S arguments by the "focus on" technique described above, and then call MATCH on them

One future work will be to develop a system for proving theorems about list structures by structural Induction, which uses CAN as a simplification sub-system.
We expect that CAN will greatly help the task of proving theorems by structural induction since, for instance, many of the LISP theorems proved in [22] already meta-evaluate to the LISP truth-value T when given to CAN.

10. CONCLUSION

We have designed a system whose job is to understand what an expert programmer could consider as "visually understandable". CAN does no problem solving. CAN's meta-interpretation process Is straightforward, end simple interpretation is a simple case of meta-interpretation [5]. Ue then- think that CAN should be put in the service of expert programmers, or program understanding systems which would be specialised in narrow areas, since conceptual representations are much more easily manipulated than LISP code, for instance. It may effectively be expected that the size of program understanding systems is directly related to the number of ways of representing a relation within a programming language.
The CAN system is written in VLISP-10 [3] C81 and occupies about 20K words.

REFERENCES

[1] BALZER, GOLDMAN & WILE (1977) "Meta-evaluation as a tool for program understanding" 5th IJCAI, MIT Cambridge, August 1977

[2] BOYER, ELSPASS & LEVITT (1975) "SELECT--A formal system for testing and debugging programs by symbolic execution" Int. Conf. on Reliable Software, April 1975, pp. 234-245.

[3] CHAILLOUX (1976) "VLISP-10 manuel de référence" Dept. d'informatique, Univ. PARIS 8, RT 17-76.

[4] GOOSSENS (1978) "A system for visual-like understanding of LISP programs" A.I.S.B. Conf. Hamburg, July 1978.

[5] GOOSSENS (1978) "Compréhension visuelle de programmes controlée par méta-filtrage" Groplan: Bulletin de l'AFCET, Groupe Programmation et langages, 1978

[6] GOOSSENS (1978) "L'unlflcetlon au service de le comprehension" RT-14. Dept. d'Informatlque, University de Ylncennes.

[7] GREEN & BARSTOU (1975) "A hypothetlcel dialogue exhibiting e knowledge bese for e progrem understanding system" Stanford A.I. Lab. MA 258, Cpt Science Dept. Report n. Stancs-75-476.

[8] GREUSSAY (1977) "Contribution a la definition interpretative et a L'implementstlon des X-tengages" These, University PARIS 8.

[9] HEUITT & SMITH (1975) "Towerds e programming apprentice" IEEE Trans, on soft. engineering, Vol. SE-1. pp. 26-45.

[10] HOARE & LAUER (1973) "Consistent and complementary formal theories of the sementics of programming languages" Ibid 3, pp. 145-182.

[Il] HUET (1975) "A unification algorithm for typed lambda-calculus" Theorltlcel computer science 1. pp 27-57

[12] HULLOT J.M. (1979) "Assoc iatlve-commutatlve pattern-matching" 6th IJCAI. Aug 1979. Tokyo.

[13] IGARASHI, LONDON, LUCKHAM (1975) "Automatic program verification I: A logical basis and Its Implementation" Acta Informatlea, Vol. 4, pp. 145-182.

[14] KING (1975) "A new approach to program tasting" Int. Conf. on reliable software, April 1975, pp. 228-233.

[15] RICH & SCHROBE (1975) "Understanding LISP programs: Towards a programming apprentice" Master's Thesis, EECS M.I.T.

[16] SIEKMANN (1975) "String unification" Essex university, Memo CSM-7.

[17] SIEKMANN LIVESEY (1975) "Termination and decidability results for string unif Icatlon'' Essex university, CSM-12.

[18] STICKEL (1975) "A complete unlflcetlon algorithm for assoclatlve-commutetlve functions" 4th Ijcal, Tbilisi Georgia USSR. sept. 1975.

[19] UERTZ (1978) "Un system© da comprehension, de programmes incorrects^M Proc. 3eme colloque sur la programmetlon, Paris B. Rob Inst ed. pp 31-49.

[20] WERTZ (1978) "Un systeme de comprehension, d'amelioration, et de correction de programmes incorrects" These de 3eme cycle, Univ. P. et M. CURIE.

[21] YONEZAUA & HEWITT (1976) "Symbolic evaluation using conceptual representetions for progrems with side-effects" M.I.T., A.I. Leb., AI-Memo 399.

[22] BOYER & MOORE (1973) "Proving theorems about LISP functions", Stanford University, Stanford CA. 486-493.

HUGE PROCESSING BY EXPERIMENTAL ARRAYED PROCESSOR

Hitoshi Matsushima Takeshi Uno Hasakazu EJiri

Central Research Laboratory
Hitachi Ltd.,
Higaahi-koigakubo 1-280,
Kokubunji, Tokyo l85, Japan

An image processor based on an arrayed structure was implemented.. It has five design features: (1) memory unit separation into a picture memory for images and a control data memory for programs, (2) processing unit separation into a parallel processing part and a serial processing part, (3) parallel processing by an arrayed structure, (if) use of a special micro-processor for an array's processing element, and (5) direct control of parallel processing by programming. It deals with images consisting of 256 x 256 pixels and processes them by sixteen processing elements. The original images are given by a TV camera* The processed images are outputted to CRT display equipments. The basic operations in the array are data conversions, local operations, sequential processings, manipulations between two images and so on. *By* using those operations, many kinds of image processing can be done in this processor.

1. INTRODUCTION

Many proposals for pattern processing have been made to date, and soma have been practically implemented, especially for certain industrial and medical applications, in which special processors such as image analyzers are in practical use* However, most image processors cam only deal with binary images..[l]

On the other hand, recently large-scale pattern processors have been studies in some research laboratories.*[2]* However there are still problems in developing image processors for multi-leveled Images Tor example, processors tend to become too big and expensive..

However, recent developments in commercially available IC's have had a profound effect on the size* speedy and cost of hardware- In addition* the need for general purpose image processors Is getting stronger. Thus, the image processor presented here was experimentally developed*. The outstanding feature in our Image processor IP is parallel processing ability.

PM * nanor riAftcrlhfid the ttfincinles and

Z. DESIQM PLINCIPLSS

The main points considered in establishing our IP are discussed below.

Information included in Images depesds on the two-dimensional arrangement of pixel^{1}6 values.. So two-dimensional memory access is employed for data utilization. Therefore, the memory was separated into two distinct parts, one for images and the other for control data. The former has useful functions for parallel processing of images.

This work was contracted with the Agency of Industrial Science and Technology, Ministry of International trade and Industry, as a part of the National Research and Development Program "Pattern Information Processing System?

In image processing, the same operations are repeated many times* In these cases, increasing processing speed by ordinary computer architectures is not efficient. Therefore, the execution unit in our IP was separated into a conrentional mini-computer's central processing unit and a parallel processing part. The former mainly deals with control data and sometimes processes pixels one by one, while the latter usually processes pixels by the parallel processing.

In the parallel processing part, an arrayed structure was adopted, mainly to increase processing speed* In pre-processing of feature-extraction of images the same operator is executed at every sampling point* For such position-invariant processes. an arrayed structure is very useful. Many sampling data are inputted to many processing elements at a time. Then all processing elements execute the same operation simultaniously. Increasing the number of processing elements by a certain factor should cause the processing time to decrease by the same factor.

One cause reducing the effect of an arrayed structure is the occurrance of irregular flow such as conditional jump. In such cases, all processing elements can not be controlled by one sequence. Some PEs will have to be in a halt state. To avoid this problem and increase efficiency of parallel processing, a special micro-processor was developed which can absorb irregularities and do such special calculations as summing products, table look up and getting namming-distance at high speed.

It is difficult to control the array. Therefore a usage of micro-instructions was planned to control executions in a processing element and data's flows between the array and a picture memory.

3. SYSTEM OUTLINE

A block diagram of our IP is shown in Fig* 1. It consists of a parallel processing part and the other part. The former comprises a processing element array(PE Array), a picture memory(PM), local registers (LR) a micro-instruction control(MC) and input/output control(IOC).

Image data are inputted through the IOC and stored in the PM. They are originally obtained 6 bits in the IOC. The image data are then transmitted to the PE Array through the LR. The LR can store a 4 x 4 local-image or an 8 x 8 one. The PE Array processes it parallelly, and the results are stored again in the PM.

The PM is a memory for images consisting of three banks of 256 x 256 pixels. The access to the PM is controlled by micro-instructions in a program* The accessing unit is a 4 x 4 square with the upper-left corner address (4n,4m), or one pixel. The letters n and m stand for integer from 0 to 63.

The PE Array consists of sixteen processing element(PE). Each PE is a special micro-processor comprising a controller, which interprets commands and controls calculations in synchronization with an external clock. The commands comprise addition, subtraction, multiplication, summing products, logical operation,

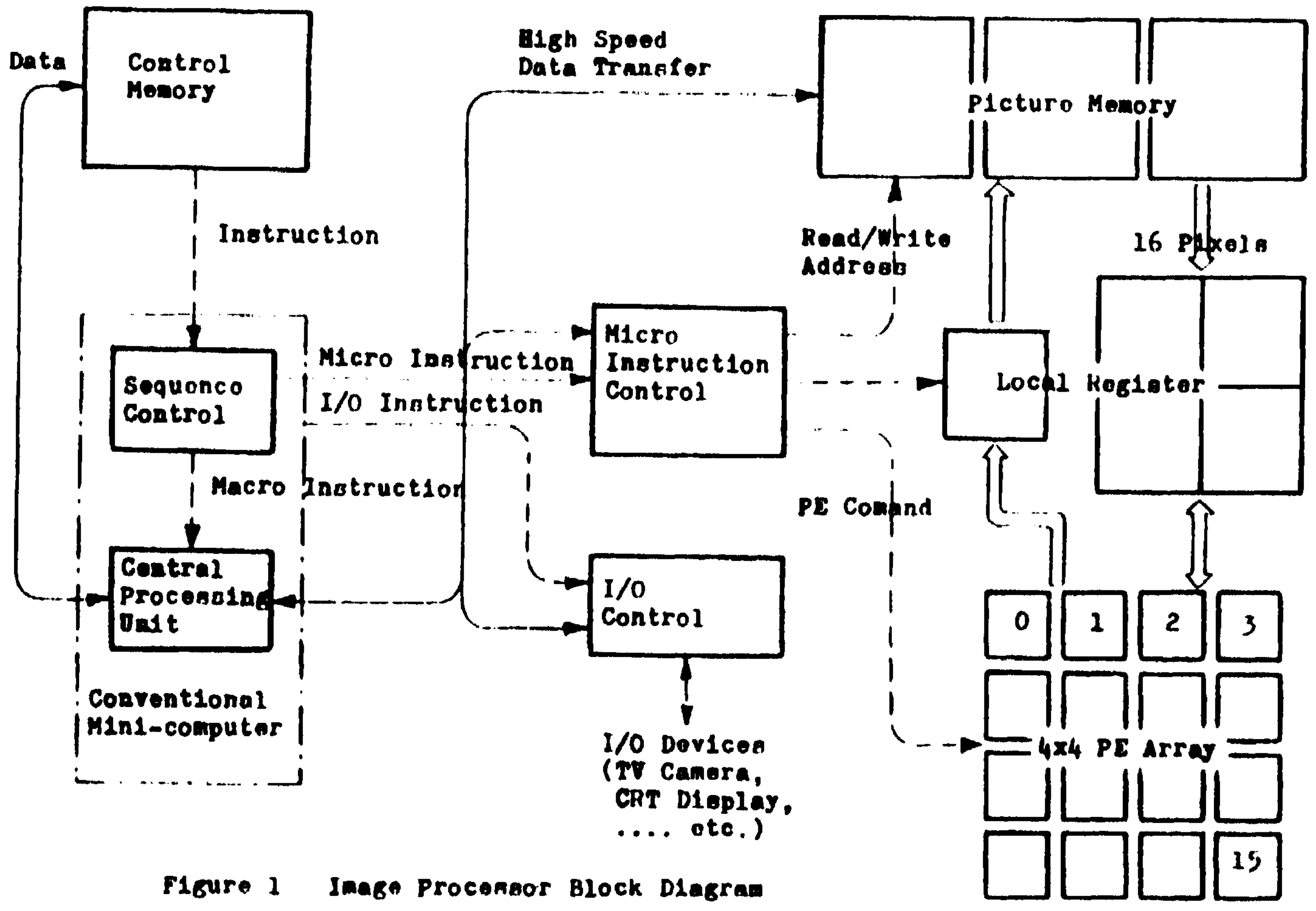


Figure 1 Image Processor Block Diagram

table look up and so on. Each PE has a 256 words 16 bits scratch pad memory. The scratch pad memory can store a sequence of PE commands. Therefore each PE can execute its own program. The depth of calculations is twelve bits long. The execution time is 0.35 #$ and the cycle time of the PM is 0.7 >uS.

The LR has shift functions in an 8 x 8 local image. The PE Array can obtain the free 4 x 4 square pixels from the PM by these functions. Also, PEs can comunicate each other through the LR.

4. BASIC OPERATIONS

In order to describe functions in the PE Array, several notational definitions are given as follows;

D(x,y); a value of pixel(x,y) in the original image.

$$\mathbb{D}(x,y)=\begin{bmatrix} D(x,y), & \cdots & ,D(x+4,y) \\ \vdots & \vdots & \vdots \\ D(x,y+4), & \cdots & ,D(x+4,y+4) \end{bmatrix}$$

R(x,y); a result by a certain operation in a PE of the PE Array.

F(),G(); functions exécuted by a combination of PE's commands.

In the PE Array, sixteen PEs process local images in the LR simultaneously. Therefore sixteen results $\mathbb{R}(x,y)$;

$$\mathbb{R}(x,y)=\begin{bmatrix} R(x,y), & \cdots & ,R(x+3,y) \\ \vdots & \vdots & \vdots \\ R(x,y+3), & \cdots & ,R(x+3,y+3) \end{bmatrix}$$

are obtained simultaneously.

The representative basic operations in the PE Array are as follows.

(a) Data conversion.
$R(x,y)=F(D(x,y))$
This is used for thresholding, contrast stretching and so on. In a thresholding of a 256 x 256 image, it takes about 11.5 mS.

(b) Local operation.
$R(x,y)=F(\mathbb{D}(x,y))$
For example, in a case of two-dimentional filterings the results are as follows.

$$R(x+m,y+n)=\sum_{i=0}^{4}\sum_{j=0}^{4} f(i,j)\times D(x+i+m,y+j+m)$$
$$0\leq m\leq 3;\quad 0\leq n\leq 3$$

where f(i,j)s are weight coefficients.

In all PEs, twenty-five times' executions of the PE command "SUM OF PRODUCT" are done. It takes 34.4 mS in a 256 x 256 image. By this local operation, spacial differentiation, noise elimination. template matching, and so on are available.

(c) Sequential processing.
$R(x,y)=F(\mathbb{D}(x,y),\mathbb{Q}(x-s,y-t))$
$\mathbb{Q}(x,y)=G(\mathbb{D}(x,y),\mathbb{Q}(x-s,y-t))$
Where $\mathbb{Q}$ is PE's internal states specified by several flags and contents of the scratch pad memory. The point (x-s,y-t) shows a preceding processing point to point (x,y). Values of letter s and t depend on a scanning in the PM.
This calculation is used for histogram's calculation, area counting and so on. For example, the processing speed of obtaining pixel value's distributions is 28 mS.

(d) Manipulation between two images.
$R(x,y)=F(\mathbb{D}_1(x,y),\mathbb{D}_2(x,y))$
where $\mathbb{D}_1(x,y)$ and $\mathbb{D}_2(x,y)$ show values of different images' pixels.
For example, it takes about 18 mS to get an add-image or exclusive OR-image from two images.

5. CONCLUSION

An image processor based on an arrayed structure was implemented. It deal6 with images consisting of 256 x 256 pixels and processes them by a 4 x 4 processing elements[1] array. It was constructed in 1977 and used to study processor's architecture for the object recognition, ny these studies, it has been established that the array of our IP is useful for high 6peed processing.

The basic operations in the array are data conversions, local operations, sequential processings, manipulations between two images.and so on. By them, many kinds of image processing can be done in this array

REFERENCES

[1] M. J. E. Golay, "Hexagonal Parallel Pattern Transformations", IEEE Trans. Comput., Vol. C-18, pp., 733-740, Aug. 1969
[2] K. S. Fu, "Special Computer Architectures for Pattern Recognition and Image Processing An Overview", Proc AFIPS, Vol. 57,pp.1003-1013, June 1978.

AUTHOR/CO-AUTHOR INDEX

Note: Volume II begins with page 610.
Papers are arranged alphabetically by first author - A-M, Vol. I; N-Z, Vol. II.
A supplement containing papers received late is found at the back of Vol. II.

TOPIC INDEX FOR TECHNICAL PAPERS

NOTE: Papers are listed by first authors only.

TOPIC INDEX FOR TECHNICAL PAPERS

TOPIC INDEX FOR TECHNICAL PAPERS

TOPIC INDEX FOR TECHNICAL PAPERS

R_026