STRUCTURED ANALYSIS REPRESENTATIONS AS PRODUCTION SYSTEMS: AN INTERPRETATION AND ITS IMPLICATIONS

by

Vasant Dhar

and

Barry D. Floyd
Leonard N. Stern School of Business
New York University
624 Tisch Hall, 40 West 4th St.
New York, NY 10003

March 1989

Center for Research on Information Systems
Information Systems Department
Leonard N. Stern School of Business
New York University

Working Paper Series

CRIS #202
GBA #89-21

Table of Contents

Abstract

Much of transaction processing involves *classification*, that is, the categorization of inputs into outputs based on various tests. In Artificial Intelligence (AI), classification systems are generally represented in terms of AND/OR graphs. Such graphs are collections of production rules that capture declaratively the logic of an application domain. If one views a transaction processing system as a classification system, it becomes natural to represent it in terms of an AND/OR graph. In this paper, we present an interpretation of dataflow diagrams used in Structured Analysis as AND/OR graphs. By examining the dataflow diagrams, production rules capturing application-specific knowledge can be constructed. This interpretation has two implications: 1) production rules can be used to unify analysis and design since the same data structure (the rule) is used for both purposes, and 2) the resulting design can be simulated for purposes of explanation and what-if analysis. We also discuss some of the general pros and cons of production systems as they pertain to systems analysis and design.

1. Introduction

There are many analysis tools that can be used to specify the functional requirements for computer-based information systems. The specification primitives provided by these tools are meant to enable the designer to cope with the complexity of the problem and to provide users with an accurate and comprehensive view of the system as conceptualized by the designer. Further, these primitives should enable a smooth translation into high-level and detailed design of the system and continue into the coding phase. For a survey of analysis methods see deMarco (1978), Gane and Sarson (1979), and McMennamin and Palmer (1984). Various design tools have been described by Borgida et.al (1985), Alford (1985), Ross (1985) and Pressman (1987).

The primitives provided by the various analysis techniques have been shaped primarily by the type of problem being modeled. To model transaction processing systems for example, analysts often focus on the processes which occur to transform the inputs into outputs. In particular, structured methods, such as data flow diagrams, use a representation whose primitives are data flows, transformations, data stores, and sources/sinks. On the other hand, in order to model judgemental reasoning or decision-making oriented tasks, an analyst typically casts the problem in terms of very different primitives such as production rules and/or structured object representations designed to capture the types of knowledge involved in these tasks. Such primitives are provided by a number of expert system building shells.

There is no clear set of guidelines that indicates when an analysis method is most appropriate. In fact, there has been considerable activity involving the application of expert system shells (typically rule-based) for prototyping transaction processing systems. In discussions with practitioners, several have provided anecdotal evidence suggesting that use of such shells accelerates and simplifies the development of prototypes. However, there has been no analysis thus far of the real or potential advantages of doing this, and of the relationships between traditional analysis and design methods and expert system shells.

In this paper, our intent is to show that the dataflow diagram (DFD) representation used in structured analysis can be viewed as a special type of production system. Production systems have been used extensively in AI to model problems involving synthesis and classification. A classification problem is one where inputs must be combined and tested to yield various outputs, where all inputs and outputs and their structures are defined a priori. In contrast, synthesis problems involve the generation of outputs that might not have been envisioned a priori, such as in design and planning.

Transaction processing systems model problems that have well defined inputs, outputs, and relationships among them. Such problems are essentially classification problems, representable in terms of production rules that define and AND/OR graph. In addition to showing the declarative logic of an application, these graphs can also be interpreted as dataflow diagrams. This observation has two interesting implications. First production systems can be used to unify analysis and design since the same representational formalism (rules) is used for both. This reduces the arbitrariness at the analysis/design boundary inherent to SDMs, that is, the arbitrariness involved in translating the information contained in the analysis to the design phase and in translating the detailed design to code. Secondly, a design specified in terms of productions is "runnable" at all levels. The specification of the abstract design in terms of productions can be run for purposes of explanation (i.e. **why** are certain data being processed, **how** is a certain output achieved, or **what** set of outputs will be produced from a given set of inputs), and the specification of the detailed design in terms of productions (which corresponds to the mini-spec in structured analysis) provides the functionality of the system.

The remainder of this paper is organized as follows. In the next section we describe the basic primitives that define a production system. Readers familiar with the production system architecture may skip this section. In section 3 we introduce a typical type of transaction processing system, namely, an accounts receivable system, and specify it both in terms of a data flow diagram and in terms of a production system. We also describe the relationship between the two representations. In section four we enumerate a set of criteria that are typically used in evaluating specification languages and point out the considerations involved in using production rules for analysis and design. We conclude with implications for practitioners and for designers of analysis and design tools.

2. Production Systems

In contrast to traditional programs that use sequenced instructions as the basic unit of computation, production systems are characterized by data-sensitive rules called *production rules* or simply, *productions*. A production system architecture involves three components:

1. **Working memory**, an evolving global database of symbols. This database typically consists of *working memory elements* which correspond to (are instances of) an abstract data type.
2. **Production memory**, consisting of a set of production rules. Each rule consists of an antecedent and a consequent. An antecedent is a set of *condition elements* each consisting of patterns that are sensitive to the data in working memory. An antecedent (and its associated rule) is said to be satisfied if each of its condition elements is matched by a working memory element. A satisfied rule is called an instantiation. An instantiation cannot be executed more than once by the rule interpreter

(defined below). The consequent generally consists of input/output statements or actions that modify working memory.

3. **A Control Regime** (also known as the rule interpreter), consisting of a set of rules that determine the execution (or firing) of production rules. Control consists of a *match/select/execute* cycle. In the match stage, all rules whose antecedents are satisfied are collected into a conflict set. Multiple instantiations of the same rule can exist in this set. In the select stage, one instantiation is selected using a conflict resolution strategy. This is executed, which results in modification of working memory or input/output. In effect, control is driven by data in working memory.

The conflict resolution strategy can be based on a variety of criteria, many of which have been enumerated by Winston (1984). Commonly used strategies are *recency* where the rule matching the most recently deposited symbols in working memory (i.e. the most recent working memory elements) is chosen, and *specificity ordering* where a satisfied rule with the maximum number of condition elements is chosen. In addition *meta rules*, that is, rules whose sole function is to control execution of other rules, can be employed. The first two strategies control the execution of object level rules based on syntactic or domain-independent knowledge, whereas meta-rules express domain-knowledge which is used to guide rule execution.

Productions can be used to implement standard control constructs such as conditionals, iteration, and recursion. Since the basic unit of computation is an *if/then* statement, conditionals are naturally encoded as rules. Iteration is expressed easily since the control cycle is essentially a *dowhile* loop that produces all instantiations of each rule. Recursion is also easily implemented, particularly when the conflict resolution is based on recency (see Brownston et. al (1985) for examples of how to implement various control constructs using production systems).

It may be necessary sometimes to execute actions in a certain sequence. For example, a master file might need to be updated before it is used to generate monthly statements. In such cases, the firing of productions (actions) must be explicitly controlled. This can also be achieved using recency in conflict resolution, that is, by ensuring that working memory elements are created in an order opposite to that in which they need to be processed. Alternatively, it can be controlled using domain-specific knowledge encoded in meta-rules. The choice of what type of control strategy is adopted has important consequences for the modularity of a system. We shall illustrate this point in the context of an example in section 3.

3. Analysis of Representations

There are several appealing features about the dataflow diagram representation. The DFD captures two of the ubiquitous features of transaction processing systems, namely, the types of data involved in processing and how and where the data is transformed. Equally importantly, such a diagram is a powerful communication tool since users find that it provides an intuitively understandable picture of the processing logic. However, the DFD also presents two problems for the analyst.

One of the DFD's basic drawbacks is its imprecise semantics, that is, the meaning of flows and transformations is derived purely from the labels assigned to them rather than from the structural features of the diagram. Analysts often stop decomposing the processing bubbles leaving the logic between the dataflows into the process and emanating from it unclear. While it may be reasonable not to concern oneself with the logic at the higher levels within the DFD, the lack of precision at the lower levels can cause different users to interpret the same dataflow diagram very differently.

Secondly, the real, detailed processing logic is often not clear from the diagram, but is buried across several levels of the design. In structured design, the processing logic is often documented only in the 'mini-specs' - a separate document describing the processes at the lowest level in the hierarchy of DFD's. At times, analysts document the processing logic only in the code of the programs. In effect, there can be a basic indeterminism involved in going from the design to the code.

The resulting diffusion of system logic across many different representations is particularly problematic from a maintenance standpoint since it is not clear from the design what the repercussions of a change (driven by user requirements or otherwise) will be at the level of code. Over time, this can cause the design and implementation to become out of sync thereby rendering the design useless.

In the remainder of this section we show how to integrate analysis, design and implementation using productions as the underlying representation for each. We illustrate how system logic buried within a DFD can be made more explicit within the domain of the DFD representation by requiring each transformation bubble to possess only one output data flow. Once the DFD has been defined under this diagramming rule, we show how the DFD can be reinterpreted as an augmented AND/OR graph which can be rewritten as a set of productions. At this stage of the analysis the system is runnable to answer macro level questions regarding the relationship between sets of inputs and outputs.

The design can then be completed within the production system representation by writing the detailed logic of the lower level data transformation bubbles (the mini-spec) using production rules. The system is then runnable at the lowest level of system logic, providing the functionality required.

In order to illustrate this approach, we consider a typical transaction processing system, namely, an accounts receivable system. Figure 1 shows an abstract DFD representation (level 1) of one type of accounts receivable system. In this example, five processing steps and four data stores are used to represent receivables information. The first and second steps update the consolidated A/R file. The third step, which occurs after the file has been updated, matches charges with payments and write-offs. This results in a file containing records of invoices that have been paid or are to be written off and a file containing records of invoices where payment is still outstanding. The fourth step categorizes the invoices based on the age of the outstanding receivable and generates an aging schedule. The fifth step combines customer information with the aged accounting schedule and detailed accounting data to print customer monthly statements.

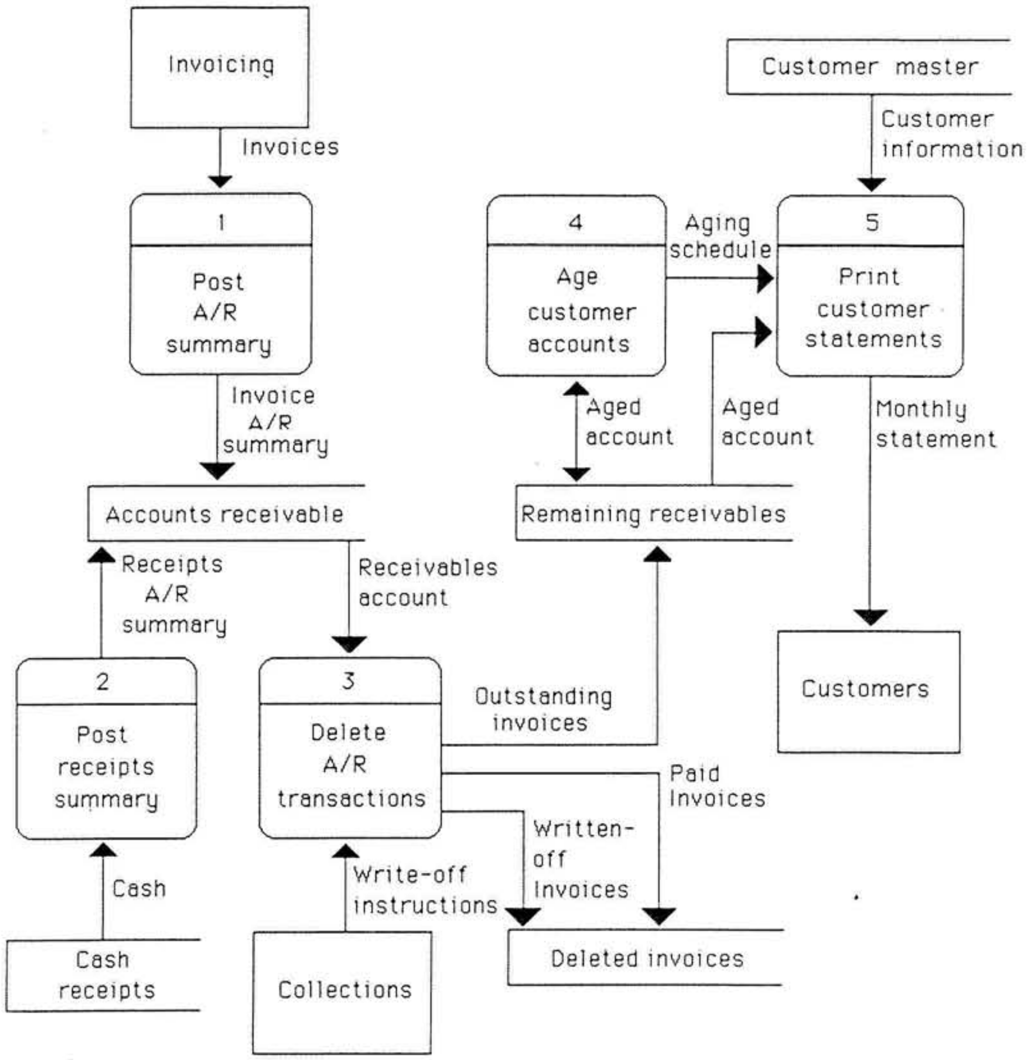

Figure 1. Five steps in processing accounts receivable information

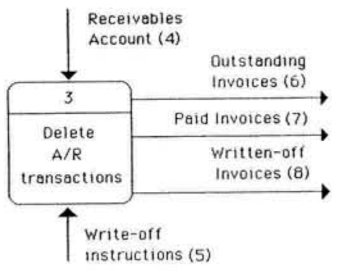

Figure 2. Process 3 of the A/R DFD

Bubble 3 in Figure 1 which has three outputs, illustrates the hidden logic which can occur in lower level DFD's. The correct interpretation of this bubble (shown in Figure 2) is that each account (data flow 4) with write-off instructions (5) must be classified as a written-off invoice (8), each invoice without such instructions must be classified as an outstanding invoice (6) or a paid invoice (7) depending on the balance due. However, this logic is not clear unless the reader ascribes the correct interpretation to the labels and identifies the relationships between various subsets of these labels. The ambiguity surrounding bubble 3 is resolved partially at level 2 (Figure 3) where it is clear how written-off invoices are produced. The ambiguity surrounding bubble 3.1 is similarly resolved at level 3 (Figure 4). In general, the logic surrounding the relationship between the incoming dataflows and the outgoing dataflows of a bubble become unambiguous when the bubble has exactly one output.

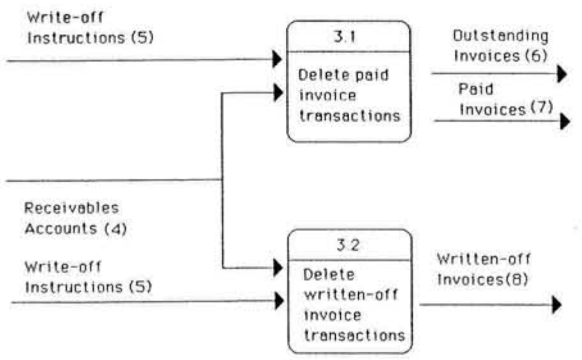
Write-off Instructions (5)
3.1
Delete paid invoice transactions
Outstanding Invoices (6)
Paid Invoices (7)
Receivables Accounts (4)
Write-off Instructions (5)
3.2
Delete written-off invoice transactions
Written-off Invoices(8)

Figure 3. Detail of process 3.0

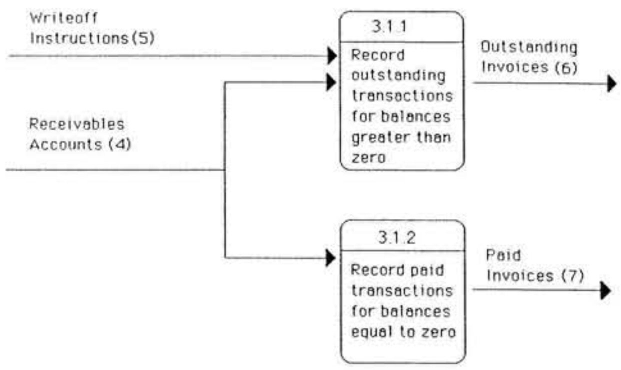
Writeoff Instructions (5)
3.1.1
Record outstanding transactions for balances greater than zero
Outstanding Invoices (6)
Receivables Accounts (4)
3.1.2
Record paid transactions for balances equal to zero
Paid Invoices (7)

Figure 4. Detail of process 3.1

The notion of requiring each lower level processing bubble to possess one, and only one outgoing dataflow allows the transformation to production rule notation to be done automatically. Such a diagramming rule also forces the analyst to consider in detail the relationship among the inputs and outputs. However, we note that there are circumstances where the representation of the system logic is clearer when this rule is violated. For example, in modeling payments from customers, we may have a check and an invoice arriving in the same dataflow and we show the invoice being sent to the accounts receivable process and the check going to a bank. Such situations seem to call for two outgoing dataflows from one processing bubble. Such situations can remain in the diagram as long as *all incoming dataflows are necessary in producing the outgoing dataflows.* Note, however, that the analyst may still draw the diagram with only one output dataflow by showing the dataflow split into two dataflows, one going to each subprocess (see Figure 5.).

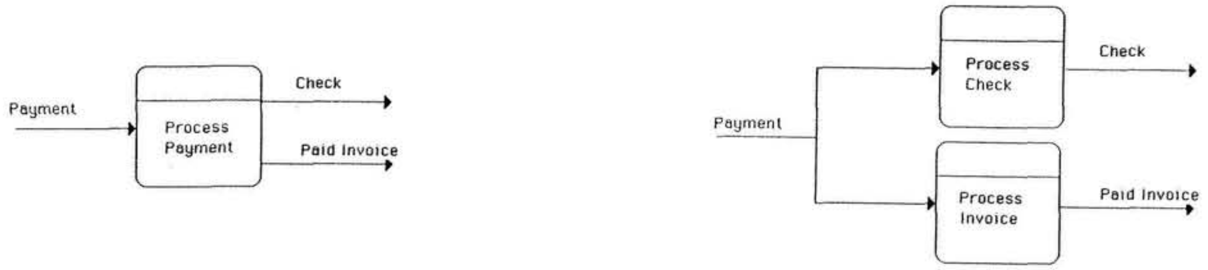

Figure 5. Creating single output processing bubbles

In the preceding discussion, we have been treating arrows as flows and bubbles as transformations not only in the traditional way, but also as entities that expose the logic of the application. More precisely, we have been interpreting arrows as predicates and bubbles as logical connectives. This can be graphically depicted as an augmented AND/OR graph. The AND/OR graph corresponding to the logic of Figure 1 discussed so far is shown in Figure 6. In this graph the half-circle symbols correspond to AND logic components (i.e., both incoming 'dataflows' must exist for the outgoing dataflow to be created). The rectangular symbol represents a datastore (the augmented part of the AND/OR graph) and the dotted arcs represent dataflows emanating from them, feeding back into transformations that update the datastores. This graph contains all the information in Figures 2, 3 and 4. In addition, the logic is explicit, defined in terms of the AND/OR primitives which have well defined semantics. For example, AND symbol 2, requires both dataflow CASH and dataflow RECEIVABLE ACCOUNTS in order to produce the output dataflow RECEIPTS A/R SUMMARY.

This graphical representation of the logic of the system can then be translated directly to a production system notation. In effect, the bottom part of Figure 3 then expresses a rule, interpreted as:

IF (4) is true and (5) is true THEN (8) is true

Similarly, Figure 4 expresses the rules:

IF (4) is true and (5) is true THEN (6) is true

IF (4) is true THEN (7) is true

More precisely, the conditions express existential quantification such as

If there exists a receivables account and there exists a write-off instruction for that account, then it is true that that account should be written off.

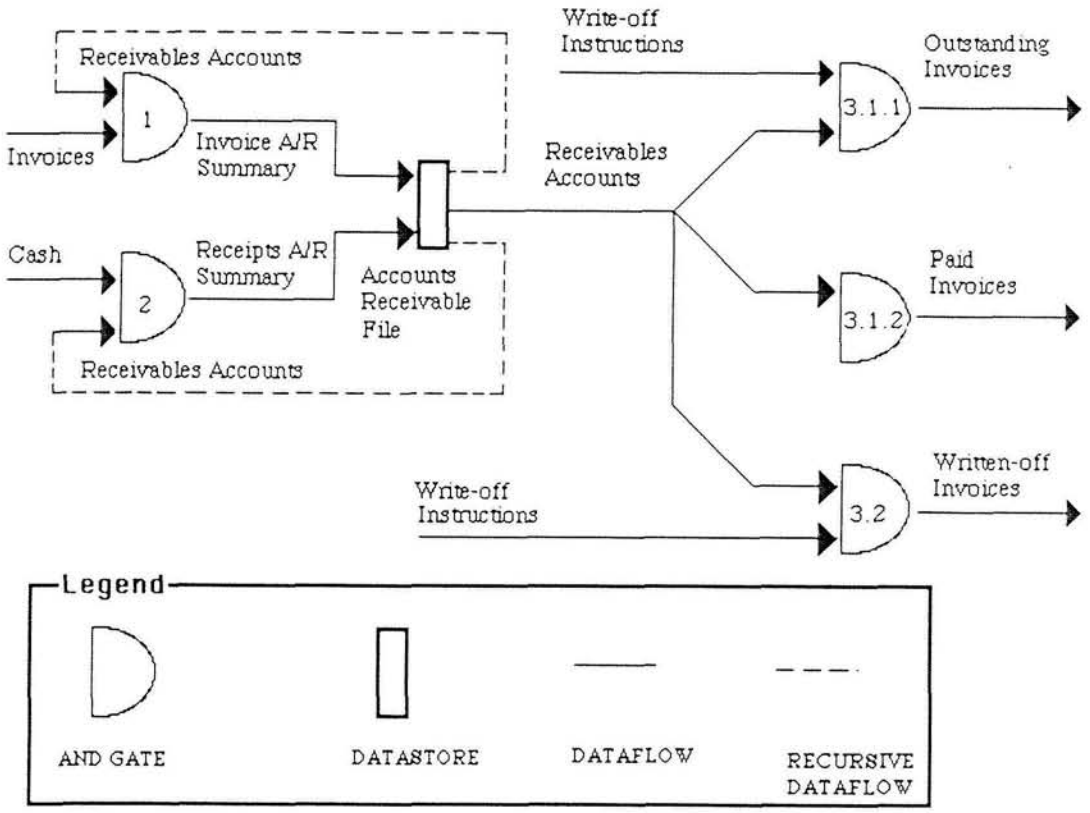

Figure 6. Augmented AND/OR graph for processes 1, 2 and 3.

The major implication of this view is that the design is "runnable" and can be used for explanation. For example, a question such as "**how** are written-off invoices produced" can be answered as "when a receivables account has write-off instructions"; likewise, the user or designer might want to know what outputs will result when no write-off instructions are provided.

Similarly, mini-specs are also expressible as productions. For example, consider the mini-spec corresponding to the first process, Post A/R Summary, which is essentially a sequential file update program. As rules, this logic is expressible as follows (variables are enclosed in angle brackets, with condition elements on separate lines):

```
IF   there is an unprocessed master record number <m>
     there is an invoice record <t> where <t> = <m>,
     there is NOT an unprocessed master number less than <m>
 THEN
     create an updated master record and write it.


IF there is an unprocessed master record number <m>,
   there is NOT any invoice record <t> where <t> = <m>,
   there is NOT an unprocessed master number less than <m>
 THEN
     write the master record without change.
```

The first rule updates a master record, whereas the second rule writes it out unchanged. The two rules assume that the records are not sorted (if they are sorted, the rules can be modified to achieve the efficiency obtainable by sorting the two files). The third condition elements ensure that the records are considered in ascending order. An encoding of the two rules above (ignoring file handling) is shown in terms of the two OPS5 rules in Table 1. Note that each condition element involves a data type (an 'object') that has a specified structure -- in this case a set of attributes or 'slots' that take on values. This is similar to the function provided by the data dictionary in structured analysis methods.

We view our representation as 'runnable' at many levels. Certainly if the mini-spec processing logic is defined then our representation becomes the implementation. This logic will contain not only the macro relationships among the dataflows but will also express the detailed transformation logic and the required control information which is not found in the dataflow diagram. As shown earlier, even without this detailed logic definition we still can run our representation, though with limitations. In moving from the minispec level to the lowest level DFD's we immediately lose the control information found in the minispecs and detailed transformation logic. As we 'move up' the representation hierarchy (i.e., move up levels in the dataflow diagram) relationships among the incoming and outgoing dataflows become less clear as expected. For example, in viewing Figure 1, we can ask the same question we asked earlier: "**how** are written-off invoice produced". The answer will be 'When we have either receivable accounts or write-off instructions or both'. We do not know what inputs are directly associated with what outputs. In general, without having the information contained in the detailed design, we are left with uncertainty - the uncertainty which is comprised of 'or' conditions regarding the requirements for input for a particular output.

The above analysis can be summarized as follows. Since dataflow diagrams are considered a useful communication tool between users and analysts, it makes sense to retain them as an analysis tool. However, the logic underlying the design that is buried across various levels can be made explicit and represented in terms of production rules (corresponding to the AND/OR graph). If one describes dataflows as objects consisting of attributes and values and follows the one-output diagramming convention, the rules can be constructed automatically. Control information will be expressed only in the production rules corresponding to the mini-specs as described in Table 1.

4. Discussion

A representation can be evaluated along several criteria. Borgida et. al (1985) categorize evaluation criteria into two broad categories which we shall adopt here, namely *expressiveness* -- the ability of the representation to capture the relevant features of the concepts being modeled, and *organization* -- organizing the knowledge so that it is easily understood by designers and users, and so that inconsistencies in a design (at least syntactic ones) can be detected easily. Finally, the implementation language should provide a natural and efficient encoding of the detailed design.

4.1. Expressiveness

In terms of expressiveness, the representation should have the following properties:

1. it should describe properties of the concepts in the application domain,
2. it should describe change,
3. it should make explicit the constraints relevant to the problem,
4. it should facilitate incremental specification and development since many problems are difficult to specify at the outset and requirements often change.

We have not focused on the first two criteria in this paper. In general, however, the second criterion, namely, describing temporal events is probably the most difficult to incorporate into a representation. While some progress has been made in limited domains, most representations tend to ignore the time element. Certainly, dataflow representations and the AND/OR graph both ignore time. The first criterion is dealt with

Object declarations:

```
(literalize acct-rec ; data type corresponding to a master record
            mno      ; field of mrec: the master record number
            status)  ; field of mrec: indicates whether record
                     ; has been processed

(literalize invoice  ; data type corresponding to a transaction rec
            tno)     ; field of trec: the transaction record number
```

Rules:

```
(p update
  (acct-rec ^mno <m> ^status unprocessed)
  (invoice ^tno {= <m> <t>})
 -(acct-rec ^mno {< <m>} ^status unprocessed)
 -->
  (write (crlf) Processing master number <m> and trans number <t>)
  (modify 1 ^status processed))

(p noupdate
  (acct-rec ^mno <m> ^status unprocessed)
 -(invoice ^tno {= <m>})
 -(acct-rec ^mno {< <m>} ^status unprocessed)
 -->
  (write (crlf) Processing master record number <m>)
  (modify 1 ^status processed))
```

Working Memory:

```
Master file (working memory elements):       Transac file (WMEs):

 ------------------------------------           ----------------
| acct-rec ^mno 1 ^status unprocessed |        | invoice ^tno 2 |
 ------------------------------------           ----------------
| acct-rec ^mno 2 ^status unprocessed |        | invoice ^tno 4 |
 ------------------------------------           ----------------
| acct-rec ^mno 3 ^status unprocessed |
 ------------------------------------
| acct-rec ^mno 4 ^status unprocessed |
 ------------------------------------
| acct-rec ^mno 5 ^status unprocessed |
 ------------------------------------
```

Table 1

to varying degrees by different representations. Data dictionaries capture to a small extent the properties of the symbols (concepts) used in modeling the application domain. More sophisticated schemes that use object oriented representations have been described by Borgida et.al (1985) and Dhar and Jarke (1988).

With respect to the third criterion, an important advantage of the production rule interpretation of flow diagrams is that it makes explicit the functional constraints that characterize the problem. This is important from a validation standpoint since designers and users now have a uniform interpretation of the problem. Although we have interpreted dataflow diagrams in terms of rules, more generally they may be viewed purely declaratively, as constraints. These can be modeled using production system languages such as OPS5 or logic programming languages such as PROLOG. The AND/OR graph corresponding to the rules can be traversed in various ways depending on the control strategy adopted. Again, this is useful from a validation standpoint since the design can be "run" to produce simulations that can be examined by the designers and users. Essentially, the simulations would involve backward chaining (i.e. how is output "X" produced) or forward chaining (what will be the result/outputs from a given set of inputs).

The fourth criterion raises some important issues. Conceptually, the modularity (loose coupling) provided by the production system architecture should make such systems easy to modify in response to changes. This is because rules are meant to be self contained pieces of declarative knowledge that are automatically invoked under appropriate problem conditions. In practice, however, it is difficult to avoid embedding control information in rules. For example, if the right hand side of a rule results in the creation of two working memory elements, the order in which they are created can affect the behavior of the system. This leads to situations where the rule may be written not as an independent module but with the control strategy of the interpreter in mind. Such situations often violate the spirit in which the expert/user expressed the knowledge, that is, as a truly declarative piece of knowledge. While massaging control information into such rules can be employed to great advantage, unless the programmer exercises extreme caution, modularity can be lost, making the program extremely brittle to changes.

Unfortunately, the embedding of control information into productions can also result from the designer not understanding the problem domain adequately or not bothering to express adequate domain knowledge in the system. For example, in Figure 1, processing invoices in the right order (i.e. first post, then record, then age) could be implemented using the control structure of the interpreter. However, if another step were added to the design and its corresponding rules need to be specified, the antecedents of the new rules and most probably the consequents of other rules would have to be fashioned carefully in order to have the new rules fire at just the right time[1]. On the other hand, if the designer expressed sequencing knowledge explicitly (say using a meta-rule that expresses knowledge about sequencing), the task of adding the new rules becomes much simpler. This latter approach requires making explicit as much of the domain knowledge as possible, leaving as little as possible to the syntactic criteria used by the interpreter, thereby minimizing the effect of control knowledge expressed in the object level rules. For a detailed analysis of the properties of meta-rules, the reader is referred to Davis (1982).

4.2. Organization

Borgida et.al (1985) propose that in terms of organization, a representation should have the following properties:

1. it should enable the designer to deal with abstraction,
2. it should provide the designer and users with multiple conceptualizations of the problem, thereby potentially reducing the likelihood of errors.

An attractive feature about dataflow diagrams is the leveling technique that is useful in dealing with abstraction. Production systems are equally powerful in this respect since productions can represent knowledge at different levels of abstraction. Specifically, the AND/OR graphs corresponding to sets of rules can be viewed at various levels of abstraction in the same way as levels of dataflow diagrams. Also, exactly the same integrity rules apply for ensuring consistency in inputs/outputs among the different levels (i.e., a bubble must have the same inputs/outputs as its detailed breakdown) and in labeling.

Multiple conceptualizations can help make a description more complete. If these conceptualizations must be somehow consistent (as is usually the case), this type of redundancy in description reduces the likelihood of error. By preserving the dataflow interpretation and enhancing it with a logical one, we have provided a more complete and powerful modeling capability. For example, Figure 5 can be interpreted both as a dataflow diagram, and by ignoring datastores, as an AND/OR graph. This dual property of the graph enables us, as pointed out above, to employ the same consistency checking rules that are applied to dataflow diagrams in CASE tools.

Finally, researchers have developed graphical techniques for displaying productions (Lewis, 1983). Although we have not advocated drawing DFD's for the mini-specs, if the mini-specs are written as production rules, they may be reviewed graphically through using systems such as GETREE. This provides a visually rich, and consistent representation for debugging the micro level design logic.

[1]Such situations invariably lead to surprising behaviors that the designer never envisioned.

4.3. Implementation Language Considerations

At the outset of the paper we mentioned that using productions as the representational formalism unifies analysis and design in that rules can be used to describe the abstract design and can also serve as the code. In practice, however, two considerations must be taken into account in deciding whether the code should be implemented as productions or in a procedural language.

The first consideration is whether the code (or perhaps more appropriately, the mini-spec) is more naturally represented in terms of declarative statements or procedures. Clearly, problems involving numerical analysis or approximation (linear programming, differential equation solvers, Monte Carlo techniques) are better coded as procedures. Brownston et.al (1985) also suggest that problems that are highly sequential with a precisely specifiable control are better implemented using conventional languages. In general, however, the decision is not a straightforward one. For example, in the previous section we coded a simplified sequential file update program (typically viewed procedurally) using two productions. It is debatable, however, whether this is a more natural encoding of the update process.

The other important consideration in the implementation language decision is efficiency. In general, production systems are one to two orders of magnitude slower than traditional languages. Thus if minimizing processing time is of prime importance, using productions for implementation is not a good idea. However, with the dramatic hardware improvements and faster implementations of production systems on the horizon, acceptable levels of performance should be easier to achieve using production systems.

If the production implementation is too slow for practical purposes, it could be transformed into a procedural language. Although there is not a standard set of transformations for doing this, it can be relatively straightforward if the rules are well defined. In effect, the production system could be used as a prototyping environment to elicit and make explicit all domain knowledge before translation. Clearly, however, the advantages of such an approach are diminished, particularly for applications where the knowledge or requirements often change.

5. Summary

It has been widely recognized that the boundary between structured analysis and structured design is fuzzy, requiring a designer to exercise a great deal of judgement in deciding how to use the products of analysis for design. Accordingly, there have been attempts to remedy this situation along several directions. These include specifying more precisely the semantics of dataflow diagrams (Adler, 1988), adding control and timing via additional primitives (Ward, 1986), and elaborating entities (making distinction among several types) in flow diagrams in order to effect a smoother transition between analysis and high level design (Shoval, 1988).

The approach we have outlined represents another, somewhat different method for unifying analysis and design. By virtue of the dual representation of AND/OR graphs, the simplicity and expressiveness of structured analysis is maintained and a uniform, largely declarative, representation is provided that is runnable at all levels of the design. This model can be used for purposes of simulation and explanation.

At the outset of this paper, we cited anecdotal evidence from practitioners supporting the use of production systems as a prototyping tool. Based on our analysis, we conjecture that the advantages arise because the formalism enables users and analysts to express the logic of the application declaratively and incrementally (even though the implementation can lead to surprises). By laying out this logic using AND/OR graphs and employing its dual interpretation as a flow diagram, we feel that designers have a powerful analysis tool at their disposal. However, as we discussed in the previous section, at the implementation level, production systems can become increasingly brittle as the number of productions grow unless great care is taken to ensure explicit control of reasoning. In summary, production systems do not guarantee modularity as is often asserted.

Finally, the implications for analysis tools should be apparent. A major function of current CASE tools is that they provide assistance in leveling and syntactic checks. If such tools were extended to express rules and generate the AND/OR graph and flow diagram corresponding to them, they would bridge the gap between analysis and design. This would provide analysts with a comprehensive tool to specify in a top-down manner the complete functionality of systems.

References

Adler, M., An Algebra for Data FLow Diagram Process Decomposition, *IEEE Transactions of Software Engineering*, volume 14, number 2, Feb 1988.

Alford, M., SREM at the Age of Eight; The Distributed Computing System, *IEEE Computer*, April 1985.

Borgida, A., Greenspan, S., and Mylopolous, J., Using Knowledge Representation for Requirements Modeling, *IEEE Computer*, April 1985.

Brownston, L., Farell, R., Kant, E., and Martin, N., *Programming Expert Systems in OPS5*, Adisson-Wesley, Reading Mass, 1985.

Davis, R., Teiresias: Applications of Meta-level Knowledge, in *Knowledge-Based Systems in Artificial Intelligence*, pp 227-490, Randall Davis and Douglas Lenat (eds), McGraw-Hill, New York, 1982.

De Marco, T., *Structured Analysis and System Specification*, Yourdon Press, New York.

Dhar, V., and Jarke, M., Dependency Directed Reasoning and Learning in Systems Maintenance Support, *IEEE Transactions of Software Engineering*, volume 14, number 2, Feb 1988.

Eliason, A. L., *Online Business Computer Applications*, Science Research Associates, Inc., 1987.

Gane, C., and Sarson, T., *Structured Systems Analysis: Tools and Techniques*, Prentice-Hall, 1979.

Lewis, J.W., An Effective Graphics User Interface for Rules and Inference Mechanisms, *Human Factors in Computing Systems, CHI'83 Conference Proceedings*, pp. 139-143, A. Janda (Ed.), ACM, 1983.

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

Pressman, R.S., *Software Engineering: A Practitioner's Approach*, McGraw-Hill, New York, 1987.

Ross, D.T., Applications and Extensions of SADT, *IEEE Computer*, April 1985.

Shoval, P., ADISSA: Architectural Design of Information Systems Based on Structured Analysis, *Information Systems*, volume 13, number 2, 1988.

Ward, P., The Transformation Schema: An Extension of the Data Flow Diagram to Represent Control and Timing, *IEEE Transactions on Software Engineering*, volume 12, number 2, Feb 1986.

Winston, P., Artificial Intelligence, *Adisson-Wesley*, Reading Mass, 1984.