1986

LOGIC BASED INFORMATION SYSTEM SPECIFICATION VERIFICATION

Waldo C. Kabat
University of Illinois at Chicago

Wojtek Kozaczynski
University of Illinois at Chicago

Vicki Lovegren
University of Illinois at Chicago

LOGIC BASED INFORMATION SYSTEM SPECIFICATION VERIFICATION

Waldo C. Kabat
Department of Electrical Engineering and Computer Science
University of Illinois at Chicago

Wojtek Kozaczynski
Department of Information and Decision Sciences
University of Illinois at Chicago

Vicki Lovegren
Department of Information and Decision Sciences
University of Illinois at Chicago

ABSTRACT

The purpose of this paper is to present the logic-based approach to the problem of automatic verification of the different specifications of an information system. The data flow analysis method and its basic product, data flow diagrams (DFDs), are used as an example. A traditional approach to automated DFD verification is illustrated. In this approach, DFDs are represented by database logical files, and verification rules are implemented as data manipulation procedures. Next described is the logic-based approach. First, the DFD verification problem is conceptualized. Then it is described in terms of logic, as implemented in Prolog. A comparison of the two approaches is made by looking at respective implementations of a particular DFD verification policy. Advantages of the logic-based approach are discussed, and its usefulness for the automatic verification of other system descriptions, like data dictionary or conceptual data models is pointed out.

SYSTEMS DEVELOPMENT METHODOLOGIES

There are a number of methodologies describing the process of information system development. Though terminology differs among the different methodologies, and packaging of activities into phases or stages is not uniformly recognized, most would agree that a systems development life cycle exists and has been shown to be effective when applied towards the development of relatively stable, transaction type, information systems.

Another methodology, which has been experiencing increased popularity in recent times, is known as prototyping. It can prove to be more effective when applied towards the development of loosely-defined or less structured information systems (Burns, 1985). Its general approach is one of constructing a series of systems, each providing more features for an "active" user to experience. Immediate feedback and sense of accomplishment provide motivation for the user to "play with" the system and thereby generate new requirements, which the analyst/developer will seek to implement in subsequent prototype versions.

The more traditional life cycle methodology, on the other hand, is founded in the completion of sequential phases. Each phase is finished before beginning of the next phase. Output documents from one phase are thus provided as input documents to the next. Most would agree that these phases can be generally classified as analysis, design, development, and implementation. Due to the linear or sequential nature of this process, errors introduced in one phase will be propagated to later phases with significant magnification. It is well known that correcting design errors in response to complaints that the system "does not meet user requirements" is extremely expensive. Thus, accuracy during the early stages of analysis and design is of utmost importance.

Motivated by the goal of improving accuracy during these early stages of development, a number of structured techniques have been proposed (Jackson, 1975; Myers, 1975; Stevens, 1974; Warnier, 1976). Use of these techniques enforces structure in the analysis/design phases so that errors are less likely to be evident and better systems result.

Data Flow Analysis Method

One such method designed to reduce errors, was proposed by Yourdon and Constantine (Yourdon, 1978) and later improved by others (DeMarco, 1978; McMenamin, 1984; Page-Jones, 1980), and is known as data flow analysis. It has been widely accepted by IS professionals and recognized by the Data Processing Management Association-Education Foundation, as part of the CIS model curriculum.

This method is based on functional system decomposition and stresses the flow and transformation of data in the system. Although system specification is partly described by verbal techniques, the methodology relies heavily upon diagramming or graphical techniques. The graphical tools and techniques which form the basis of this methodology are known as data flow diagrams, or DFDs. These techniques, collectively, provide an informal means of communication between systems developers and future system users, and serve as a formal descriptive language for analysis and design.

An analyst using this methodology for logical design of a new information system would produce a set of the following documents: a set of DFDs, the system dictionary, the system logical database structure description, and a set of system implementation criteria. In so doing, he would first produce the DFDs, and then build upon this foundation in constructing the other documents. (Actually, the dictionary components are more frequently developed in parallel with the DFDs, but it is the DFDs which drive the process of system description.) In order for these components of logical design to be accurate, they must necessarily be consistent with one another, let alone be "correct" in their own right.

NEED FOR AUTOMATED TOOLS TO ASSIST THE SYSTEMS ANALYST

An analyst who relies upon the data flow analysis methodology in the process of application design and development could greatly benefit from automated tools to help him with the task of producing the products mentioned above. As the analysis and/or design progresses from general to specific (or from high-level to low-level), the task of producing the design products becomes more and more tedious and susceptible to error. Tools which could free the analyst from this attention to detail could allow him to focus on higher level problems and issues.

There are several obvious functions of analysis/design which are natural targets of automation. A graphical interface between the analyst and the analysis/design product, could allow the analyst to enter appropriate commands or keystrokes, thus directing the system to draw the diagrams. Its function would be primarily one of providing an effective graphic environment. Another important function is one which performs some level of verification, both syntactic and semantic. An automated tool designed to achieve this level of functionality, would focus on assuring (or providing some level of assurance) that the products of the various analysis/design activities are consistent, complete and correct.

Significant progress has been made in providing the first capability mentioned, i.e. the graphical interface. There are already a number products which handle this task very effectively, supporting such diagramming techniques as data flow diagrams, structure charts, Warnier diagrams, E-R diagrams, Jackson diagrams, and others. These products also support, to some degree, the second function mentioned above, the verification capability. A major shortcoming, however, lies in the fact that none of them offers an open architecture. The system user cannot change to the system "preprogrammed" verification rules and/or even add his own rules. A great deal of subjectiveness exists in the system design activities, especially in the preliminary stages of the design process. Consequently, different analysts may want to customize an analysis/design supporting system to their individual work style. Unfortunately, none of the available products offers such capability. This is mainly due to the implementation environment they use.

The purpose of this paper is twofold. First, we want to present logical principles of the automated verification methods used in the design of a prototype of the **Systems Analysts' Apprentice**, a system which will support the analyst in all phases of information system development (and in the production of each of the analysis/design specifications) as reflected in the DFA methodology. For purposes of exposition, we have chosen to limit our discussion to that of data flow diagram verification. Our motivation for choosing this verification phase rests in the fact that DFDs are the first of the design products to be constructed and are fundamental to the construction of the others. In addition, DFDs enjoy wide use in other structured analysis and design methodologies. Secondly, we want to contrast two possible methods of implementing a verification algorithm. The traditional method relies upon a set of programs and a corresponding database. The database stores the system specification upon which the programs perform consistency verification. A second, logic-based method, uses logic to express both the verified system specification and the verification process.

DATA FLOW DIAGRAMS

One can pick up just about any text on systems analysis and design (Myers, 1975; Page-Jones, 1980; DeMarco, 1978) and find a description of DFDs. The notation may differ slightly, and the requirements may vary somewhat, but most will agree that the basic ideas are as follows. A DFD is a graphical device which is designed to show the flow of data through a system, and the intermediate processes which transform the data from one form to another. There are four basic graphical symbols used: arrows, bubbles, double lines, and rectangles. The arrows represent the flow of data within the system. The bubble represents a process of data transformation. The double line represents a system file (store), and the rectangle represents an external user or interface. A set of these symbols is called a diagram. (We shall henceforth refer to the set of objects represented by these symbols, together with the diagram itself, as primitives. When the context is evident, we may refer to the symbols themselves as primitives, e.g. we will be referring to the six primitives: bubble, arrow, file, user, external interface and diagram.)

In the example in Figure 1, note the presence of the six primitives just mentioned. The small number of symbols and the complementary verbal description make DFDs easy to understand, giving them the "communicability" feature so needed in the early development phases. It is their hierarchical nature, however, which gives them their expressive and structuring power.

In this hierarchical structure, the most global description of the system is represented in the form of the so-called context diagram. This represents the top level of the DFD hierarchy. In the context diagram, the entire system is represented by a single process (bubble). All external entities (rectangles), are connected to the system by data flows (arrows) into and out of the process (bubble). This context diagram, or rather the sole bubble on the context diagram, is subdivided, or "exploded" into the so-called 0 Diagram, which normally shows a relatively small number of main system processes. The example DFD is such a 0 Diagram. It has three processes, corresponding to the processes carried out by three departments, Sales, Chips, and Accounting. Each of these processes, in turn, can be exploded into subprocesses, and so on. This hierarchical process of top-down refinement continues until the bubbles at the lowest levels (leaf bubbles) represent processes which are simple enough to be implemented as units. (These are then described in the corresponding system dictionary.) The set of DFDs describing a typical medium-size system could contain hundreds of processes, dozens of files and exter-

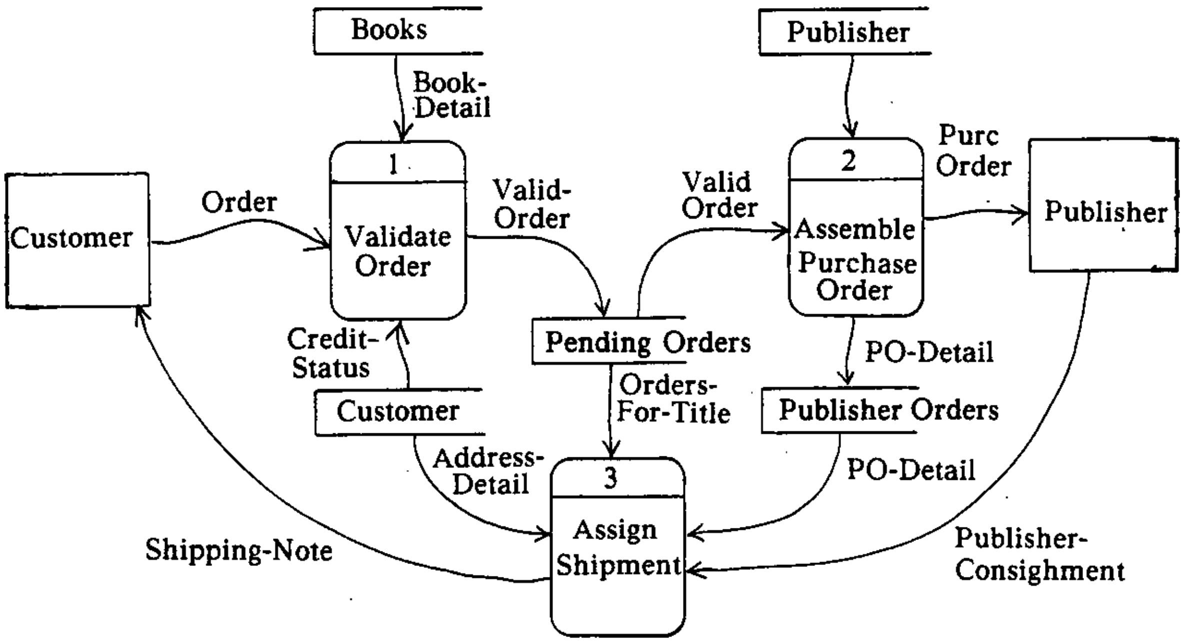

Figure 1.

nal entities and hundreds of data flows. (And before existing as a "final" set of diagrams, the set will have undergone extensive revisions from its initial definition. This is all in keeping with the characteristically iterative nature of the activities in the analysis/design phases.)

Verification of DFDs

It is easy to see how an analyst might get bogged down in the task of producing the DFDs alone, much less in the task of maintaining consistency of all design (analysis) products. Assuring that the DFDs are accurate is obviously a very tedious and time consuming task. It is believed by some that verifying a set of DFDs "correct" is impossible. They claim, and rightly so, that there are certain semantic errors which can elude capture. Nevertheless, there is clearly a significant class of "capturable" errors, and it is these which we hope to pursue. The task of verifying a set of DFDs can be accomplished by testing them against a set of rules. Satisfying each of the rules is necessary for the DFDs to be accurate (although, of course, not sufficient.)

DFD verification rules

What are these rules? By scouring the DFD literature, one can find some of them, but they are likely to be hidden in a relatively informal exposition about "how to construct" DFDs. Some are stated more emphatically than others, in phrases such as "ALL files MUST have at least one outgoing arrow and .... " while others are more casually mentioned as "conventions." While there simply is no universally recognized and formally stated set of DFD verification rules, there are enough of them which are widely accepted to include in a general DFD verification tool. (Any such tool must, understandably, be flexible enough to adapt to the methodology and conventions of the particular analyst who may be using such a tool.)

These recognized rules range from the simply stated "each bubble has a unique name" rule, to more complicated ones involving "balancing" of diagrams. This "balancing" rule can be stated as follows:

> The set INF_D of input data flows and the set $OUTF_D$ of the data flows contained in a diagram D are respectively equivalent to the sets INF_B and $OUTF_B$ of the bubble B, where bubble B has the same number as diagram D.

Clearly this rule represents a much more complex logical dependency between primitives than does the former "uniqueness" rule. It is convenient to classify the rules according to the complexity of this dependency between primitives. The uniqueness rule we classify as a global verification rule. Rules which are derived from more complicated dependencies are typically classified as intra-diagram rules or inter-diagram rules. Examples of these are, respectively, "an arrow cannot arise from and be directed to the same bubble," and, "if an arrow does not have a source or destination bubble, it may still be good, provided that it corresponds to an arrow at a higher level which does have both source and destination." (Source and destination are the obvious endpoints of the arrow.) Such an arrow, "anchored" at only one end on a particular diagram, is a byproduct of the conventions of the hierarchical DFD decomposition. We will be using this type of arrow in our illustrations and will henceforth refer to it as an "open-arrow."

The above discussion mentions one way of classifying verification rules, i.e. that the verification rule be one of the following types: global, intra-diagram, or inter-diagram. It is also convenient to classify the rules according to the primitive(s) to which they refer, i.e. bubble rules, arrow rules, etc. We say that any particular rule applies to a primitive category (or primitive subcategory, e.g. the "open-arrow" category.)

A DATABASE APPLICATION

A system to support an analyst in DFD verification could be developed in traditional technology and considered a database application. The primitives could be suitably represented, conceptually, as entities, and simple dependencies between the primitives could be thought of as relationships between these entities (Hawryszkiwycz, 1984). Simple DFD integrity rules could thus be expressed in terms of static database integrity constraints. For example we could allow only Bubble, File and External entities to be in the relationship Source-Of with Arrow entity. However, to enforce some of the more complicated integrity rules, e.g. those inter-diagram rules that must be satisfied to assure DFD consistency, an additional integrity checking program would be required. Thus, the traditional database application approach would require two components:

1. the database, as an implementation of the DFD conceptual model, and
2. programs manipulating and analyzing data to assure that the data form the correct DFD representation.

A distinct shortcoming of this approach, aside from the separation into two "related" parts, i.e. the data and programs, arises when any change is to be introduced. Consider the introduction of a new rule, perhaps one which one analysts uses, whereas another analyst does not. Or consider changing a rule to make it more or less restrictive. Or even introduce a new primitive subcategory. Changes such as these could result in program modification of unpredictable scope, or, in the extreme case, changes in the database structure as well.

Illustration of the Database Application Approach to DFD Verification.

Suppose we want to assure that all open-arrows are "good" according to the definition provided earlier. For this we need to provide both the data model to represent the related primitives and the data manipulation procedure to return a Yes or No answer to the question "is this open-arrow a good open-arrow?"

The database

We can conceptualize the primitives in the following way. Let the Diagram, Arrow and SmallElement be entity classes. Let SmallElement be subdivided into subclasses Bubble and FixedElement. Let FixedElement be further subclassified into File, User and ExternalInterface. Then we can define the following relationships between the primitives:

1. Source-Of; tertiary relationship between Arrow, Diagram and SmallElement assigning a SmallElement, SE, to an arrow A on the diagram D as the source of the arrow;

2. Destination-Of; tertiary relationship between Arrow, Diagram and SmallElement; analogous to the above Source-Of relationship, but describing the destination of an arrow;

3. Contained-In; binary relationship between Diagram and Bubble that assigns bubble A to diagram D.

Source-Of, and Destination-Of connect an arrow to its source and destination on a particular diagram. Any SmallElement can be a source or destination, except in certain situations. For example, on the context diagram all arrows must be anchored between the sole Bubble (with number 0) and a User or External-Interface, thus limiting the role of the SmallElements. Due to space limitations, we will not go into details of the conceptual model of the DFDs such as entity/relationship membership class or cardinality. Let us, however, for purposes of illustration, choose one type of data model, say relational, and one corresponding DML, say SQL, and construct the two components necessary to implement the good open-arrow rule.

A relational implementation

Using well known rules of conceptual model translation and subsequent logical model flexing (Briand, 1985; Howe, 1983), we can arrive at the following relational database design.

FE (NAME, TYPE, D_NO)

BUB (NAME, B_NO, D_NO)

SOURCE (A_NAME, NAME, TYPE, D_NO)

DEST (A_NAME, NAME, TYPE, D_NO)

The relations defined above correspond to: FixedElement, Bubble, Source of an arrow, and Destination of an arrow, respectively.

The ATTRIBUTE domains are as follows:

TYPE - one of {"Bubble", "External", "File", "Open"}

NAME - any string of characters, representing Bubbles or FixedElements - these names are unique as specified later (p.15)

A_NAME - any string of characters, representing Arrow names

B_NO - any string of single digit numbers, representing bubble numbers (with implicity decimal points between the single digit numbers).[1]

D_NO - any string of single digit numbers, representing diagram numbers (as above)

The following pseudocode represents one way of verifying whether a given open-arrow is indeed a "good" open-arrow. We assume that the procedure GOOD OPEN ARROW receives two parameters: D (diagram number) and A (arrow name), identifying an "open-arrow" on diagram D. The procedure sets a third parameter, ANSWER, to value "Yes" if arrow A on diagram D is a good open-arrow and "No" otherwise.

The procedure shown on the next pages makes explicit assumptions about routine enforcement of certain "integrity" rules, and thus frees itself of the responsibility for assuring them. Such integrity rules include the following:

1. All NAMEs of FEs are unique, and distinct from the unique NAMEs of Bubbles. Bubble Numbers (B_NOs) are unique and are alternate keys of the BUB relation. Diagram Numbers (D_NOs) are inherited from a Bubble in a "parent" Diagram. In any BUB tuple, the B_NO must be a one-level "extension" of the D_NO.

2. For each A_NAME/D_NO key value in the SOURCE relation, there is a corresponding A_NAME/D_NO key value in the DEST relation.

3. For every NAME value in SOURCE (and thus DEST), there is a tuple in FE or BUB with a matching NAME value.

[1]In the database implementation, the representation of diagram and bubble numbers is difficult. The commonly used convention is for all bubbles on diagram D to have numbers composed of the diagram number D, extended by a period and bubble number relative to the diagram (e.g., bubble 2.3.4 is the 4th bubble on the diagram 2.3). This convention makes it necessary to store the bubble and diagram numbers as variable length strings. Thus, in order to answer a simple question about whether or not a bubble is an ancestor of a certain diagram, a very low level string manipulation procedure would be required.

Good Open Arrow Verification Segment

```
beginsegment GOOD OPEN ARROW (D,A,ANSWER)
     select TYPE into T from SOURCE
          where A_NAME = A and D_NO = D
     if T = "open" then
          do ANCHOR_DEST
     else
          do ANCHOR_SOURCE
     endif
endsegment

beginsegment ANCHOR_SOURCE
     ANSWER = "?"
     dowhile ANSWER = "?"
          OLDDIAG = D
          select D_NO into D from BUB where B_NO = D
          select B_NO into NUM from BUB where NAME in
               (select NAME from SOURCE where A_NAME = A and
                D_NO = D)
          if not (DBFOUND) or not (NUM = OLDDIAG) then
               ANSWER = "No"
          else
               select Type into T from DEST where A_NAME = A
                  and D_NO = D
               if D = null then
                    if T = "External" then
                         ANSWER = "Yes"
                    else
                         ANSWER = "No"
                    endif
               else
                    if not (T = "Open") then
                         ANSWER = "Yes"
                    endif
               endif
          endif
     enddo
endsegment

beginsegment ANCHOR_DEST
     ANSWER = "?"
     dowhile ANSWER = "?"
          OLDDIAG = D
          select D_NO into D from BUB where B_NO = D
          select B_NO into NUM from BUB where NAME in
               (select NAME from DEST where A_NAME = A and
                  D_NO = D)
          if not (DBFOUND) or not (NUM = OLDDIAG) then
               ANSWER = "No"
          else
               select Type into T from SOURCE where A_NAME =
```

```
A and D_NO=D
                if D = null then
                    if T = "External" then
                        ANSWER = "yes"
                else
                        ANSWER = "No"
                    endif
                else
                    if not (T = "Open") then
                        ANSWER = "Yes"
                    endif
                endif
            endif
        enddo
    endsegment
```

One can see that the procedure presented above is not easy to follow. It requires some training in program development and database concepts. It is also critically dependent upon the structure of the underlying database. The most important point, however, is that it is very difficult to read the procedure without knowing *a priori* its intended purpose. For this reason, we provide the following summary of purpose of the GOOD OPEN ARROW procedure.

The procedure, upon learning that a certain arrow is "open," first determines which of the ends is open and which is anchored. It then proceeds, in an iterative fashion, to determine on which upper level (diagram) the arrow finally is anchored on both ends. While so doing, it checks to see that there are, indeed, corresponding arrows on the respective levels, and that the bubbles corresponding to the anchored end are consistent with the original anchored bubble. If there is no level at which the arrow becomes fully anchored, then the procedure will reject the original arrow. If, in addition, the anchored level is the context level, the procedure will guarantee that the entity corresponding to the open end is none other than an external entity.

It is also important to point out that any change in the good open-arrow policy may necessitate significant changes in the corresponding rule verification procedure.[2] Verification measures for this good open-arrow policy would involve the checking of corresponding data dictionaries, and thus is beyond the scope of this paper.) Because of this apparent resistance to change, or at least resistance to easy change, this traditional database approach does not seem well suited for such a product as a DFD verification tool. As pointed out earlier, such a tool must necessarily be flexible enough to support the methodology with which an analyst is most comfortable. The analyst, equipped with such a tool should be able to easily adapt the tool to his purposes, i.e. add/delete/change verification rules to his liking.

A LOGIC-BASED APPROACH

Another approach towards automating the DFD verification process uses logic as a means of describing the problem (Kowalski, 1979). With this approach, the same language can be used to represent the data (simple facts) as well as the data dependencies (in the form of rules or predicates). As a result, the problem of verifying DFDs becomes, in essence, a problem of proving or disproving logical statements (rules) related to the correctness of such diagrams. Given a set of DFDs, if all of the rules hold for that set, then the diagrams are correct (relative to the underlying correctness rules.)

Observing that there is nothing in this approach which specifically ties it to DFD verification in particular, a strong case can be made for using this approach with a general class of verifica-

[2] A typical change might involve the weakening of this restriction somewhat to allow for the separation of arrows into several arrows in the process of moving down in the DFD hierarchy (or equivalently, the merging of arrows in the process of moving up

tion problems. Of particular interest to us, however, is its use in any type of system **specification verification.** This rule-based (logic-based) approach to DFD verification has other advantages as well:

1. The verification algorithm follows expert rules (mimics an expert), therefore the verification process is represented (described) in the most natural and understandable way;

2. The verification algorithm can be easily changed by incorporating, dropping, or changing rules, unlike the traditional database application approach;

3. The representation of the facts can be changed without changing the verification rules (as opposed to the traditional database application approach, where changes in the database structure would likely necessitate subsequent changes in the programs).

Our choice of languages in which to implement our logic-based DFD verification module is Prolog. (We are currently using Arity Prolog on microcomputers and Waterloo Prolog on the IBM3081. Both are consistent with the Edinburgh style syntax (Clocksin, 1984).)

Prolog

Prolog is a logic programming language allowing programs to be written which describe a particular application domain. The Prolog interpreter, which executes the program, makes most of the control decisions. The Prolog programmer's responsibility is one of providing the axioms (rules and facts) which describe objects and the relationships between the objects. These axioms are expressed in a logic language known as Horn Clauses (Kowalski, 1979). (Axioms expressed in this form can be subjected to certain inference mechanisms, known as resolution (Robinson, 1965) and unification, thus enabling the provability of posed queries about the application domain.) The set of axioms comprise the knowledge base (possibly referred to later as the Prolog database) and take the form of routine facts about objects and more general rules describing relationships between objects.

Horn Clauses take the form:

```
" conclusion:-
        condition1,
        condition2,
         ...
        conditionN.   "
```

Here,

":-" means "if"
"," means "and"

Thus, the whole clause means "conclusion is true if condition1 and condition2 and ... conditionN are all true." (The conclusion is known as the head of the clause.)

DFD verification in Prolog

The Prolog knowledge base for our DFD verification application could be roughly divided into two parts: 1) the part specifying the rules related to DFD verification (accuracy), and 2) the part describing the set of DFDs themselves, i.e. the specification of DFD primitives.

The latter part of the knowledge base would consist, primarily, of simple statements (for example "Bubble 2.4, known as 'order-entry' is a bubble on Diagram 2). Expressed in Horn Clause syntax, this statement could be expressed as follows: bubble(2,order-entry,2.4). (This is, in fact, the syntax we've adopted for defining the bubble primitive.) Note that this particular clause takes the form of a conclusion without any conditions. This type of clause is commonly known as a fact. The DFD specification (of primitives) would thus be comprised almost exclusively of facts.

The part of the knowledge base which specifies the conditions for accuracy would be comprised of more complex Horn Clauses, those with conditions commonly known as rules. For example, one rule giving some conditions for an open arrow to be "ok" is (in natural language):

arrow A is an **ok_open arrow** on the diagram D
if

A is on diagram D **and**
T is the type of the Source of the arrow
A **and**
T is a bubble **and**
A can be proven to have a **good_open_source** on one of the ancestors of the diagram D.

Expressed in Horn Clause form, this becomes the following rule (see Rule 3 in the illustration of the logic-based approach appearing later):

```
ok_open(A,D) :-
    arrow(D,_,A,Source,_),
    elttype(Source,T),
    T = bubble,
    good_open_source(A,D).
```

Other rules may be more complex, having many conditions, each of which is the head of another rule. There are, in fact, many generic or utility rules which are the building blocks of the more complex rules which describe the essence of DFD verification.

The process of verifying a posed query against the knowledge base is taken care of by the Prolog interpreter, fulfilling its responsibility of providing the control (decision, inference) mechanism. The method by which it does this, depending upon the particular implementation, relies on variations of techniques known as resolution and unification. (It is not important to know precisely what these techniques are - see Kowalski (1979) and Robinson (1965) - but it is important to realize that the control decisions are taken care of by the Prolog interpreter serving as a general inference engine.)

The knowledge base, consisting of the two parts mentioned above, are defined by the appropriate Horn Clauses. Certain clauses are provided as part of the DFD verifier "core," whereas others, in particular those describing the DFD being analyzed (DFD primitives), are to be defined by the analyst. The concise and simple form of the DFD knowledge base permits relatively easy addition or modification of rules and/or facts as might be required by a particular analysis team. After the Prolog knowledge base is defined, a simple command (query) is all that is needed to initiate the verification procedure. The analyst can request global verification, i.e. "Do all primitives satisfy the set of verification rules currently in the knowledge base?" or he may request verification of a particular rule, i.e. "Are all the arrows good arrows?" or "Is arrow 'line-item' a good arrow?"

Conceptualization of the DFD Verification Problem in a Logic-Based Format

Conceptualization Of primitives

We conceptualize the DFDs as having six primitives, as before. They are, again, the following: bubble (process), arrow (data flow), file, user, external interface and diagram. Though the literature often does not differentiate between users and external interface, calling them both external entities, many experts make this distinction and have rules which apply to one and not to the other. The diagram primitive is necessary to establish the domain of the problem.

We found it convenient to define superclasses or generalizations of primitives as follows:

External = user or external interface

Fixedelt = file or External

Smallelt = Fixedelt or bubble

Element = Smallelt or diagram

Symbol = arrow or Element

These are useful for simplifying system rules which apply to groups of primitives.

It was also convenient for us to define terms representing subcategories of the basic primitives. For example, we defined the following subcategories of the primitive category "arrow":

Singlearrow - an arrow going in one direction only

Doublearrow - an arrow going in two directions

Bubblearrow - an arrow whose source and destination are bubbles

Openarrow - an arrow with either source or destination not defined, or "open"

Filearrow - an arrow going from bubble to file or vice versa

Extarrow - an arrow going from bubble to External or vice versa

Fixedeltarrow - Filearrow or Extarrow

These kinds of subcategories are also useful for simplifying verification rules and for providing simple edits (e.g. an arrow which cannot be classified according to these subcategories is illegal). Note that they are not necessarily mutually exclusive.

Illustration of the Logic-Based Approach to DFD Verification

The following subset of the Prolog database is sufficient to verify the "good open-arrow" rule that has been used as an illustration.

```
/*
        Utility pedicates
                                                    */

length([],0).
length([_|x},M) :-
      length(X,N) ,M is N+1.

expansion(U,[]).
expansion([X|V},{X|V} :-
      expansion(U,V).

append([],L,L).
append([KL1],L2,[K|L3]) :-
      append(L1,L2,L3).

upperlevel(Y,X) :-
      append(Y,V,X),length(V,1).

lowerlevel(Y,X) :-
      append(X,V,Y),length(V,1).

/*
DFDs are represented by the following facts:
                                                    */
```

1. **arrow(Diagram,Shape,Name,Source, Destination)**
2. **bubble(Diagram,Name,Number)**
3. **file(Diagram,File)**
4. **user(User)**
5. **interface(Interface)**
6. **diagram(Diagram)**

```
/*
        Definitions
                                                    */

diagramname(P) :- diagram(X),bubble(_,P,X).

external(P) :- user(P).
external(P) :- interface(P).

fixedelt(P) :- file(_,P).
fixedelt(P) :- external(P).
smallelt(P) :- fixedelt(P).
smallelt(P) :- bubble(_.P,_).
element(P) :- smallelt(
element(P) :- diagramname(P).
symbol(P) :- element(P).
symbol(P) :-arrow(_,_,P,_,_).

elttype(X,arrow) :- arrow(_,_,X,_,_).
elttype(X,bubble) :- bubble)_,X,_).
elttype(X,user) :- user(X).
elttype(X,interface) :- interface(X).
elttype(X,file) :- file (_,X).
elttype(X,interface) :- interface(X).

/*
        Exemplary diagram rules
                                                    */

parent(F,_) :-
      diagram(F),
      f = [context],
      !,fail.
parent(F,G) :-
      diagram(F),
      diagram(G),
      upperlevel(F,G).

ancestor(B,D) :-
      diagram(D),
      bubble(_,B,N),
      (expansion(N,D);N = [0]).

/*
        Exemplary arrow rules
                                                    */

openarrow(X,D) :- arrow(D,_,X,open).
openarrow(X,D) :- arrow(D,_,X,_,open).

/*
```

the following predicate "good_open_arrow" succeeds only if a source/destination of an open arrow A on diagram D can be traced up to a correct source/destination of the same

arrow connected to the ancestor bubble of the diagram D.

***/**

```
/* 1 */ good_open_arrow(A,D) :-
        openarrow(A,D).
        D = [context],
        !,fail.

/* 2 */ good_open_arrow(A,D) :-
        openarrow(A,D).
        ok_open(A,D).

/* 3 */ ok_open(A,D) :-
        arrow(D,_,A,Source,_),
        elttype(Source,T),
        T = bubble,
        good_open_source(A,D).

/* 4 */ ok_open(A,D) :-
        arrow(D,_,A,_,Destn),
        elttype(Destn,T),
        T = bubble,
        good_open_destn(A,D).

/* 5 */ good_open_source(A,D1) :-
        parent(D2,D1),
        arrow(D2,_,A,X,_),
        D2 = [context],
        !,
        external(X).

/* 6 */ good_open_source(A,D1) :-
        parent(D2,D1),
        arrow(D2,_,A,X,_),
        elttype(X,Z),
        (Z = bubble; Z = file).

/* 7 */ good_open_source(A,D1) :-
        parent(D2,D1),
        arrow(D2,_,A,open,Y),
        ancestor(Y,D1).
        good_open_source(A,D2).

/* 8 */ good_open_destn(A,D1) :-
        parent(D2,D1),
        arrow(D2,_,A,_,Y),
        D2 = [context],
        !,
        external(Y).

/* 9 */ good_open_destn(A,D1) :-
        parent(D2,D1),
        arrow(D2,_,A,_,Y),
        elttype(Y,Z),
        (Z = bubble; Z = file).

/* 10 */ good_open_destn(A,D1) :-
        parent(D2,D1),
        arrow(D2,_,A,X,open),
        ancestor(X,D1).
        good_open_destn(A,D2).
```

It is worth noting that the rules presented above contain the simple integrity-type rules which were assumed in the traditional database example. Had they been incorporated into the verification procedure, as they are here, the pseudocode would have been significantly larger.

Several differences between the two presented approaches are very apparent. First of all, the Prolog text is self-documented, i.e. one doesn't need to be a Prolog programmer to read it. Secondly, because both data and operations on data (rules) are expressed in the same language, the intent of the program is evident in the program structure itself. A simple translation of the analyst-supplied verification rules into Horn Clauses (Kowalski, 1979), and then subsequent translation into Prolog predicates is all that is required.

Look back at the example as we attempt to illustrate the process of rule verification. Utility predicates, definitions and exemplary diagram rules have been shown to make the example complete, and thus stand independently. The "good-open-arrow" rule that we use in the example, is found in the paragraph: exemplary arrow rules. For easy reference, we numbered this rule and its subordinate rules.

Rule 1 is a translation of the statement "if there is an open-arrow on the Context diagram, it cannot be a good open-arrow." Rule 2 "catches" all open-arrows on lower diagrams and uses rules 3 and 4 to determine whether the destination of the arrow or the source of the arrow is open. These rules also state that, for any open-arrow on a diagram, bubbles and only bubbles can anchor the arrow. (This was assumed to be true in the traditional database example.) Rules 5,6 and 7 are triplets which deal with the case when an open-arrow has an open source, while 8, 9, and 10 deal with the case when an open-arrow has an open destination. In the former case, the destination must be a bubble (this is verified in the primitive subcategory rules.) Rule 5 states that "if an open source is traced all the way up to the Context diagram, (before being finally anchored), the anchor must be an External entity." Rule 6, on the other hand, says that "if the source has been anchored on the diagram strictly below the Context diagram, that anchoring entity must be a file or bubble. Rule 7 recursively calls "good-open-source" in the case when an open-arrow is still open on the "father" diagram. This rule also determines whether ancestors (bubbles) of the original destination (bubble) are consistent with the original destination. Rules 8, 9, and 10 perform equivalent tests for the case when the open side of the arrow is the destination and the bubble side is the source.

SUMMARY

Automated tools to aid in the tedious process of information specification are sorely needed. Analysts need not spend excessive amounts of their valuable time checking and verifying low-level analysis/design documents. Construction of data flow diagrams is one of the tasks which is prime for this sort of tool. A number of automated tools exist for supporting the graphical aspects of this activity. Some support, in addition, the verification function. However, none provide the open architecture so important to the analyst for tailoring the tool to his design style. We illustrate an open architecture and logic-based approach towards DFD verification which can be used for system specification in general. Under current development is a prototype multi-purpose analysis/design product, called the **Systems Analysts' Apprentice**, which is based upon this logic-based approach.

In this paper we compare this logic-based approach to a more traditional database approach using a typical example of DFD policy. This example shows what is required, in both cases, to verify that a DFD primitive subcategory is valid. It is easy to see that the logic-based approach is the more natural approach. It relies upon a single representation of the data and the logical dependencies between the data, whereas the traditional database approach requires two separate components. The logic-based approach emphasizes the problem, not the solution/implementation of it. Furthermore, it supports an implementation which is more flexible and expandable than an implementation of the other. Although there are certain advantages of the traditional approach, i.e. higher performance and lower sensitivity to problem size, we feel that the simplicity/flexibility factors of the logic-based approach are significant enough to outweigh these advantages. (The number of rules required is not excessive enough to make the product ineffectual; SAA currently performs well on the IBM 3081 mainframe. We get satisfactory results, as well, when using the Arity Prolog Compiler on a microcomputer.) We also feel that efficiency is not the major factor that it is in transaction-type environments. Analysts would probably prefer a more flexible tool than a rigid one that runs in less time.

Our current version of the DFD verifier, a part of the prototype **Systems Analysts' Apprentice** (eventually to become a complete system specification tool), has a knowledge base containing over 30 major rules (on the order of complexity of the "open-arrow" rule illustrated). The numbers, as classified by primitive category are: Arrow rules (17), Bubble rules (7), File rules (2), External rules (2), Diagram rules (2), and Global rules/uniqueness rules (3). The natural extension of this DFD verification capability is the data dictionary verification capability, on which we are currently working. The DD module is based upon the same logic-based methods and is designed to be complimentary to the DFD module. As such, it provides the capability of detecting many more semantic-type errors. The next stage towards completion of a full analysis/design verification product will involve the building of the verification module for the conceptual data model (Briand, 1985). It, too, will be based upon the logic-based approach.

A market version product like the SAA should be suitable for use in traditional life cycle information system development environments, but should also be used in prototyping environments as it is presented, for example, in Bjornerstedt (1983), with respect to relational databases. We feel that its contribution could be even more significant in the latter environment, where application generation techniques are finding wide usage.

REFERENCES

Bjornerstedt, A. and Hulten, C. "RED1-A Database Design Tool for Relational Model," SYSLAB Report No. 18, University of Stockholm, Sweden, 1983.

Briand, H., Habrias, H., Hue, J. and Simon, Y. "Expert System for Translating an E-R Diagram Into Database," in *Proceedings of Fourth International Conference on Entity-Relational Approach,* Chicago, Illinois, October 1985.

Burns, R. and Dennis, A. "Selecting The Appropriate Application Development Methodology," DATA BASE, Volume 17, Number 1, Fall 1985.

Clocksin, W. and Melish, C. *Programming in Prolog,* 2nd Ed., Springer-Verlag, Berlin, Germany, 1984.

DeMarco, T. *Structured Analysis And System Specification,* Yourdon Press, New York, New York, 1978.

Deyi, L. *A PROLOG Data Dase,* Research Studies Press Ltd; John Wiley & Sons Inc., New York, New York, 1984.

Hawryszkewycz, I. *Database Analysis and Design,* Science Research Associates, Inc., Chicago, 1984.

Howe, D. *Data Analysis for Data Base Design,* Edward Arnold Publishers Ltd, London, England, 1983.

Jackson, M. *Principles of Program Design,* Academic Press, London, England, 1975.

Kowalski, R. *Logic for Problem Solving,* The Computer Science Library, North-Holland, New York, New York, 1979.

McMenamin, S. and Palmer, J. *Essential Systems Analysis,* Yourdon Press, New York, New York, 1984.

Myers, G. *Reliable Software Through Composite Design,* Petrocelli/Charter, New York, New York, 1975.

Page-Jones, M. *The Practical Guide To Structured Systems Design,* Yourdon Press, New York, New York, 1980.

Robinson, J. "A Machine Oriented Logic Based on the Resolution Principle," *Communications of the ACM,* Volume 12, 1965.

Stevens, W., Myers, G. and Constantine, L. "Structured Design," *IBM System Journal,* Volume 13, Number 2, 1974.

Warnier, J. *Logical Construction of Programs,* Van Nostrand Reinhold, New York, New York, 1976.

Yourdon, E. and Constantine L. *Structured Design,* Yourdon Press, New York, New York, 1978.