APPLICATIONS SOFTWARE AND ORGANIZATIONAL CHANGE: ISSUES IN THE REPRESENTATION OF KNOWLEDGE

Ronald M. Lee

February 1983
WP-83-30
(Supercedes WP-80-182)

INTERNATIONAL INSTITUTE FOR APPLIED SYSTEMS ANALYSIS
2361 Laxenburg, Austria

CONTENTS

I. THE PROBLEM: SOFTWARE FOR ORGANIZATIONAL CHANGE 1

II. ANOTHER PROBLEM: TRANSPORTABILITY OF KNOWLEDGE 4

III. THE PROBLEM WITH PROGRAMS:
PROCEDURAL LANGUAGES VS. PRODUCTION SYSTEMS 7

IV. THE PROBLEM WITH DATA:
DATA FILES VS. PREDICATE CALCULUS 17

V. COMBINING THE APPROACHES:
PRODUCTION SYSTEMS AND PREDICATE CALCULUS 24

VI. CONCLUDING REMARKS 25

REFERENCES 27

APPLICATIONS SOFTWARE AND ORGANIZATIONAL CHANGE: ISSUES IN THE REPRESENTATION OF KNOWLEDGE

Ronald M. Lee

I. THE PROBLEM: SOFTWARE FOR ORGANIZATIONAL CHANGE

It is a commonplace observation that organizations, to survive, must adapt to changes in their environment. Those that do not are forced out of business, if they are companies in a competitive market; have their budgets canceled,in the case of government bureaucracies; or are overthrown, in the case of governments themselves.

Just how an organization should be designed to accommodate change is, of course, a much more difficult matter, and has been the subject of many volumes of organizational theory. One aspect of this general problem that seems to have been neglected, namely the effect of information technology on the organization's ability to adapt and change.

Certainly, there are numerous clear cases where the installation of an information system adds to the organization's flexibility. For instance, the installation of a centralized database may allow data to be accessed and combined in a variety of ways that would have been practically impossible when that data was recorded in paper files scattered throughout the company.

The flexibility of a given computer application obviously depends on the foresight of its designers. To this end, programming students are generally taught to seek the most general definition of the problems they are given so that the resulting program can handle not only the immediate problem but also variants of it that might arise.

This strategy has obvious limitations. In seeking to find a generalized solution, the programmer may waste undue amounts of time on conditions that will never arise. He/she must therefore make a choice as to how much flexibility to encode into the program logic. We refer to the level of flexibility chosen as the 'designed flexibility' of the system.

Selecting the appropriate level of designed flexibility is however difficult and, almost certainly, new requirements will later arise that were not planned for originally, so that the program must be modified. This is where the problem arises.

Anyone who has written even small programs will know that it is much easier to incorporate a given feature in the program logic in its original writing rather than try to add this feature afterwards. This difficulty rises exponentially with the complexity of the original program or system. (By 'system' is meant a collection of programs and data files with interdependent functions.) Indeed, the cost and effort of modifying such systems often exceeds that of their original development. For instance, Wulf (1977) refers to:

> the extreme difficulty encountered in attempting to modify an existing program. Even though we frequently believe that we know what we will want a piece of software to do and will be able to specify if precisely, it seems to be invariably true that after we have it we know better and would like to change it. Examination of the history of almost every major software system shows that so long as it is used it is being modified! Evolution stops only when the system is dead. The cost of such evolution is almost never measured, but, in at least one case, it exceeded the original development cost by a factor of 100.

Altering existing computer systems is not only expensive, it is also risky. De Millo, et al. (1979) noted:

> Every programmer knows that altering a line or sometimes even a bit can utterly destroy a program or mutilate it in ways we do not understand and cannot predict...

Indeed, beyond expense and risk, there seems to be an eventual limit to the number of modifications these systems can undergo. Winograd (1979) remarks

> Using current programming techniques, systems often reach a point at which the accretion of changes makes their structure so baroque and opaque that further changes are impossible, and the performance of the system is irreversibly degraded. (p.392)

To summarize, the basic problem with current application systems is that they are 'brittle'; i.e., they cannot easily be reformed to adapt to changing circumstances. This brittleness has profoundly disturbing consequences as more and more organizations, ranging from small and medium size companies to immense governmental agencies, convert their information processing to computer software. The immediate gains of increased efficiency, speed of processing, rapid access to centralized data files, etc., are clear (or the investment would not be justified).

However, there may be a long term, possibly devastating hidden cost as the organization finds its ability to adapt and respond to new environmental conditions hampered by its inability to modify its information systems accordingly.

II. ANOTHER PROBLEM: TRANSPORTABILITY OF KNOWLEDGE

By 'application system' (or simply 'application') we refer to a computer system composed of various programs and data files which together perform some identifiable organizational task—e.g., sales order processing, inventory control, etc. Out attention is therefore to the software that deals directly with the organization's operations and not for instance operating systems etc., which service the internal operations of the computer.

Applications software of this sort is by and large custom made for each organization usually by an in-house data processing (DP) department. More importantly, these applications are typically written 'from scratch'. That is, they do not make use of previously developed program code pertinent to the problem domain.

The exception to this is the use of 'off the shelf' program packages and, occasionally pre-written subroutines which the new program can call at the appropriate point. For instance, numerous packages exist to do statistical analyses and quantitative algorithms and are used quite frequently in scientific applications. Likewise, off-the-shelf packages exist to do such organizational tasks as payroll processing, inventory control, etc. This latter class of pre-written software has, however, been less successful.

The problem, once again, has to do with the 'designed flexibility' of the package. In scientific applications, the contexts in which a particular analysis or algorithm is used is relatively well specified. For instance, in any application of a linear programming algorithm one must specify the objective function, constraints and technological co-efficients and one receives as a result, the values of the decision variables. For most organizational applications, however, the problems are less standardized. Probably the most regular of these is payroll processing, but even there considerable variations may exist from one firm to another as to benefits to be added, automatic deductions, classifications of labor, etc.

In order to make use of an off-the-shelf package for such applications, the particular characteristics of the organization's problem must fall within the designed flexibility of the package. When this does not occur the DP department may sometime try to modify the package. However, the general experience is that it is usually easier and more reliable to re-program the whole thing from scratch.

We call this aspect of application software development the problem of 'transportability of knowledge' from one application to another. As observed, this is generally an all or nothing proposition. One may transport chunks of knowledge from one system or program to another only in the case that the chunk corresponds to a whole program or subroutine. There seems to be no middle ground; that is, where one could make use of an arbitrary part of one program function in developing another.

The consequence of this is that software for organizational information processing is not a smooth evolution; it does not build naturally from previous experience. Thus, for example, after a quarter century of automated payroll processing, firms still often have to write new payroll programs.

By contrast, knowledge in the form of human expertise is easily transportable. For instance, when company X hires a new bookkeeper, it is doubtful X's accounting system exactly fits the bookkeeper's training or previous experiences. However, provided the new person is reasonably competent, he/she can adapt to the new system after a brief orientation period. The situation with applications software is as if a complete re-education, starting with grammar school, would be necessary.

We summarize the arguments thus far. The basic claim is that a fundamental problem exists in the basic architecture of applications systems, namely that they are too 'brittle' and resistant to change. This has two important consequences. One, as discussed in the last section, is that as an organization becomes increasingly reliant on its information system, it too becomes brittle and unable to adapt easily to new situations. The other consequence, the point of this section, applies not just to individual organizations, but to information system technology at large: current software architecture does not provide the proper framework for a smooth evolution of problem solving capability. We are forced to repeatedly re-invent wheels. Progress (what little can be seen) has always been in the form of someone's coming up with a bigger wheel. That this is wasteful of money and effort is the smaller part of the problem. The deeper difficulty is that when someone finds an improved method for some organizational task, these advances cannot easily be promulgated to other software for related tasks. The industry of applications software development thus cannot build on its accomplishments, and must continually re-start from the ground.

In the sections to follow, we examine the technical reasons why applications systems are so brittle. This has two closely related aspects: the first arising from the way program logic is structured; the second due to the ways data is organized in data files and data bases. An alternative architecture for applications software will be proposed that avoids these problems, albeit not without certain costs.

III. THE PROBLEM WITH PROGRAMS: PROCEDURAL LANGUAGES VS. PRODUCTION SYSTEMS

Statements in a programming language are in the form of *commands* to the machine — i.e., add this, move this data from here to there, print this on the terminal, etc.

A computer program is thus a *sequence* of such statements, e.g.,

```
10  LET X = 2
20  LET Y = 3
30  LET Z = X + Y
40  PRINT Z
```

Here, the statements have been numbered for identification purposes. Importantly, the ordering of the statements in this program indicates the sequence in which the commands are to be performed by the machine.

This otherwise linear sequence of execution can be modified by what are called 'control statements'. Consider, for instance, the program:

```
10  LET X = 0
20  ADD 1 TO X
30  PRINT X
40  IF X = 100 GO TO 60
50  GO TO 20
60  STOP
```

When executed, this program prints the numbers from 1 to 100. Here, statements 40 and 50 are control statements. In statement 40, if X has reached 100, program control jumps to statement 60 where it stops. Otherwise, statement 50 directs the program control back to statement 20 where X is again incremented, printed, etc.

Thus, the execution sequence in such computer programs normally follows the top to bottom ordering of the statements, except when superceded by the effects of control statements.

Computer languages of this type are called *procedural*. These are basically the only type used in commercial practice, and include all the well known languages for data processing and scientific applications—e.g., COBOL, FORTRAN, PL/I, BASIC, ALGOL, etc.

In these cases, the knowledge embodied in the computer program is expressed as the specific steps for doing it. A key thing to recognize is that this procedurality makes the statements of the program inter-dependent. Generally (though not always) changing the order of any two statements makes a serious change to the program's operation.

While it may not be patently obvious from the two tiny examples above, it is this inter-dependence that makes computer programs so difficult to modify.

As a result of an interesting blend of computer science and formal linguistics, an alternative approach has emerged over the last decade or so. This approach is based on so-called 'production systems' (PS's) which enable the knowledge of the program to be expressed in a form that is independent of its execution sequence.

The concept of production systems was first proposed by the linguist Post in 1943 to aid in the formal specification of natural language grammars. The basic idea is extremely simple. A single production is a rule of the form:

IF <pattern> THEN <action>,

or, in the more usual notation,

$$\langle\text{pattern}\rangle \rightarrow \langle\text{action}\rangle.$$

A production *system* consists of a 'data base' and a collection of such production rules. (This is a database in a fairly restricted sense, not to be confused with those maintained by database management systems.)

The pattern in each rule is some condition to be matched by the database and the action is typically some modification to the database. In the purest form of a production system, the rules are arranged in a linear order. Starting from the beginning the patterns are compared to the database until a successful match is found. The corresponding action is then performed and the process is then repeated, starting once again from the beginning comparing the patterns to the database.

Nilsson (1980:21) summarizes this as the following generalized procedure:

```
Procedure PRODUCTION
1.  Data ← initial database
2.  Until DATA satisfies the terminal condition, do:
3.  begin
4.  select some rule, R, in the set of rules that can be applied to
    DATA
5.  DATA ← result of applying R to DATA
6.  end
```

Consider for instance the following example for recognizing a certain type of English declarative sentence.

1	THE → DET	8	N → NP
2	ON → PREP	9	ADJ NP → NP
3	HUNGRY → ADJ	10	DET NP → NP
4	BIT → VT	11	PREP NP → PP
5	DOG → N	12	VT NP → VP
6	CAT → N	13	VP PP → VP
7	NECK → N	14	NP VP → S
		15	S → halt

The production rules on the left represent a lexicon indicating the grammatical categories of various words. The rules on the right indicate the grammar proper.

In formal grammars, a distinction is normally made between *terminal symbols*, i.e., the basic symbols in the language (English words in this case), as opposed to *non- terminal symbols* which indicate grammatical constructs. However, in a production system implementation of such a grammar, these are simply different elements of the database. When the database consists only of the symbol "S", the sentence is accepted as grammatical and the system halts.

For example, suppose we have the sentence:

"The hungry dog bit the cat on the neck."

the database transformations would be as follows.

THE	HUNGRY	DOG	BIT	THE	CAT	ON	THE	NECK	initial
DET	HUNGRY	DOG	BIT	DET	CAT	ON	DET	NECK	rule 1
DET	HUNGRY	DOG	BIT	DET	CAT	PREP	DET	NECK	rule 2
DET	ADJ	DOG	BIT	DET	CAT	PREP	DET	NECK	rule 3
DET	ADJ	DOG	VT	DET	CAT	PREP	DET	NECK	rule 4
DET	ADJ	N	VT	DET	CAT	PREP	DET	NECK	rule 5
DET	ADJ	N	VT	DET	N	PREP	DET	NECK	rule 6
DET	ADJ	N	VT	DET	N	PREP	DET	N	rule 7
DET	ADJ	NP	VT	DET	NP	PREP	DET	NP	rule 8
DET	NP		VT	DET	NP	PREP	DET	NP	rule 9
NP			VT	NP		PREP	NP		rule 10

NP	VT	NP	PP	rule 11
NP	VP		PP	rule 12
NP	VP			rule 13
S				rule 14
halt				rule 15

Note that the production system would have reached the same conclusion had the ordering of the rules been reversed. This could hardly be done in an ordinary computer program. On the other hand, with the rules reversed, the system would have been much less efficient since, for instance, the initial translation of terminal symbols would have needlessly searched through the higher level transformation rules.

The initial applications of production systems in computer science were in the area of compiler theory, i.e., in specifying the syntax and interpretation of programming languages (as opposed to natural languages). Subsequently, it has been recognized that PS's have a potential much broader range of usefulness. For instance, one classic application was the Logical Theorist of Newell, Shaw and Simon (1963). Beginning with the initial axioms and rules of inference of Russell and Whitehead's *Principa Mathematica*, the Logical Theorist successfully proved all the theorems of this massive text. Indeed, in several cases it found original proofs, simpler than the original.

Another famous example of the use of production systems was Shortliffe's MYCIN system (1976). The purpose of MYCIN is to perform medical diagnosis. In this case, the database is the patient's symptoms, as revealed by various laboratory tests, etc. The production rules (some

300 of them) are thus the sort of medical deductions a doctor might make based on these symptoms. For example:

IF	the infection type is primary-bacteremia, the suspected entry point is the gastronitestinal tract, and the site of the culture is one of the sterile sites,
THEN	there is evidence that the organism is bacteroides.

Within the area of Artificial Intelligence (AI) numerous other applications of production systems have been explored.

Davis and King (1975), is an excellent survey article on production systems. Commenting on the types of applications where PS's are best suited, they observe that

> where the emphasis of a task is on recognition of large numbers of distinct states, PS's provide an advantage. In a procedurally-oriented approach, it is both difficult to organize and troublesome to update the repeated checking of large numbers of state variables and the corresponding transfers of control....
>
> [PS's are] characterized by the principle that "any rule can fire at any time," which emphasizes the fact that at any point in the computation, any rule could possibly be the next to be selected, depending only on the state of the database at the end of the current cycle. Compare this to the normal situation in a procedurally oriented language, where such a principal is manifestly untrue: it is simply not the case that, depending on the contents of the database, any procedure in the entire program could potentially be the next to be invoked.
>
> PS's therefore appear to be useful where it is important to detect and deal with a large number of independent states, in a system which requires a broad scope of attention and the capability of reacting to small changes.

With regard to the ease of modification of PS's, they continue (p.20):

> We can regard the *modularity* of a program as the degree of separation of its functional units into isolatable pieces. A program is ***highly modular*** if any functional unit can be changed (added, deleted, or replaced) with no unanticipated change to other functional units. Thus program modularity is inversely

> related to the strength of coupling between its functional units.
>
> The modularity of programs written as pure production systems arises from the important fact that the next rule to be invoked is determined solely by the contents of the database, and no rule is ever called directly. Thus the addition (or deletion) of a rule does not require the modification of any other rule to provide for (delete) a call to it. We might demonstrate this by repeatedly removing rules from a PS: many systems will continue to display some sort of "reasonable" behavior, up to a point. By contrast, adding a procedure to an ALGOL-like program requires modification of other parts of the code to insure that it is invoked, while removing an arbitrary procedure from such a program will generally cripple it...
>
> Thus where the ALGOL programmer carefully chooses the order of procedure calls to create a selected sequence of environments, in a production system it is the environment which chooses the next rule for execution. And since a rule can only be chosen if its criteria of relevance have been met, the choice will continue to be a plausible one, and system behavior remain "reasonable," even as rules are successively deleted.

As described so far, pattern matching proceeds from the beginning of the rule set each time until a match is found, in which case that corresponding action is taken and the process is repeated.

However, in the notion of a 'pure' PS, each rule supposedly has an equal chance of firing — i.e., its position in the rule set should not affect its chances of firing. This only causes difficulty when the patterns of more than one rule match the database, in which case a choice must be made which action to take. A variety of approaches have been used to resolve such rule contention, for instance:

rule order	—	use the first matching rule.
data order	—	data elements are assigned priority: pick the rule whose match gives the highest priority.
generality order	—	use the most specific rule
recency order	—	use the most recently executed rule.

Recall that each rule is matched against the entire database and that two simultaneously activated rules may have matches on completely separate parts of the database. Clearly, rule contention is only problematic when the firing of one rule would disable the database match of the other candidate rule(s).

Thus, in the pure form of a PS, *all* of the rules should be tested against the database on each cycle, the subset of matching rules selected, and a choice made (by same criterion) which of those should be allowed to fire.

However, as the database and/or number of rules gets large, the system degrades for lack of efficiency. In consideration of this, a number of production system implementations have allowed some degree of control structure to creep back in. Thus, various strategies or 'heuristics' have been employed to increase the likelihood that, for certain contexts, the applicable rules will be found quickly and that the entire rule set need not be examined without danger of ignoring an applicable rule.

Thus, a number of PS implementations exhibit a greater or lesser degree of 'partial procedurality' as production systems augmented with a control structure mechanism. The design of such control structures, so as to provide efficient search without nullifying the advantages of flexibility offered by the basic PS orientation, has become a matter of intense interest and debate within computer science (see, e.g., Winograd 1975; Kowalski 1979a).

This is an interesting development for the context of this paper since it provides a framework for examining various styles of rule organization and management along a *continuum* of procedurality, instead of a flat choice between the two extremes.

A sign of the potential viability of production systems has been the rapidly increasing popularity of the language PROLOG.* Originally developed in the early 1970's by Colmerauer at Marseille, France. It has since been re-implemented and extended numerous times at universities and research institutes in France, England, Canada, Portugal, Hungary and elsewhere.

PROLOG is a 'backward inferencing' production system; i.e., PROLOG programs are written to deduce backwards from a specified goal to the available facts in the program's database. Partial procedurality may be introduced through a special device called a 'cut'. Thus, PROLOG programs may be written as purely declarative rules, without using the cut; but may be made increasingly procedural through extended uses of the cut operator.

Excellent texts on PRLOG are by Kowalski (1979b) and also Clocksin and Mellish (1981); a wide range of PROLOG applications and example programs are discussed in Coelho et al. (1980). A perceptive critique of the language for the American artificial intelligence community (which seems to be more committed to the language LISP), is given by McDermott (1980).

* The name PROLOG, standing for PROgramming in LOGic, is now more of a historical acronym due to the language's construction around the "resolution principle," a technique used in automatic theorem proving. While theorem proving remains as one of the application areas of PROLOG, its usage has since broadened considerably to include relational databases, natural language parsing, expert systems, etc.

IV. THE PROBLEM WITH DATA: DATA FILES VS. PREDICATE CALCULUS

Most application software used in organization centers around the processing of large amounts of data (as opposed to, for instance, optimization routines which are much more computation intensive on relatively small amounts of data). Hence, inflexibilities introduced by the way data is organized in data files and databases are equally (if not more) important than those introduced in the design of procedural programs. At any rate, as will be seen shortly, the problems are highly inter-related.

A note on terminology. In the last section, the term database was used to designate the data repository of a production system. In this section, the term database will be used more in the sense associated with database management (DM). Later we return to compare the two views, at which point they will be distinguished as PS databases and DM databases.

For the moment, however, we consider a *general* view of data maintained in data processing applications, whether this data is accessed through a database management system or not. The term 'data file' will therefore be used to indicate a conventional data processing file or a logical segment of a database (e.g., the tuples of a single relation in a relational database; the instances of a single record type in a CODASYL database). The term 'database' will then be used to refer to a collection of such data files with inter-related subject matter (e.g., sales file, inventory file, back-order file), whether or not the access to these is coordinated by a DBMS.

Data files are usually organized as a rectangular table with labeled columns called 'fields'. For instance, a file on employees might have fields for the employee's name, address, age, salary, etc.

EMPLOYEE FILE

Name	**Address**	**Age**	**Salary**
Adams	5 Pine Street	30	20,000
Peters	101 Broadway	45	18,000
Smith	3 Park Place	37	24,000

Sometimes data files have more complicated organizations – e.g., some columns may have multiple entries for a given data item. This tabular view is sufficient for the purposes here, however. Also, this is the basic view maintained by the more popular database management models (i.e., Network, Relational).

Note that each data file has three levels of description: the ***data file name*** (e.g., EMPLOYEE), the ***field names*** (e.g., NAME, AGE), and the ***data values*** (e.g., Smith, 37). It is important to note also that a data file represents a ***model*** of some aspect of the organization, in this case, what are considered to be the important features of employees.

The structure of the data file often carries certain implicit information as well. Often, as in this example, each row of the data file implies the existence of some entity in the environment, in this case an employee associated with the company. The converse assumption is also sometimes made, e.g., if a person's name does *not* appear in the file, then he/she is not an employee.

Other data files, however, might have different existence assumptions, for instance a file for parts inventories.

PART FILE

ID#	Color	WT	QTY
3	R	10	200
12	B	8	65
7	W	13	0

This file indicates the identification number (ID#), color, weight (WT) and quantity (QTY) on hand of various manufactured parts. In this case, each row of the file does not imply the existence of a part, but only elaborates the features of each generic part type. The existence of actual parts is instead indicated by the QTY field.

These might be called the existential assumptions associated with a file. Other assumptions refer to the possible data values that may appear in a given field, e.g., that SALARY must be less than 50,000.

The basic point, however, is that the data file structure itself is not sufficient to convey all these assumptions. Instead, these appear in the logic of the programs that interpret these data files. Thus, the model of the organization represented in the application system is found not only in the data files but also in the code of the various application programs.

This is a problem that has been recognized for some time in database management, and has led to a number of proposals for the separate specification of so called 'data base constraints', conditions that the data in the database must always fulfill. Such constraints are maintained in a separate table, and verified by each updating program. However, these approaches do not go far enough. There is a basic problem that remains, which has to do with the very notion of 'data' itself.

In *all* data processing files and database management systems, there is a distinction between *data structure* and the data itself. What we have called the datafile names and field names, are the data structure elements of the view presented here. (Other views of data may have further structural elements.) Thus, for instance, in the above data file for parts, we have in the first row: COLOR = "RED", where the three character string "RED" is the value of the field COLOR. The point is that these data values are regarded as *strings of characters rather than as properties of objects in the environment*. Viewed only as character strings, one is unable to specify even very commonplace inter-relationships between these properties; for instance, that if a thing has a color, it must be a physical object, hence, having weight, physical extension, geographical location, etc.

The basic problem is that the variables in data management models range over sets of *character strings* (so-called 'attribute domains' in the relational model), rather than over *objects* in the environment. For instance, a database constraint that all parts are either red, blue or white would look something like:

```
PART.COLOR = "RED" OR "BLUE" OR "WHITE"
```

To recognize that these are properties of objects in the environment, a predicate calculus notation might be used, introducing the variable x to range over these objects:

1. $\forall x$ PART (x) $\rightarrow$ RED (x) OR BLUE (x) OR WHITE (x)

(the symbol "$\forall$" is read "for all"). The point is that in this form, one can begin to elaborate more general properties, i.e., not just of parts, but of anything that has a color.

2. $\forall x$ RED (x) OR ORANGE (x) OR YELLOW (x) OR GREEN (x) OR ... OR BLACK (x) $\leftrightarrow$ COLORED (x)

3. $\forall x$ COLORED (x) $\rightarrow$ PHYSICAL-OBJECT (x)

4. $\forall x$ PHYSICAL-OBJECT (x) $\rightarrow$ $\exists n$ $n > 0$ & WEIGHT (x) = n.

(the symbol "$\exists$" is read "there exists").

Statement (2) is a disjunct of all color names used in the organization, indicated that any of these implies the general feature of being colored, and vice versa, that being colored implies one of these properties. Statement (3) says that anything that is colored is also a physical object (though some physical objects — e.g., glass, mirrors — may not be colored). Statement (4) says that for any physical object there exists some positive number that is its weight (presuming some unit of weight measure).

The direction intended by this example should begin to become clear. Reconsider the problem of transportability of knowledge discussed in section two. Clearly there are many commonplace connections between properties that any organization would agree upon — e.g., the simple physics of colors, weights, physical extent, etc. These rules will hold for any physical object, from peanuts to box cars. Other classes of properties might be restricted to a particular social system—e.g., the number of spouses an employee might have, whether dual nationalities are recognized. Other classes of properties pertain to specific industries within a given social system—e.g., the accounting practices for banks vs. those for educational institutions. Lastly, there are clearly those properties that are organization specific, such as the ranks of personnel or the parts it manufactures.

Ideally, the inter-relationship of properties at any one of these levels should only have to be developed once—e.g., commonplace physics by a national or world wide bureau of standards, accounting practices by an industry accounting board, etc. Then, the task of any particular organization in developing its application software would only be to specify the ***differences*** of its local practice from that of the standardized models.

The proposal here is, therefore, to offer a predicate calculus (PC) notation as a replacement for the usual data structure view with the claim that it provides a richer framework, capable of specifying the inter-dependence of properties of objects, not just structured organizations of character strings.

It should be mentioned that this is not necessarily a recommendation that facts about the environment actually be ***stored*** in this form—the underlying implementation might actually make use of a more conventional data management model—but rather that the top-most ***level*** or ***view*** of the database have the PC form.

It should also be mentioned that a predicate calculus notation is not the only candidate to meet the objectives of abstracting the relationships of general properties. The various graphical representations called 'semantic' or 'associative' networks also share this goal. However, the predicate calculus has had a longer history of development and study and, in our opinion, is a more robust representation. The predicate calculus is, however, only a *framework*, a meta-theory in which more detailed theories can be described.

It can, for instance, be used to describe theories of mathematics, in which case the variables would range over numbers, or to theories in chemistry, where the variables would range over the physical elements. Thus, the real work in pursuing this proposed direction would be to develop a predicate calculus specialized to the problems of administration.

A key problem here is to develop an appropriate *epistemology*, that is a definition with the basic classes of entities involved in administrative problems. At first blush these seem to be basically people and the physical objects that are manufactured, bought and sold. However the dynamism of business requires that time also be recognized in a fundamental way. Beyond that, the import role played by such contractual objects as receivables, leases, licenses, insurance policies, notes, bonds stocks, etc. requires careful analysis.

Explication of these concepts has been the objective in the ongoing development of the formal language CANDID, first development in Lee (1980), and later refined in Lee (1981).

V. COMBINING THE APPROACHES: PRODUCTION SYSTEMS AND PREDICATE CALCULUS

The point of the previous section was to recommend a predicate calculus notation as a richer form of data representation. In section three, a production system approach was suggested as a more flexible framework for specifying the potential *actions* of an application system. The final step in the proposal here is to combine these frameworks, i.e., to use the predicate calculus form of database as the database of the production system.

Actually, production systems acting upon predicate calculus databases have been in experimental use for some time within the computer science area of artificial intelligence (AI). (See e.g., Nilsson 1980, for further background information.)

Systems with this design are usually called 'theorem provers', in that the function of the production system is to seek prove some 'goal' theorem, based on a set of initial axioms in the database. The term 'theorem proving' is not, however, confined to simply proving mathematical theorems. As noted in the previous section, the predicate calculus may be used to represent a wide variety of subject domains beyond mathematics.

Whereas the purpose of the database is to describe facts and interrelationships of properties about the environment, the function of the production system in this context is to ***deduce*** new facts and relationships. Thus the production rules in this design amount to ***rules of inference*** for the predicate calculus; that is, they serve to derive new predicate calculus statements from the original ones. These inference rules are 'truth preserving': if the original statements are true, so too, will be the deduced ones.

The general predicate calculus framework provides a number of such rules of inference. These rules are 'analytic' in that they apply regardless of the subject domain. In an applied predicate calculus, where the subject domain is specified, additional 'synthetic' inference rules may apply, specifically to this domain.

The development of such an inference structure specific to the context of organizational administration is thus another task of the research direction proposed here.

In the development of CANDID*, for example, the formal description of the obligatory contexts involved in contracts has led to the inclusion of higher order (so called "intensional") operators. In the general case, such higher order logics encounter certain fundamental, proof theoretic difficulties. However, in the context of specific application domains, as CANDID assumes, these difficulties are avoidable by restricting the scope of inference.

VI. CONCLUDING REMARKS

The problems initially set forth were twofold: the difficulties involved in modifying applications software in response to organizational change; and the problem of 'transportability of knowledge', i.e., the difficulties of using parts of previously developed software in the development of new systems.

* currently being implemented using PROLOG.

The causes for this inflexibility in application systems were diagnosed as the procedurality of programs and the view of data as structures of character strings. In response to the problem of programs, a production system approach was suggested; in response to the problem of data structures, a predicate calculus formalism was proposed along with a final observation that the two frameworks can feasibly be combined.

In a short paper such as this, one is forced to omit certain details and perhaps over-simplify others. We have tried to argue that the application software architecture suggested here is a potentially feasible solution to the organizational problems identified. The major difficulties in this recommendation is the development of what might be called an 'epistemology of administration', reducing the concepts in this domain to a formally tractable system. This has been the objective in the development of CANDID, mentioned above.

REFERENCES

Clocksin, W.F. and C.S. Mellish. 1981. Programming in Prolog. Berlin, Heidelberg, New York: Springer-Verlag.

Coelho, E., J.C. Cotta and L.M. Pereira. 1980. How to Solve It with Prolog. Lisbon: Laboratorio Nacional De Engenharia Civil.

Davis, R., and J. King. 1975. An Overview of Production Systems. Stanford AI Lab Memo AIM-271, Stanford Computer Science Report. STAN-CS-75-524. Stanford, California.

DeMillo, R.A., R.J. Lipton, and A.J. Perlis. 1979. Social Processes and Proofs of Theorems and Programs. *Communications of the ACM*, 22(5):271-280.

Kowalski, R. 1979a. Algorithm = Logic + Control. *Communications of the ACM*, 22(7):424-436.

Kowalski, R. 1979b. *Logic for Problem Solving*. New York and Oxford: North Holland.

Lee, R.M. 1980. CANDID: A Logical Calculus for Describing Financial Contracts. Ph.D. dissertation, available as WP-80-06-02, Philadelphia, PA: Department of Decision Sciences, the Wharton School, University of Pennsylvania.

Lee, R.M. 1981. CANDID Description of Commercial and Financial Concepts: A Formal Semantics Approach to Knowledge Representation. WP-81-162. Laxenburg, Austria: International Institute for Applied

Systems Analysis.

McDermott, D. 1980. The Prolog Phenomenon. SIGART Newsletter, No. 72(July):16-20.

Newell, A., J. Shaw, and H. Simon. 1963. Empirical Explorations of the Logical Theory Machine. In *Computers and Thought*, E. Feigenbaum and J. Feldman (eds.), New York: McGraw-Hill, pp.109-113.

Nilsson, N.J. 1980. *Principles of Artificial Intelligence*, Palo Alto, CA: Tioga Publishing Co.

Shortliffe, E.H. 1976. *Computer- Based Medical Consultations: MYCIN*. New York: America Elsevier.

Winograd, Terry. 1975. "Frame Representations and the Declarative/ Procedural Controversy." in *Representation and Understanding*, D.G. Bobrow and A. Collins (eds.). New York: Academic Press, pp.185-210.

Winograd, T. 1979. Beyond Programming Languages. *Communications of the ACM*, 22(7):391-401.

Wulf, W.A. 1977. Some Thoughts on the Next Generation of Programming Languages. In *Perspectives on Computer Science*, A.K. Jones (ed.). New York: Academic Press.