1986

An Integration Of Decision Tables And A Relational Database System Into A Prolog Environment (Knowledge Representation, Rule-Based Systems, Logic Programming).

Akram I. Salah
University of Alabama at Birmingham

Salah, Akram I.

AN INTEGRATION OF DECISION TABLES AND A RELATIONAL DATABASE SYSTEM INTO A PROLOG ENVIRONMENT

The University of Alabama in Birmingham PH.D. 1986

PLEASE NOTE:

In all cases this material has been filmed in the best possible way from the available copy. Problems encountered with this document have been identified here with a check mark ✓.

1. Glossy photographs or pages _____
2. Colored illustrations, paper or print _____
3. Photographs with dark background _____
4. Illustrations are poor copy _____
5. Pages with black marks, not original copy _____
6. Print shows through as there is text on both sides of page _____
7. Indistinct, broken or small print on several pages ✓
8. Print exceeds margin requirements _____
9. Tightly bound copy with print lost in spine _____
10. Computer printout pages with indistinct print _____
11. Page(s) _____ lacking when material received, and not available from school or author.
12. Page(s) _____ seem to be missing in numbering only as text follows.
13. Two pages numbered _____. Text follows.
14. Curling and wrinkled pages _____
15. Dissertation contains pages with print at a slant, filmed as received ✓
16. Other _____

_____

_____

University
Microfilms
International

AN INTEGRATION OF DECISION TABLES AND A RELATIONAL DATABASE SYSTEM INTO A PROLOG ENVIRONMENT

by

Akram I. Salah

A Dissertation

Submitted in partial fulfillment of the requirements for the degree of Doctor of Philosophy in the Department of Computer and Information Sciences in the Graduate School, The University of Alabama at Birmingham

BIRMINGHAM, ALABAMA

1986

ABSTRACT OF DISSERTATION
GRADUATE SCHOOL, UNIVERSITY OF ALABAMA AT BIRMINGHAM

Degree Ph. D. Major Subject Computer & Info. Sciences

Name of Candidate Akram I. Salah

Title An Integration of a Decision Table System and a Relational Database System into a Prolog Environment

A conceptual system based on integration of notions from logic programming, relational database theory and decision tables processing, along with prototype implementations of some central components, is presented. Part of the system is dedicated to representing rule systems expressible in decision table forms, taken as an extension of typical production rules. The rules and associated descriptive information are formulated so that they can be stored in a relational database, in a manner compatible with conventional data, and can be manipulated by programs written as applications code in a "Database Prolog" system developed by M. Bruynooghe.

Theoretical foundations for the integrated system are given through: modifications and generalizations of formal definitions for decision tables, based on a set-theoretic triple given previously by a CODASYL task force; and a data organizational scheme designed to explore data models based on term, predicate, and hybrid representations, several of which are illustrated through Prolog prototype programs.

The practical utility of the system conceptualization is assessed in terms of a selected application originating in differential diagnosis and involving decisions of a more complex nature than typically found in production systems.

A prototype implementation illustrates central concepts in this case, its compatibility with the main system design being assessed in the discussion. The theoretical background of the set-theoretic approach is utilized as an assessment device for rule-based systems properties and for promotion of automatic handling of portions of expert systems building process.

The integrated system concept has several significant attributes. It offers flexibility in expressing systems, e.g., more than can be secured from any one of its individual components. It admits analysis and query processing for rules as well as for data. Its underlying theory permits new perspectives on known decision table properties and the prototype implementations suggest novel mechanisms to export relational database concepts to adjacent domains. Comparing potential with powerful expert systems methodologies such as R1-Soar suggests that further work along the lines of this thesis would be productive.

Abstract Approved by: Committee Chairman Kevin D. Reilly

Program Director Warren J. Jones

Date 6/6/86 Dean of Graduate School Anthony Barnard

TABLE OF CONTENTS

ACKNOWLEDGEMENTS

I have to express my appreciation to Dr. Kevin Reilly, chairman of my Ph.D. committee, for his support and counsel during the conduct of this research. I thank Dr. Warren Jones, Chairman of the Department of Computer and Information Sciences, for providing moral support and scientific advice. I thank Dr. Anthony Barnard, Dean and Co-Director of the Graduate School, for his support and his comments on the dissertation. I also thank all other committee members Dr. Pardip Dey and Dr. Fouad, and special thanks to Dr. Chao-Chih Yang whom with I started this research and who continued helping me even after he left UAB.

I thank the Graduate School of The University of Alabama at Birmingham for the financial assistance provided by a graduate research fellowship.

I want to express special appreciation to my wife, Shura, for her help and encouragement and for putting up with all the inconveniences during my graduate studies. She and our children, Husain, Ibrahim, and Aly, make the effort worthwhile.

LIST OF TABLES

LIST OF FIGURES

Appendix C

1. INTRODUCTION

This dissertation adduces results of research on an integration of a Prolog system, a Decision Table system, and a Relational Database system. During the past decade, each one of these systems has had major successes and achievements, individually. This research was based on a hypothesis that an integration of the three systems yields a system with greater power than each individual component. Here, a summary of the research and its results is provided.

This research produced a number of publications [59, 60, 64, 65, 67, 70] and technical reports [58, 66, 68], as well as other papers submitted for publication [69, 71]. The activities achieved in this research include installing a Prolog-Database interpreter [5] on VAX 11/750 under Unix and then adding some Prolog programs to be utilized by its management system in order to facilitate easier query expression [64]. This extended version forms the environment for the integration.

To characterize the different phases of the research, this dissertation cites five documents (copies of them appear in appendices A-E). Two of these papers represent studies conducted to provide mathematical and programming foundations for decision tables in order to devise their integration into the system. The third proceeds from formal foundations to justify integrating decision tables into a relational database system and then describes a configuration for an integrated system. The other two papers demonstrate interrelationships, both in theory and application, between the integration approach and the current activities in Rule-Based systems and Expert Systems.

The body of the dissertation includes a general discussion of the research: how it was conducted; results that were developed; and future plans. Details are supplied in the individual papers in the appendices. In section 2 , Prolog, relational database, and decision table systems are briefly introduced and their history, uses, advantages, and limitations are discussed. Then, section 3 discusses pairwise relationships among the three systems, showing that features in one system can contribute to overcoming limitations in others. Section 4 includes a description of an integrated system, including elements from all three systems, with a commentary on its features and significance. Finally, section 5 discusses how the integrated system, and methods evolved through its development, motivate other studies in expert systems and rule-based systems. Section 6 covers conclusions and future prospects.

2. OVERVIEW

This section is dedicated to introducing the three systems involved in the integration. Each system is briefly overviewed, pointing out features, advantages, and limitations. This overview is a prelude for discussions in subsequent sections.

2.1 Prolog System

Prolog (Programming in Logic) is a high level programming language that is designed to emulate characteristics of first order predicate logic. It has a built-in theorem prover which operates in a top-down, depth-first search and backtracking manner. All inferences made in Prolog result from purely syntactic operations. This means that the system provides neither interpretation of variables nor evaluation of formulas.

The first order predicate logic is a logic which allows quantifications over individual variables but not over predicates or function symbols. Over and above the propositional logic (0th order), the first order predicate logic incorporates the logic notions of terms and predicates in addition to quantification.

Logic Programming, which constitutes the formal foundation for Prolog, works on clausal forms of logic [8, 41]. A clausal form requires that all variables be implicitly universally quantified and all formulas be in disjunctive normal form. These restrictions result in no loss of deductive power, since all expressions of standard logic, which include existentially quantified variables, can be converted to equivalent expressions using universal quantification [8]. Prolog works on a subset of clausal forms called Horn clauses in which a clause is restricted to having at most one unnegated atom.

Prolog was originally developed in France [63] based upon the work of Kowalski [38, 41]. Prolog has attracted attention since its inception in the mid 70's. Many computer applications, such as natural language processing, operating systems, drug design and analysis, and architectural design have been done using Prolog [41, 48]. Of special interest to this research is its use in relational database systems, which will be discussed later in this dissertation.

An important advantage of Prolog lies in its strong formal basis. This facilitates Prolog implementation for problems that have a formal specification. Any problem that can be expressed in clausal forms can directly be written as a Prolog program. Second, Prolog is not a "strictly typed" language. This means that a variable in Prolog can be bound to values of any type, i.e., character, integer, or floating point. Furthermore, a variable in Prolog may be bound to a simple value or a structure such as a string, a list, or even a function symbol. This property, in our view, makes Prolog a more user-friendly language. A user does not have to excogitate the internal representation of variables when his problem is expressed in Prolog. Third, the semantics of Prolog expressions can be understood in two ways, a procedural way and a declarative way. The procedural semantics is perhaps more conventional, and describes the sequence in which a program is executed. The same clause can be understood, declaratively, as a relationship between a conclusion and a set of conditions [48], a property that makes a Prolog program self-declarative and easy to understand.

Another advantage is that Horn clauses, in theory, define two types of nondeterminism in the execution of programs. A Prolog interpreter is defined as a procedural interpretation of Horn clauses [41]. Prolog works its way through its clauses in a top-to-bottom, left-to-right manner and employs backtracking in case of a failure. Non-determinism increases the expressive power of Prolog. However, it affects its efficiency since backtracking is inefficient when employed in conventional machines.

Prolog provides ways to control backtracking, but they are not automatic. Thus, it is the responsibility of the programmer to express a problem in an efficient way.

2.2 Relational Database Systems

A Relational Database System (RDBS) is a database system based on the data model of Codd [15] in 1970. A relation in such a system may be envisioned as a table of values. Names are generally associated with the columns of such a table; these names are called *attributes.* Values of an attribute A_i are taken from a finite set called the *domain* of this attribute, D_{A_i}. A relation R with attributes A_1, ..., A_n defines a *relation scheme* $R(A_1, ..,A_n)$, whereas the values in R are taken from the complex product of D_{A_1}, ..., D_{A_n}. An element in R, which corresponds to a row in the table of R, is called a *tuple.* A specific relation, i.e., with specific tuples, is said to be an *instance* of the relation scheme.

Not all instances of a relation scheme have meaningful interpretation, i.e., they do not correspond to valid sets of data according to the intended semantics of the database. Therefore, a set of constraints, referred to as *integrity constraints,* associated with a relation scheme ensures that the database meets the intended semantics. Thus, a RDBS scheme consists of a set of relation schemes together with a set of integrity constraints. A database instance is a collection of relation instances, one for each relation scheme in the database scheme.

Formal foundations of RDBS have been proved to be complete and sound. Such strong formal foundation facilitates the development of algorithmic processes for design and construction of RDBS as well as for enforcing integrity constraints to control updating, deletion, and insertion anomalies. Also, the formal foundation contributes to the development of a formal base for query languages such as relational

algebra, tuple relational calculus, or domain relational calculus. The process of construction and manipulation of such a system can be automated. The design procedure of RDBS is not automated yet, but design criteria and many tests on the quality of an already constructed RDBS are well defined [85]. Because of its strong foundation, RDBS have become the formal standard for database systems.

A limitation on RDBS is that it requires that the domains of the attributes contain only constants. This is not a major limitation for databases since data items are mostly constants, but it limits the use of a RDBS with its powerful formal concepts to atomic data items only. For example, a data item cannot be a variable or a function symbol.

2.3 Decision Table Systems

A Decision Table (DT) is a technique used to describe and analyze problems that contain a decision situation. Such problems are normally characterized by one or more conditions, such that a state of these conditions determines the execution of a set of actions. A DT is a structure that gathers a set of related *decision rules,* each rule identifying one state of the conditions and associated actions.

DTs were introduced in the 1950's and a prominent use of them has been as a problem description tool. In literature, many DT properties have been studied in relation to other description tools such as narratives, flow charts, and decision trees [43, 52]. Procedures have been developed to automate the process of producing computer programs from a DT description of a problem.

Despite the longevity, it seems that formal definitions for DTs have not been a point of emphasis until the CODASYL Decision Table Task Group report in July, 1982 [12]. This report defines a DT as a relation that maps from conditions to actions, and

then devotes considerable attention to DTs as functions. For the purpose of this research, the CODASYL definition has been studied and modified slightly, the more general definition form is explored, and various DT formats and properties are elucidated, as part of an examination of DTs in relation to functional description, relational databases, and the use of the logic programming language Prolog for compatible implementations.

Appendix A contains: "Decision Table Properties, Representations, and Implementations," a paper submitted for journal publication. This paper covers a set-theoretic definition of decision tables, a restatement of DT properties in view of the set-theoretic definitions, and representations of decision tables with compatible implementations. Also, the paper introduces a notion of "query processing" policies for decision tables, which is used later to characterize flexibility of our proposed DT representations.

During the long use of DTs in problem description and analysis, many advantages of DTs have been cited. DTs are compact and clear when compared with flowcharts and narratives. Also, DTs are much easier to check for missing or contradicting rules [35, 71]. DTs have been used limitedly in the early stage of software development as a description tool and for documentation. In this thesis, DTs are also employed as a structured rule representation. Design criteria for a well structured DT system and procedures for checking on its properties as a rule system still remain to be defined.

3. PAIRWISE SYSTEMS INTEGRATIONS

Our proposed integration of Prolog, DT, and RDBS is based on an abstraction of basic concepts of each individual component, analogies among concepts of the components, and augmentation of the expressive power of each component to overcome limitations in other components. In this section, a discussion of dual relationships of the three components, (i.e., Prolog-RDBS, DTs-Prolog, and RDBS-DTs) proceeds, to show how advantages in one component can contribute to the other.

3.1 Prolog and RDBS

During the past several years, many studies concerning relationships between logic and RDBS have been performed. Some of these studies discuss stating RDBS properties, such as integrity constraints and inference rules, in a standard form of logic [22, 26, 27, 30, 69, 85]. RDBS properties are easier to prove when they are expressed in logic. In addition, it is easy to deduce new properties from the known ones. Other studies have focused on using Prolog as a query language for a RDBS [3, 74, 85]. Despite Prolog's inefficiency relative to conventional programming languages, in some applications, its expressive power, simplicity to understand, and ease to use make it worthy to be integrated with a RDBS, as, e.g., in Bruynooghe's system [5]. Recent studies [3, 85], show the potential contribution of Prolog to a RDBS.

First, as mentioned above, Prolog is not a strictly typed language. This may seem as a disadvantage in some applications where types for variables need to be enforced on their values. But since a RDBS strictly requires that each value has to be taken from a predefined domain, the validity of values is already enforced in a RDBS.

Thus, the use of Prolog as a query language is advantageous to the user without loss of RDBS integrity. Second, it is easy to show that Prolog can emulate properties of other query languages for a RDBS. Queries expressed in relational algebra, tuple relational calculus, or domain relational calculus can be transformed systematically to Prolog expressions. Third, Prolog is a programming language that can be used for expressing application programs. This means that database queries may be embedded into application programs if Prolog is used. Many other languages are defined to be either, mainly, a programming language or a query language, which creates difficulties in communication between application programs and a database. These difficulties can be avoided if Prolog is used for both queries and application programs for a database.

In summary, Prolog provides for a RDBS the following: it allows the user to extract more meaning from the data; it easily allows addition of meaning to the data; it supports a wider range of queries; and it easily supports sophisticated modeling techniques.

This research work includes installing a Prolog RDBS and adding Prolog programs that can be utilized by its management system to facilitate easier query expression and better query performance. These studies result in what is referred to as the Extended Prolog Relational Database Management System in [85]. It runs on a VAX 11/750 under Unix.

3.2 Decision Tables and Prolog

The study of DTs in comparison with Prolog shows very interesting points. Neither DTs nor Prolog is strictly procedural, though both can be utilized at least semi-procedurally. A Prolog clause states the dependency of a conclusion on a conjunction of conditions. A decision rule associates a condition state with some actions in a problem. A Prolog program is a set of clauses, and a DT is a set of decision rules.

However, whereas a DT is a description tool and requires additional coding to manipulate, Prolog clauses are directly executable. If we combine the two, as we propose, DTs can be used to describe problems and Prolog can execute the DT rules.

Some of the most important DT forms can be represented as Horn clauses in various ways, all of which are executable in Prolog [67, 68, 71]. First, a DT can be viewed as a function. The domain and range of such a function are expressed in Prolog assertions. The mapping from the domain to the range is performed by other Prolog clauses. Several variations of functional representations are explored in this dissertation. Secondly, a DT can be viewed relationally as a disjunction of decision rules, each one represented as a logic implication. Each of these implications can directly be expressed as a Prolog clause. Alternatively, a DT can be implemented as a set of Prolog assertions, each representing a decision rule with additional Prolog clauses defining how to extract information from these assertions. Thus, DTs and Prolog can be used together to facilitate easier development for programs by using DTs for description and Prolog for implementation.

Appendix B contains a paper to be submitted, "A Logic Programming Perspective on Decision Table Theory and Practice," which discusses several implementations of a prototype decision table problem in Prolog. Also, it discusses a logic programming view of DT practices such as don't care, ordering issues, and systems of tables. The paper shows that Prolog is more natural and easier, compared to conventional programming languages, for implementing DTs. It also shows that the compatibility of Prolog and DTs can be utilized to define a systematic way to develop software systems.

3.3 RDBS and Decision Tables

A relation in a RDBS can be viewed as a tabular representation of related data. RDBS theory restricts data items to be constant values. DTs are tabular structures that

describe decision situations in a problem. Since DTs represent decision rules, they may include non-constant values, i.e., variable names, function symbols, or relation symbols. Thus, DTs and relations in a RDBS are analogous in being tabular forms, but each of them represent different type of information.

However, in simpler cases, a DT also contains only constant values for entries. In more complicated DTs discussed by us, non-constant data is allowed in the action portion of the table. Viewing a DT system as a RDBS adds advantages to both systems. First, for a RDBS, it extends its territory to include storing and manipulating not only data items but also relationships among data in the form of decision rules. Secondly, for a DT system, it provides a storage and manipulation media. In addition, design criteria and procedural methods for testing a well constructed relational system may be borrowed and applied to a DT system.

To achieve this goal, a part of this research is devoted to formal definition of DTs as a relational form compatible with a RDBS. The formal definition generalizes the CODASYL set-theoretic definition to accommodate the changes we have made. Also, it includes expression of DT properties in the light of the generalized definition. The research covers storage and manipulation of a DT system in a RDBS.

The relationship between a DT system and a RDBS forms grounds for adding Prolog code to Bruynooghe's Prolog RDBS to store a DT system. Appendix C contains: "An Integration of a Rule Representation into a Relational Database System," a paper submitted for conference publication. The first part of the paper reports results of a study on the relationship of DTs & RDBS. This is a prelude to defining EXPRD, our prototype system, which demonstrates how our ideas can be realized in practice.

4. EXPRD: AN INTEGRATION OF THE THREE SYSTEMS

An integration of the three systems is defined in three stages: First, integration of added Prolog code into a Prolog RDBS such that use of Prolog as a query language and as a management system is made more flexible; second, extending the capabilities of Prolog RDBS to store and manipulate decision tables; third, extending the management system such that it can recognize non-constant DT entries, and, automatically, perform evaluation of their values when needed.

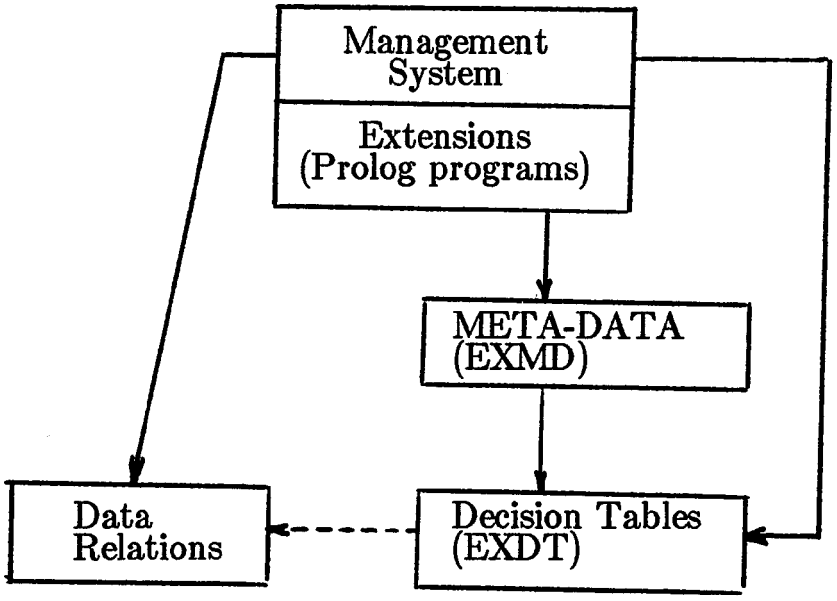
These three stages are documented as follows: The first stage appears as an appendix in Dr. Chao-Chih Yang's text book "Relational Databases," [85, 64]. The second stage appears in Appendix A of this dissertation, in the section entitled "Representations and Their Compatible Implementations of Decision Tables." Also, in Appendix B a discussion of representing a DT as a "single relation" covers a logic programming view for this stage. Finally, the third stage appears in Appendix C, starting by a comparison between a DT system and a RDBS which shows that some DTs can directly be stored in a RDBS, and then introducing an integrated system which can represent a more general DT system.

4.1 Configuration of the Integrated System

The integrated system can be viewed as a generalization of a RDBS since it evolved from a RDBS. Alternatively, and more apt, is description of it simply as a more general relational processing system than a RDBS. It is capable of storing and manipulating relations that contain variable names, function symbols, and relation symbols as well as constant symbols. The special case in which a relation contains only constants

represents the normal data relations of a RDBS description of a problem. A relation that contains non-constant values, represents relationships among data items in the form of decision rules.

To handle decision tables, EXPRD manages two additional components besides data relations. The additional components are: 1) a meta-data component, EXMD, which contains descriptions of DT entities, e.g., DT names, condition subjects, action subjects, types of values; and 2) a rule base, EXDT, which is a DT system constructed according to the specification in EXMD. These two components are stored in the form of relations in the RDBS. Prolog programs are added to the management system to facilitate proper interpretation of data and rule elements. These Prolog programs utilize meta-data to perform such interpretations. The following is a general description of the two components, EXMD and EXDT. The overall system description appears as:



4.1.1 EXMD: The Meta-Data Component

Meta-data is information about entries used to fill DT rules. This information must be defined, as needed, by a user to specify a particular rule system. EXPRD utilizes this information: a) to construct relations to store DTs; b) to enforce integrity constraints, when adding a new decision rule or updating an existing one; and c) to recognize and evaluate non-constant entries during rule-query processing.

Meta-data is represented in four data relations. The schemes for these relations are independent of any particular DT system. They are an integral part of EXPRD and cannot be deleted. First, a relation contains data about DTs stored in the system. Each tuple in this relation represents one DT, and contains its name, number of conditions, number of actions, and its type. A DT can have one of two types: ambiguous or non-ambiguous. This relation acts as the top level for a rule base. Its contents are supplied by a user as an initial step in the definition of a DT system. During processing, EXPRD utilizes this information to recognize whether a DT exists in the system or not.

Secondly, a relation contains data about subjects used in DTs. Each tuple represents one subject, and contains the DT name in which this subject is used, the subject, and its type. A subject type is either condition or action. A special case, in which an action subject is used as a condition subject in another table, makes this subject appear twice with different types.

Thirdly, a relation contains data about the alternatives used to fill entries in DTs. Each tuple in this relation represents an alternative, and contains an alternative, its type, and the subject in which this alternative is used. A type of an alternative can be: constant, variable, function symbol, or relation symbol. EXPRD recognizes an alternative together with its subject, since an alternative may be used in more than one

table with different meanings. This relation is a key feature of the integrated system since it is the one used to differentiate constant entries from non-constant ones.

The fourth relation is an auxiliary one and used only for DT entries that are variable names. Each tuple in this relation represents a variable, and contains a variable name, its value, and the DT name (where the variable is used). This relation acts as a working memory for EXPRD.

4.1.2 EXDT: The Rule Base Component

EXDT is the component of EXPRD that stores the rule base. As discussed before, a rule base is represented as a set of DTs, each DT stored as a relation in EXDT. When a user wants to define a system of DTs, EXPRD directs him to provide information to fill the meta-data relations. Upon defining a DT name and its condition and action subjects, EXPRD creates a relation with a scheme:

$$\text{TABLENAME}(CS_1, ..., CS_n, AS_1, ..., AS_m)$$

where CS_i, AS_i are the condition and action subjects, n and m are the number of conditions and number of actions, respectively. EXPRD defines a key for this relation depending on the type (each DT has a type given by the user and stored in the meta-data). If a DT is given the type non-ambiguous (a function), then its key is the condition subjects. If a DT is given the type ambiguous, then the key is the condition and action subjects, i.e., an all-key relation.

When a user finishes defining all his DTs, a relation for each DT has been created and is ready to store decision rules. With addition of each decision rule, EXPRD checks entries in this rule against values defined previously (stored in the meta-data). If the rule is valid, EXPRD stores it in the relation representing the proper DT. If not, an error message is returned to the user.

4.2 Significance of the Integrated System

The integrated system generalizes a RDBS such that it can be used if the data items are not atomic. A key feature of such a system is that both data and rules are represented in a compatible form. All the information is stored in tabular forms. (Note that most of conventional systems represent rules in a form different from data representations.)

In addition to the advantages of its components, the integrated system acquires other advantages from the integration process. It provides a means to store and manipulate rules as well as data, in a uniform way. With rules and data stored in the same tabular form, communication between them is facilitated. Both rules and data are managed by the same management system, effecting many economies in the overall organization. Rules can be retrieved and updated as easy as data items and programs to check on missing or contradicting rules can be added as part of the management system or as application programs. Thus, procedural methods to construct a well structured set of rules in DT form can be developed by users.

When this research was proposed, no applications for the integrated system were proposed since the approach to the integration was meant to be an abstract one and not directed to any particular application. However, after the formal foundation was established and the integrated system was developed, they have been used in application studies to demonstrate relationships to current computer science activities. Two of these studies are included in the subsequent section. We can conclude that this system can be used in a wide variety of computer applications wherever databases, AI, and software engineering use rule representations together with databases to perform a task.

5. APPLICATIONS TO EXPERT SYSTEMS

During previous presentations of the concepts and process of integration, two issues dominated the discussions: 1) How does the formal basis, which the research has introduced for decision tables, apply to knowledge representation or rule-based systems? and 2) How are the integrated system and methodology related to current applications in expert systems? These issues motivated two studies, both of which appear in this dissertation. Complete copies of the papers appear in appendices D and E. We now overview them in section 5.1 and 5.2.

5.1 Theoretical Basis for Rule Systems

This study, done to support the assumption that a decision table system can be viewed as a general rule-based system, appears in a paper in Appendix D and is entitled "Rule-Based System: A Set-Theoretic Approach." A version of this paper has been presented at, and is published in the proceedings of, the Third Annual Computer Science Symposium on Knowledge-Based Systems: Theory and Applications, Columbia, SC, March 31-April 1, 1986.

The study is based on applying the set-theoretic approach, introduced in Appendix A for decision tables, to a rule-based system. Basic concepts of a rule-based system, such as a condition, a consequence, an action, or a conclusion, are thus viewed in a set-theoretic form. The concept of a decision situation, a collection of structured rules, is then introduced and characterized using the set-theoretic definitions of the basic concepts. A rule-based system is defined as a set of such decision situations.

The paper shows that when the set-theoretic approach is applied to view rule-based systems, some shortcomings are solved directly. For example, one deficiency in rule-based systems is that they do not have a theoretical size limit, which implies, hypothetically, that they could be infinite, or, practically, that they may grow to such a large size that they become incomprehensible and difficult to handle. The set-theoretic approach, simply, furnishes a theoretical size limit for a rule-base system. Also, the approach provides a foundation for automatic or semi-automatic handling of building rule-based systems. Even more, the approach formally characterizes several properties, such as redundancy, completeness, and ambiguity, which contribute to a formal basis for comparative analysis for rule systems.

5.2 Integration Methodology and Applied Expert Systems

This study aims to demonstrate that the integration methodology not only contributes to the theory of rule based systems, but also to applications in expert systems. A paper describing this study appears in Appendix E and is entitled: "A Reduction Methodology for a Differential Diagnosis Expert System." This paper has been presented at, and published in the proceedings of the Third Annual Computer Science Symposium on Knowledge Based Systems: Theory and Applications, Columbia, SC, March 31-April 1, 1986. It also has been selected to be considered for publication in the International Journal of Man-Machine Studies special issue edited by J. Bezdek.

The study applies the methodology developed through the system integration to increase expressibility and efficiency of an expert system. The developed methodology is characterized by three points, all of them unique in the integration approach: 1) it is based on structured rules, i.e., related rules that have the same structure are gathered in one table; 2) it employs a management system on the meta-level of the rules to

manipulate rules; and 3) it facilitates rule interaction with data relations and programs through the management system.

These principles are applied to provide a representation for an extended form of rules, which is particularly needed to express some differential diagnosis expert systems [79]. The extended form appears when a diagnostic decision situation is characterized by a set of symptoms, or observations, such that any subset of these symptoms establishes a diagnosis. Such rule may be expressed as:

$$\text{Any subset of k out of } \{C_1, C_2, ..., C_n\} \rightarrow \text{D}$$

where C_i, for $1 \leq i \leq n$, are a set of conditions, $k \leq n$, and D is a conclusion. In a differential diagnosis expert system, the conditions are symptoms or observations and the conclusion is a disease or class of diseases.

The redundant and inefficient breaking down of such rules to a set of simple rules is rejected in favor of this study's "reduction" method, which enables direct expression of such a rule. Management programs can then handle a reduction of such rules to check whether the diagnosis is established or not. The reduction methodology uses a set of decision tables and a reduction algorithm. Decision tables are stored in the integrated system and the reduction algorithm is embedded in the management system. The study shows that the reduction methodology, besides providing direct expression of an extended rule form, is more efficient than the naive approach in which a set of simple if-then rules is used.

6. CONCLUSIONS AND FUTURE PROSPECTS

The research establishes, in theory and practice, relationships between DTs, RDBS, and Prolog, utilizing previous work in the field when it is available and extending it when necessary. For DTs, CODASYL definitions have been generalized, properties have been formulated, and several representations have been investigated. Prolog implementations for basic DTs, as well as a logic programming view of many of their practices and conventions, have been discussed in detail and demonstrated as an approach to software development. These foundations are used in defining EXPRD, an integrated system, as well as in forming a methodology for expert system theory and application.

EXPRD is a prototype system which integrates a decision table system into a Prolog-RDBS environment. EXPRD stores, manipulates, and processes queries on a rule system as well as on data relations. Also, it automatically evaluates an action entry in a rule when it is expressed as a variable, a function symbol, or a call to another table. This opens a new horizon for rule systems: Rules, represented and stored in a structured form, can be processed, retrieved, updated, queried, and analyzed in an automatic fashion.

A methodology for rule-based systems has evolved from viewing rules through relational theory. Rules are viewed as symbolic structures managed by a supervising program. This view provides new insights into existing concepts and new notions on rule-based systems. The formal foundation, introduced here, facilitates characterization of the process of designing, building, and analyzing a rule-based system. Properties such as emptiness, completeness, size (domain), redundancy, and ambiguity are now

formulated and can be automatically evaluated. This furnishes a formal means for evaluation and comparison for rule-systems.

Comparing the integration approach and system to current efforts in expert systems demonstrate the strengthes of our approach in expressive power and mathematical foundations. For example, a recent system, R1-Soar [62], combines a domain-dependent knowledge-intensive rule system with a domain-independent problem-solving methodology to extend capabilities of a rule system. EXPRD combines Prolog (which is a domain-independent problem-solving methodology), Decision Tables (which represent a domain-dependent knowledge-intensive rule system), and a relational database system (which provides storage and manipulation of rule systems). Thus, besides being based on strong mathematical foundation, EXPRD has an additional data component.

Future research based on using EXPRD itself or the integration methodology can be projected. For EXPRD, its management programs can be extended, e.g., for constructing rules, analysis of an existing rule system, or more query capability. For RDBS, EXPRD and the integration methodology can be used to extend the management capabilities of a "deductive" database system, where rules of deduction can be stored in the same system. For Rule-Based systems, the integration approach developed a syntax for the rules, but their semantics still require investigation (although the concept of interpretation of non-constant entries represents a semantic property of a single rule). The formal foundation established in this research facilitates study and research, whether it be in databases, AI, rule engineering, expert systems, or software development.

BIBLIOGRAPHY

[1] K. Apt and M. VanEmden, "Contributions to the Theory of Logic Programming," JACM, Volume 29, Number 3, July 1982, pp 841-862.

[2] M. Bergman and H. Kanoui, "Application of Mechanical Theorem Proving to Symbolic Calculus," Third International Symposium on Advanced Computing Methods in Theoretical Physics, C.N.R.S., Marseilles, France, 1973.

[3] H. Berghel, "Simplified Integration of Prolog with RDBMS," Data Base, Volume 16, Number 3, Spring 1985, pp. 3-12.

[4] G. Boolos and R. Jeffrey, "Computability and Logic," N.Y.: Cambridge University Press, 1980.

[5] M. Bruynooghe, "Prolog in C for Unix Version 7: A Reference Manual," Belgium: Katholieke University 1980.

[6] A. Bundy et. al, "MECHO: A Program to Solve Mechanics Problems," DAI Working Paper Number 50, University of Edinburgh, 1979.

[7] H. Cantrell, J. King, and F. King, "Logic Structure Tables," Communications of ACM, Volume 4, Number 6, June 1961, pp. 272-275.

[8] C. Chang and R. Lee, "Symbolic Logic and Mechanical Theorem Proving," N.Y.: Academic Press, 1973.

[9] N. Chapin, "An Introduction to Decision Tables," DPMA Quarterly, Volume 3, Number 3, April 1967, pp. 2-23.

[10] N. Chapin, "Structured Analysis and Structured Design: An Overview," System Analysis and Design in the 1980's, N.Y.: North Holland, 1981.

[11] W. Clockson and C. Mellish, "Programming in Prolog," Berlin, Germany: Springer-Verlag, 1981.

[12] "A Modern Appraisal of Decision Tables," A CODASYL Report, N.Y.: ACM, July 1982.

[13] H. Coelho and L. Pereira, "The Dialectic Development of GEOM, a Prolog Geometry Theorem Prover," Department of A.I., University of Edinburgh, 1975.

[14] A. Colmerauer, "The Q-system to Formalize a Syntactic Analyzer for Ordinary Phrases," Dept. of Information Science, University of Montreal, 1973.

[15] E. Codd, "A Relational Model of Data for Large Shared Databanks," Communications of ACM, Volume 13, Number 6, June 1970, pp 377-387.

[16] E. Codd, "Relational Databases: A Practical Foundation for Productivity," Communications of ACM, Volume 25, Number 2, February 1982.

[17] C. Date, "An Introduction to Database Systems," 4th Edition, Addison Wesley Inc., CA, 1985.

[18] V. Dahl and R. Sambuc, "On The Use of First Order Logic for a Database for Natural Language Processing," Dept. of A.I., Marseille University, 1976.

[19] V. Dahl, "On Database System Development Through Logic," ACM Tran. Database Systems, Volume 7, Number 1, March 1982, pp 102-123.

[20] V. Dahl, "Logic Programming as a Representation of Knowledge," IEEE Computer, Volume C32, Number 10, October 1983, pp 106-111.

[21] R. Davis, "Runnable Specification as a Design Tool," in K. Clark and S. Tarnlund, Eds, "Logic Programming," N.Y.: Academic Press, 1982.

[22] R. Fagin, "Horn Clauses and Database Dependencies," JACM, Volume 29, Number 4, October 1982, pp 952-985.

[23] R. Fikes and T. Kehler, "The Role of Frame-Based Representation in Reasoning," Communications of ACM, Volume 28, Number 9, September 1985, pp 904-920.

[24] C. Forgy, "OPS5 User's Manual," Department of Computer Science, Carnegie-Mellon University, Pittsburgh, PA, 1981.

[25] I. Futo, F. Darvas, and P. Szeredi, "The Application of Prolog to the Development of QA and DBM Systems," Logic and Databases [26].

[26] H. Gallaire and J. Minker "Logic and Databases," N.Y.: Plenum Press, 1978.

[27] H. Gallaire, J. Minker, and J. Nicolas, "Logic and Databases: A Deductive Approach," ACM Computing Surveys, Volume 16, Number 2, June 1984, pp. 153-185.

[28] C. Gane and T. Sarson, "Structured Systems Analysis: Tools and Techniques," Englewood Cliff, N.J.: Prentice Hall, 1979.

[29] M. Genesreth and M. Ginsberg, "Logic Programming," Communications of ACM, Volume 28, Number 9, September 1985, pp 933-941.

[30] J. Grant and B. Jacob, "On the Family of Generalized Dependency Constraints," JACM, Volume 29, Number 4, October 1982, pp 986-997.

[31] F. Hayes-Roth, "Rule-Based Systems," Communications of ACM, Volume 28, Number 9, September 1985, pp 921-932.

[32] P. Horstmann, "Expert Systems and Logic Programming for CAD," VLSI DESIGN, November 1983, pp 37-46.

[33] C. Hogger, "Derivation of Logic Programming," JACM, Volume 28, Number 2, April 1981, pp 372-392.

[34] E. Humby, "Programs from Decision Tables," N.Y.: Macdonald, 1973.

[35] R. Hurley, "Decision Tables in Software Engineering," N.Y.: Von Nostrand Reinhold Company Inc., 1983.

[36] B. Jacob, "On Database Logic," JACM, Volume 29, Number 2, April 1982, pp 310-332.

[37] R. Kowalski, "A Proof Procedure Using Connection Graphs," JACM, Volume 22, Number 3, June 1974, pp. 572-595.

[38] R. Kowalski, "Algorithms= Logic+Control," Communications of ACM, Volume 22, Number 7, 1979, pp 424-436.

[39] R. Kowalski, "Logic for Data Description," in Logic and Databases [26].

[40] R. Kowalski, "The Limitations of Logic," Proc. 1986 ACM Computer Science Conference Cincinnati, February 1986, pp. 7-13.

[41] R. Kowalski, "Logic for Problem Solving," Artificial Intelligence series, The Computer Science Library, N.Y.: Elsevier North Holland Inc., 1979.

[42] A. Lew, "Decision Tables for General-purpose Scientific Programming," Software Practice and Experience, Volume 13, September, 1981, pp 181-188.

[43] A. Lew, "On the Emulation of Flowchart by Decision Tables," Communications of ACM, Volume 25, Number 12, December 1982, pp 895-905.

[44] K. London, "Decision Tables," N.Y.: D. Van Nostrand Co. Inc, 1972.

[45] R. Maes, "On the Representation of Program Structure by Decision Tables: a Critical Assessment," The Computer Journal, Volume 21, Number 4, November 1978, pp. 290-295.

[46] Z. Markusz, "How to Design Variants of Flats Using the Programming Language PROLOG Based on Mathematical Logic," Proc. IFIP, Amesterdam: North Holland, 1977, pp. 885-889.

[47] H. McDaniel, "An Introduction to Decision Logic Tables," N.Y.: John Wiley and Sons Inc., 1968.

[48] D. McDermott, "The Prolog Phenomenon," SIGART Newsletter, July 1980.

[49] B. McMullen, "Structured Decision Tables," SIGMOD Notices, Volume 19, Number 4, April 1984, pp 34-43.

[50] J. Metzner and B. Barnes, "Decision Tables Language and Systems," N.Y.: Academic Press, 1977.

[51] M. Montalbano, "Decision Tables," Palo Alto CA: Science Research Associates, 1974.

[52] B. E. Moret, "Decision Trees and Diagrams," ACM Computing Surveys, Volume 14, Number 4, December 1982, pp. 593-624.

[53] R. Nickerson, "An Engineering Application of Logic-Structure Tables," Communications of ACM, Volume 4, Number 11, November 1961, pp. 516-520.

[54] H. Parde, "A Computational Approach to Approximate and Plausible Reasoning with Applications to Expert Systems," IEEE Trans. on Pattern Analysis and Machine Intelligence, Vol PAMI-7, Number 3, May 1985, pp. 260-283.

[55] F. Pereira, "C-Prolog User's Manual," Dept. of Architecture, University of Edinburgh, December 1983.

[56] S. Pollack, H. Hicks, and W. Harrison, "Decision Tables: Theory and Practice," N.Y.: J. Wiley, 1971.

[57] R. Pressman, "Software Engineering: A Practitioner's Approach," McGraw-Hill Software Engineering and Technology Series, 1982.

[58] K. Reilly, J. Barrett, and A. Salah, "A First Study on Prolog-Mimicking Barrel Extended Entry Decision Table Presentation System," Technical Report, UAB, C&IS, July 30, 1984.

[59] K. Reilly, W. Jones, J. Barrett, A. Salah, E. Strand, J Autry, and P. Rowe, "Software Development Studies A Simulation Environment Perspective," Information Systems Engineering Tools & Technology (ISETT 1984), Ann Arbor, Michigan, August 1984, 16 pp.

[60] K. Reilly, A. Salah, P. Morgan, and P. Rowe, "Multiple Representation in a Language Driven Memory Model," Papers on Computational and Cognitive Science, edited by Edwin Battistella, Published by Indiana University Linguistic Club, August 1984, pp 96-111.

[61] K. Reilly, A. Salah, and C.C. Yang, "A Logic Programming Perspective on Decision Table Theory and Practice," To be submitted, June 1986, 30 pp.

[62] P. Rosenbloom, J. Laird, J. McDermott, A. Newell, and E. Orciuch, "R1-Soar: An Experiment in Knowledge-Intensive Programming in Problem-Solving Architecture," IEEE Trans. on Pattern Analysis and Machine Intelligence, Vol PAMI-7, Number 5, September 1985, pp. 561-569.

[63] P. Roussel, "Prolog: Reference Manual," Dept. of A.I., Universiteit d'Aix-Marseille, Luminy, September 1975.

[64] A. Salah, "An Extension of Prolog with a Database Option." In [85], pp 247-250.

[65] A. Salah and K. Reilly "A Reduction Methodology for a Differential Diagnosis Expert System," Proceedings of the Third Annual Computer Science Symposium on Knowledge-Based Systems: Theory and Application, March-April 1986.

[66] A. Salah and K. Reilly "A Study on Using OPS5 for Decision Table Representation and Implementation," Technical Report, UAB, C&IS, July 1985.

[67] A. Salah, K. Reilly, and C. C. Yang, "A Logic Programming Approach to Decision Table Definition and Implementation," Presented in the ACM Midsoutheast Conference in Gatlinburg, TN, November 1984.

[68] A. Salah, K. Reilly, and C. C. Yang, "A Restatement of Decision Table Problem in Logic Programming," Technical Report, UAB, C&IS, December 1984.

[69] A. Salah, K. Reilly, and C.C Yang, "An Integration of a Rule Representation into A Relational Database System," Technical Report, UAB, C&IS, Submitted to ACM/IEEE Computer Society 1986 Fall Joint Conference, March 1986.

[70] A. Salah and C.C.Yang "Rule-Based Systems: A Set-Theoretic Approach," Proceedings of the Third Annual Computer Science Symposium on Knowledge-Based Systems: Theory and Application, March-April 1986.

[71] A. Salah, C. C. Yang, and K. Reilly, "On Decision Table Properties, Representations, and Implementations," Submitted to IEEE Trans. on Software Engineering, June 1985.

[72] Y. Sagiv, C. Delobel, D. Parker Jr, and R. Fagin, "An Equivalence Between Relational Database Dependencies and a Fragment of Propositional Logic," JACM, Volume 28, Number 3, July 1981, pp 435-453.

[73] J. Ullman, "Principles of Database Systems," MD: Computer Science Press, Inc., 1983.

[74] J. Ullman, "Implementation of Logical Query Languages for Databases," Proceedings of 1985 ACM-SIGMOD Conference, June 1985.

[75] "UNH Prolog User Manual," N.H.: University of New Hampshire, July, 1983.

[76] M. VanEmden and R. Kowaliski, "The Semantics of Predicate Logic as a Programming Language," JACM, Volume 23, Number 4, October 1976, pp 733-742.

[77] D. Warren, "WARPLAN: A System for Generating Plans," DCL Memo 76, Dept. of A.I., University of Edinburgh, 1974.

[78] D. Warren, L. Pereira, and F. Pereira, "Prolog- The Language and its Implementation Compared with LISP," Proc. Symp. on A.I. and Programming Languages, SIGPLAN Notices, Volume 12, Number 8, Aug. 1977.

[79] S. Weiss and C. Kulikowski, "A Practical Guide to Designing Exert Systems," Rowman & Allanheld Publishers, 1984.

[80] R. Welham, "Geometry Problem Solving," DAI Research Report Number 14, University of Edinburgh, 1976.

[81] R. Welland, "Decision Tables and Computer Programming," U.K.: Heyden & Son Ltd, 1981.

[82] G. Wiederhold, "Database Design," Second edition, N.Y.: McGraw-Hill, 1983.

[83] Chao-Chih Yang, "Associative Memory Systems and Their Applications to Picture and Arithmetic Processings," Ph.D. Dissertation, Northwestern University, Evanston, Ill., 1966.

[84] Chao-Chih Yang, "Relational Database Design," Lecture Notes, UAB, 1984.

[85] Chao-Chih Yang, "Relational Databases," Englewood Cliff, N.J.: Prentice-Hall Inc., 1986.

[86] S. S. Yau and H. S. Fung, "Associative Processor Architecture - A Survey," ACM Computing Surveys, Volume 9, Number 1, March 1977, pp. 3-28.

APPENDIX A

On Decision Table Properties, Representations, and Implementations

Akram Salah, Chao-Chih Yang, and Kevin Reilly

Since RDBS and Prolog have strong theoretical basis, a formal statement of DTs is essential to build an integration on a solid theoretical foundation, rather than on a special case. This paper includes a statement of DTs and their properties in a formal way, so that it would facilitate an integration of them into the complete system. There were no widely accepted formal definition of DTs until CODASYL reported one in 1982. This paper generalizes the CODASYL definitions.

In the paper, a set-theoretic definition for DTs is modified and generalized. Properties such as emptiness, completeness, ambiguity, and redundancy, which may be used to evaluate a DT, are discussed and formulated in a concise formal way. Also, the notion of DT query processing is introduced and characterized, based on five policies which can be used as grounds for comparing DT implementations. Finally, representations of a DT as a set of rules and as a relational database table are discussed and evaluated based on query processing policies.

This paper is a prelude to the integration (see Appendix C). Also, it is used as basis to provide a set-theoretic definition for a Rule-Based system (see Appendix E).

This paper has been submitted for publication and acceptance is pending.

On Decision Table Properties, Representations, and Implementations

Akram Salah, Chao-Chih Yang, and Kevin Reilly

ABSTRACT -Formal definitions of decision tables based on the CODASYL set-theoretic triple [4] are modified and generalized as a prelude to discussion of various DT formats and properties. These latter comprise an extensive list include some new perspectives into conventional properties as well as new notions. The activity of query processing for decision tables is presented in general and comprehensive fashion, consistent with these perspectives and with a goal of compatible programming of DTs using logic and functional programmings. Decision tables are examined in relation to functional descriptions, relational databases, and the use of logic programming in Prolog and an embedded Prolog system (embedded within a relational database system). Applications, though not discussed at length, are mentioned, and possible topics for further research are proposed.

Index Terms-Associative memory, decision table, embedded decision table, Horn clause, LISP, Prolog, relational database, UNIX

1 Introduction

The decision table (DT) has been an object of research and development for almost three decades. Recent activity reveals an increasing attempt at more precision in dealing with DT fundamentals and in exploiting modern representational and implementation strategies [4, 5, 9, 10, 18]. This paper follows suit in providing precise definitions, extending the report [4], and proposing implementations using logic and functional programming. The approach adjoins to DT theory and practice such terminology as Horn clauses, conditionals, predicates, functional style, and pattern matching.

After an informal characterization of a DT, we present a formal definition based on the CODASYL set-theoretic triple. We then modify and generalize this definition prior to examining DT formats and properties such as: don't care, (non-)ambiguity, completeness, the empty table, naming, order within tables, (non-)redundancy, equivalent tables, and types of entry. Included also are comments on systems of tables.

Under query processing, we present five basic approaches. Lesser known but useful queries directed against actions and ones directed against combinations of conditions and actions comple-

ment the usual processing against conditions. When "normal" query processing is viewed in this extended fashion, some new insights accrue.

We discuss DTs through perspectives of Horn clauses, functional descriptions, and relational databases. Prototype implementations of representations are provided in Prolog to illustrate underlying potential for knowledge representations applicable to question answering, decision making, and expert systems [17]. The prototypes demonstrate attractive features of Prolog for our purposes: its nondeterministic nature and input-output indifference. We can provide multiple answers to (query) transactions without requiring much additional coding effort and conveniently implement the basic query approaches for the single-table and most table systems cases. A version of Prolog conjoined to a relational database system is used to illustrate a mode of processing in which redundancies are controlled automatically.

Possible topics for further research are addressed as they arise in the text and in a discussion section. The focus for future work lies in don't-care entries and incomplete information, reduction of redundancies, data compression of entries, systematic design of DT systems of related tables, integrity control including updates, and tabular query optimization.

2 Definitions of a Decision Table

We first present a graphical sketch of a decision table and then, a formal definition.

2.1 A graphical sketch of a DT

A DT is graphically represented by a table, as shown in table 1, which is partitioned into four quadrants by a pair of crossed doubled lines.

Table 1. A DT in a vertical format

Condition stub quadrant	Condition entry quadrant
Action stub quadrant	Action entry quadrant

The ***condition stub*** quadrant includes a number of condition subjects.

The ***condition entry*** quadrant has a number of entries to be filled by symbols, known as condition alternatives.

The *action stub* quadrant includes a number of action subjects.

The *action entry* quadrant has a number of entries to be filled by symbols, known as action alternatives.

Condition and action subjects and their alternatives are defined in a subsequent definition along with don't care entries and embedded DTs. The condition and action subjects and their alternatives are used to provide *decision rules* (DR) or, simply, *rules,* called *argument—image* or *argument—value* pairs in this paper. In the literature, pairs of question-answer, test-result, stimulus-response, antecedent-consequent, query-response, transaction-activity, body-head, or input-output are also used, but these terms may not have the same meaning in every instance.

Table 1 is in a *vertical format*: the subjects and the rules of a DT are located in columns of the table. When subjects and rules are located in rows, the DT is in a *horizontal format*. Both formats are equivalent in information content, since one of them can be transformed into the other by the well-known transposition operation of matrix algebra.

2.2 A formal definition of a DT

A DT has been formally defined in a CODASYL report [4]. This section proposes a modified and generalized version. A *DT* is defined as a set-theoretic triple (C, A, R) where C and A are finite, nonempty sets and R is a relation. The elements C, A, and R in the triple are defined in the following.

(1) The *condition set* $C = \{C_1, C_2, ..., C_n\}$ consists of n *conditions* for $n \geq 1$. A condition C_i is denoted by an ordered pair consisting of a *condition subject,* CS_i, and a finite, nonempty set $CA_i = \{CA_{i1}, CA_{i2}, ..., CA_{is_i}\}$ where each CA_{ij} for $1 \leq j \leq s_i$ is a *condition alternative.*

(2) The *action set* $A = \{A_1, A_2, ..., A_m\}$ consists of m *actions* for $m \geq 1$. An action A_i is denoted by an ordered pair consisting of an *action subject,* AS_i, and a finite, nonempty set $AA_i = \{AA_{i1}, AA_{i2}, ..., AA_{it_i}\}$ where each AA_{ik} for $1 \leq k \leq t_i$ is an *action alternative.*

(3) The *relation* R is a subset of the Cartesian product of "cspace" and "aspace;" "cspace," the *condition space*, is defined by the complex product (denoted by the symbol *) [20] of the n sets of condition alternatives, i.e., $cspace = * (CA_1, CA_2, ..., CA_n) = "\{"CA_{1j_1} \ CA_{2j_2} ...$

$CA_{nj_n} \mid 1 \leq j_x \leq s_x$ and $1 \leq x \leq n$"}"; "aspace," the *action space* is defined by the complex product of the m sets of action alternatives, i.e., aspace = * (AA_1, AA_2, ..., AA_m) = "{"AA_{1k_1} AA_{2k_2} ... $AA_{mk_m} \mid 1 \leq k_y \leq t_y$ and $1 \leq y \leq m$"}"; and the Cartesian product of "cspace" and "aspace" is equal to "{"(CA_{1j_1} ... CA_{nj_n}, AA_{1k_1} ... AA_{mk_m})"}".

Note that for unifying the notation, complex products and strings are used for both conditions and actions since they are conventionally used in relational databases and, later in this paper, when a DT is viewed as a database relation. Each element of R, a DR, is represented by an ordered pair of a string $CA_{1j_1} \cdots CA_{nj_n}$ of length n and a string $AA_{1k_1} \cdots AA_{mk_m}$ of length m. The number of all strings in "cspace" and that of all strings in "aspace" are, respectively, $|\text{cspace}| = s_1 \cdot s_2 \cdot \cdots \cdot s_n$ & $|\text{aspace}| = t_1 \cdot t_2 \cdot \cdots \cdot t_m$, where the symbol "." stands for multiplication.

Since a predicate also has arguments, we use the terms: DR argument, relation argument, function argument, and predicate argument for proper differentiation.

2.3 Tabular representation of a DT

In a DT with a vertical format, the row headings, located in the condition and action stub quadrants as in table 1, are labeled by the n condition and m action subjects, respectively. These headings, located in the left-most column, constitute the skeleton of a DT. Each column in the condition and action entry quadrants of a DT is filled by a string of n condition alternatives and a string of m action alternatives. These alternatives are usually constants. Variables and relations (including functions as a special case) can be also used as alternatives. The latter case arises when a DT has other embedded DTs. A DT is in an *extended entry format* if its quadrants are filled by subjects and alternatives, based on the above method; and augmentation of new alternatives into, deletion of existing alternatives from, or replacement of existing alternatives (by other symbols) from it is permitted. In addition to the vertical, horizontal, and extended entry formats, there are other variations. The definitions of cases known as limited and mixed entry formats are postponed until some later.

3 Properties of Decision Tables

For description convenience, each DR in a relation R is denoted by an ordered pair (c, a), the first member c representing a string of n condition alternatives, i.e., $c = CA_{1j_1}\ CA_{2j_2}\ \cdots\ CA_{nj_n}$ and the second member a representing a string of m action alternatives, i.e., $a = AA_{1k_1}\ AA_{2k_2}\ \cdots\ AA_{mk_m}$.

Note that an argument c or image a in a DT must be an element in "cspace" or "aspace," respectively. However, some elements in "cspace" or "aspace" are not necessarily an argument or image in a DT.

In the following numbered paragraphs we overview some properties of DTs, using terminology based on the above definitions and looking ahead to what we called "compatible implementations." Even though many of these properties are familiar, they have not been characterized in this way before (even by CODASYL).

(1) As described previously, a DT can have as alternatives, constants, variables (representing incomplete information), or relations (representing embedded DTs). Some condition entries in a DT can be filled by a symbol known as a *don't-care*, denoted by a constant, such as an underscore. The "don't-care" is not an alternative per se; it defines a *composite DR* obtained by merging s_i DRs if these DRs involve all s_i condition alternatives CA_{ij} through CA_{is_i} and have a common image. Note that use of relations as alternatives has not been a common practice but is compatible with certain modern implementations.

(2) R in the triple (C, A, R) of a DT is either a many-to-many relation in the most general case or a many-to-one (including one-to-one) function in the special case. A DT is *non-ambiguous* if R is a function and is *ambiguous* [9, 10] otherwise. In the literature, some writers use "consistency" and "inconsistency" when "nonambiguity" and "ambiguity," seem preferable. We prefer the latter two terms. A relation R is a function if each argument c of the relation has a unique image a. On the other hand, if at least two DRs have identical argument and different images, then the decisions to be taken are ambiguous. Non-ambiguous DTs form a proper subset of DTs

and are compatible with non-AI languages, such as FORTRAN, PL/I, and COBOL for their implementations [5]. These non-ambiguous DTs and their compatible implementations have most commonly practiced.

When a DT is ambiguous because of the existence of nonunique images of some argument, we can modify the DT by adding a condition subject, called DECISION_OPTION, together with its alternatives indicating the priorities of the corresponding DR images for the same original DR argument based on the likelihood of the actions being taken. In other words, an original DR argument that has k images, for $k > 1$, is modified into k DR arguments each having a unique image after the new subject DECISION_OPTION and its k alternatives 1 through k indicating k options are added to the DT. By this approach, the first option means the most likely decision to be taken, and so forth. By this modification, the ambiguous problem is then solved.

(3) A DT (C, A, R) is *complete* [4, 6, 19] if the domain of R is equal to the condition space "cspace", i.e., Domain(R) = cspace, and is *incomplete* otherwise where the domain of R is the set of all arguments of R. The completeness of an originally incomplete DT can be resolved depending on the following possibilities.

(i) When only one argument is missing from a DT, the completeness of the DT can be easily accomplished by adding a DR whose argument is the missing argument and whose image is an appropriate action to be taken.

(ii) When at least two arguments are missing from a DT, T_1, the case becomes more complicated. We consider only two extreme cases here.

  (a) If each missing argument is a physically possible string of n condition alternatives (i.e., no conflict among the n alternatives) and all such missing arguments could have a common image, then we add a single composite DR to T_1, which is called an *ELSE rule* whose argument is denoted by ELSE and whose image is the common image of the missing arguments.

  (b) When each missing argument is a physically impossible string of n condition alternatives because of the existence of a conflict among the alternatives or of no practical significance, but some of them could occur by accident or malfunction, the ELSE rule

becomes an error rule so that its image might be an error message.

Note that an ELSE rule defines, corresponds to, or is equivalent to a non-ambiguous DT in which the arguments have a unique image. Note also that not every DT is required to be complete since its completeness may not have any practical significance.

(4) A DT (C, A, R) is *empty* if R is the empty set. An empty DT corresponds to the skeleton of a DT, i.e., all subjects are known but entries are not filled by symbols. This notion is important when a DT is viewed as a relation in a relational database: the corresponding empty DT defines the relation scheme [20].

(5) A DT does not need a name. It can be identified by its triple. DTs are given names only for convenience, e.g., in systems of tables.

(6) Since a relation R is a set of DRs, the ordering of DRs is immaterial. When the columns representing DRs are permuted, information content is preserved, provided, of course, that the argument-image pair (c, a) of each DR is located in the same column.

(7) A DT is called *nonredundant* if there exist no duplicated DRs and is called *redundant* otherwise. A redundant DT may exist, since an inconsistent update of some DR may cause duplicates. Since a relation R is a set of DRs, nonredundancy is to be enforced. Automatic enforcement of nonredundancy is a significant feature of a relational database view of a DT (covered below).

(8) Given a DT, when all of its condition rows and all of its action rows are separately permuted, the DT and the permuted one are equivalent, since such a permutation preserves information content. Note that ordering of the condition rows does affect processing efficiency, especially when don't-cares are included in the DT [5].

(9) A DT, T_1, in an extended entry format without including an ELSE rule can be transformed into an equivalent DT, T_2, in a *limited entry format* by redefining subjects, alternatives, and DRs as follows.

(i) Each condition subject in T_2 is defined by combining a condition subject CS_i and one of its alternatives CA_{ij_i} in T_1 and written as $CS_i : CA_{ij_i}$ for some i and some j_i, $1 \leq j_i \leq s_i$ and $1 \leq i \leq n$. Then the number of condition subjects in T_2 is $n' = \sum_{j=1}^{n} s_j$. Obviously,

n' is greater than n, unless each $s_j = 1$ for $1 \leq j \leq n$. Hence, the "cspace" of a DT in an extended entry format and that of its equivalent DT in a limited entry format may not have an equal number of arguments. When the former "cspace" has fewer arguments and the former DT is complete, the latter DT is not complete.

(ii) Similarly, each action subject in T_2 is $CA_i : AA_{ik_i}$ for some i and some k_i, $1 \leq k_i \leq t_i$ and $1 \leq i \leq m$. Then the number of action subjects in T_2 is $m' = \sum_{j=1}^{m} t_j$. Similarly, m' is greater than m, unless each $t_j = 1$ for $1 \leq j \leq m$.

(iii) A DR containing no don't-care entries in T_1 is transformed into an ordered pair of a string of n' condition alternatives and a string of m' action alternatives in T_2 such that the i-th element in each such string becomes "yes" (abbreviated as "y" or "1"), "do," "check mark," or "y!" meaning "yes by implication" if the corresponding subject is relevant; and becomes "no" (abbreviated as "n" or "0"), "don't-do," "blank," or "n!" meaning "no by implication" otherwise. These symbols are called *primitive*.

If a condition entry corresponding to a condition subject CS_i in T_1 is filled by a "don't-care," then all condition entries corresponding to $CS_i : CA_{ij_i}$ for each i, $1 \leq i \leq s_i$, are filled by "don't-cares." When the condition space of T_1 has fewer arguments and T_1 has an ELSE rule, the ELSE rule cannot be transformed into a single ELSE rule in T_2 since some DRs covered in the ELSE rule of T_2 do not have any counterparts covered in the ELSE rule of T_1 although each DR covered in the ELSE rule of T_1 has a counterpart covered in the ELSE rule of of T_2 by transformation. In this case, we can only transform the DRs covered in the ELSE rule of T_1 into equivalent DRs in T_2 without creating a single ELSE rule in T_2.

A DT is in a *mixed entry format* if it is neither in an extended entry format alone nor in a limited entry format alone. A DT in a mixed entry format can be transformed into a DT in a limited entry format by transforming only the rows filled by nonprimitive symbols.

4 Decision Table Query Processing Policies

We define a *transaction* as a sequence of specific alternatives given in a query and used as input to a DT to retrieve, or to search for, relevant information. The result of a query may be a single or multiple outputs. There are five types of DT query processing policies defined on these transactions.

(1) *Condition policy map* [10] uses a DR argument as a transaction. It retrieves each DR image whose DR argument matches the transaction.

(2) *Action policy map* [10] uses a DR image as a transaction. It retrieves each DR argument whose DR image matches the transaction.

(3) *Condition entry map* uses at least one condition alternative in some DR argument as a transaction. It retrieves each DR image whose DR argument contains the transaction.

(4) *Action entry map* uses at least one action alternative in some DR image as a transaction. It retrieves each DR argument whose DR image contains the transaction.

(5) *Hybrid* (condition and action) *entry map* uses at least one condition and at least one action alternatives contained in some DR as a transaction. It retrieves all DRs matching the transaction.

Policy (3) or (4) is a generalization of (1) or (2), respectively. Policy (5) further generalizes (3) and (4). When a transaction is irrelevant to a DT to be processed, there would be no match and consequently, no output during retrieval. These policies are based on the consideration by which DTs are ambiguous. Hence, each transaction may have multiple outputs unless the underlying DT is non-ambiguous and the policy is of type (1).

5 Representations and Their Compatible Implementations of Decision Tables

In addition to the set-theoretic (i.e., triple (C, A, R)) and tabular representations in various formats, a DT has been represented by a function [6], decision tree or diagram, and a Chapin chart [4, 11, 6]. In this section, we develop new representations of a DT by a set of Horn clauses in general and a database relation in particular. Implementations compatible with Horn clauses and

database relations are examined and included in appendices 1 and 2. Those implementations can accomplish the most general type (5) of transaction without requiring very complicated codes.

5.1 Viewing a DT as a Set of Horn Clauses and Implementing it by a Prolog Program

For each DR (c, a) in a relation R, we define a Horn clause [7, 20] where c and a abbreviate a string of n condition alternatives and of a string of m action alternatives, respectively.

(1) A ***condition predicate*** for a DR argument c is defined as an n-ary predicate whose name is arbitrarily chosen to be CONDITION and whose predicate arguments are the n elements of c, i.e., CONDITION(CA_{1j_1}, CA_{2j_2}, ..., CA_{nj_n}) where CA_{ij_i} is the i-th element of the DR argument c. Let $CS_i(CA_{ij_i})$ be a unary predicate whose name is the i-th condition subject CS_i and whose predicate argument is the ij_i-th condition alternative CA_{ij_i}. Then the following headed Horn clause $CONDITION(CA_{1j_1}, ..., CA_{nj_n}) \leftarrow CS_1(CA_{1j_1}), ..., CS_n(CA_{nj_n})$ defines the condition predicate by means of the n conjuncted unary predicates $CS_1(CA_{1j_1})$ through $CS_n(CA_{nj_n})$ where "," on the right side of $\leftarrow$ stands for conjunction and the symbol $\leftarrow$ is read as "if." The left-hand side of $\leftarrow$ is the ***head*** or ***conclusion*** of the clause, and the right-hand side is the ***body*** or ***(joint) conditions*** of the clause.

(2) Similarly to a condition predicate, we define an ***action predicate*** $ACTION(AA_{1k_1}, ..., AA_{mk_m})$ such that the corresponding headed Horn clause is $ACTION(AA_{1k_1}, ..., AA_{mk_m}) \leftarrow AS_1(AA_{1k_1}), ..., AS_m(AA_{mk_m})$ for a DR image a, where AA_{ik_i} is the i-th element of the DR image a, and $AS_i(AA_{ik_i})$ is a unary predicate whose name is the i-th action subject AS_i and whose predicate argument is the ik_i-th action alternative AA_{ik_i}.

(3) Each argument-image pair (c, a) in the relation R can be represented by a headed Horn clause of the following form

$$CONDITION(CA_{1j_1}, ..., CA_{nj_n}) \leftarrow ACTION(AA_{1k_1}, ..., AA_{mk_m}); \text{ or}$$

$$CONDITION(CA_{1j_1}, ..., CA_{nj_n}) \leftarrow AS_1(AA_{1k_1}), ..., AS_m(AA_{mk_m}).$$

These clauses are readily implemented in Prolog [16] and perform a condition policy map where the goal is a headless Horn clause representing a DR argument. The case in which multiple DR

images correspond to the goal occurs when R is not a function. In this case, the term "map" does not mean a function. On the other hand, for performing an action policy map where the goal is a headless Horn clause representing a DR image, a Prolog program is based on the following headed Horn clause by interchanging the body and the head of any of the previous clauses, i.e.,

$$\text{ACTION}(AA_{1k_1}, ..., AA_{mk_m}) \leftarrow \text{CONDITION}(CA_{1j_1}, ..., CA_{nj_n}); \text{ or}$$

$$\text{ACTION}(AA_{1k_1}, ..., AA_{mk_m}) \leftarrow CS_1(CA_{1j_1}), ..., CS_n(CA_{nj_n}).$$

Thus, a DT is represented by a set of Horn clauses headed by a condition predicateas for providing a condition policy map or by a set of Horn clauses headed by an action predicate for providing an action policy map. An implementation based on Horn clauses using Prolog has the advantage of providing flexibility in updating a DT. In addition, each Horn clause may have a different number of predicates in the body. This allows adding new predicates into, or deleting existing predicates from, the body of a clause without changing the whole program. This representation is especially compatible with Prolog implementation concerning a system of related DTs in which embedded DTs are included. A Prolog implementation of a system of two related DTs (i.e., tables 2 and 3), and implementing only a condition policy map, is shown in appendix A-1.

5.2 Viewing a DT as a Relation in a Relational Database System

The data layout of a DT in a horizontal format and that of the conventional relation in a relational database [20] are similar. When all the entries in a DT are constants, then the relational database notions apply directly. If, however, a DT has (an) embedded DT(s), then we treat the embedded DT as a character string that is considered as a constant for purposes of some forms of processing. When such a constant is included in a result of a query, the corresponding embedded DT needs a subsequent stage of processing. Without further extension of a relational data model, full processing of an embedded DT is impossible. Thus, this representation is compatible with Prolog implementation for processing single DT.

The set of condition and action subjects defines the set U of *attributes* or *column names* of a relation in a relational database, i.e., $U = \{CS_1, ..., CS_n, AS_1, ..., AS_m\}$. Thus, each subject corresponds to an attribute and is used to label a column heading of the DT. Thus, the *relation scheme* is the set U and the arity of the relation is n + m. Note that the relation scheme U corresponds to the empty DT whose column headings are labeled by the elements in U. For each subject, the set of alternatives defines the *active domain* [20] of the subject.

Each DR (c, a) of R defines a *tuple* of the following string form: $CA_{1j_1} \cdots CA_{nj_n} AA_{1k_1} \cdots AA_{mk_m}$ belonging to a relation over scheme U. Thus, a set of DRs in their string forms defines a relation in a relational database.

If a DT is non-ambiguous (i.e., R is a function), then the set of condition subjects is the *primary key* or, simply, the key. On the other hand, if a DT is ambiguous (i.e., R is not a function), then we need to choose more attributes from the set of action subjects to define the key. Note that each entry, corresponding to a prime attribute (i.e., an attribute in the primary key), cannot have a null value. This approach can accomplish all five types of DT query processing policies without requiring any additional coding effort. A Prolog implementation of a single DT (table 4) is shown in appendix A-2.

5.3 Storing DTs in an Associative Memory and Processing DT Queries by an Associative Processor

For achieving concurrent comparisons between a transaction and the content of a DT, the table can be stored in an associative memory [19, 21]. In this case, a transaction can be any of the five types described in section 4. Multiple results are yielded by a multiple matching. However, this implementation is rather expensive and may put an arbitrary limit on the size of a DT.

6 Discussion

In this paper, we have modified and generalized the CODASYL definition of a DT, examined DT formats, and formally restated several properties of DTs in light of these definitions. The matter of query processing is treated in a general fashion consistent with the new properties and insights

of the approach. We have provided other new representations based on Horn clause and relational databases. These representations are compatibly implemented by Prolog and also LISP programs (LISP programs are not included in this paper). These representations and their compatible implementations are important topics in the general area called knowledge representation and may have applications in designing question answering, decision making, and expert systems.

Among the implementations, we used the AI (artificial intelligence) programming languages LISP and Prolog which involve functions, conditionals, predicates, Horn clauses, pattern matchings, etc. for symbol manipulation where Prolog programs are included in the appendices. The nondeterministic nature of Prolog provides for multiple answers to the same transaction without requiring additional coding effort. Prolog is also easy-to-learn and easy-to-use.

For processing a single DT using the Prolog database option [1, 20], the arguments of a single predicate representing the database relation of a DT have no input/output role distinction (i.e., "input-output indifference") so that the same predicate can be queried by any of the five types of transactions described in section 4. In addition, it is trivial to print a whole DT in response to a query. Nonredundancy is always maintained since duplicates are automatically prohibited or eliminated, when tuples are created or updated in a database relation. However, the database approach without further extensions is incapable of automatically handling embedded DTs or processing a system of related DTs.

Possible topics for further research include proper treatment of don't-care entries and incomplete information (including null and currently unknown entries), reduction of redundant entries in addition to use of the ELSE rule, systematic design of a system of related DTs including embedded ones, data compression of entries, integrity control including updates, tabular query optimization, applications of DTs in real world systems, and automatic processing of a system of related DTs when they are viewed as a database.

7 REFERENCES

[1] M. Bruynooghe, "Prolog in C for Unix Version 7: A Reference Manual," Belgium: Katholieke Univ. 1980.

[2] H. Cantrell, J. King, and F. King, "Logic Structure Tables," Communications of ACM, Volume 4, Number 6, pp. 272-275, June, 1960.

[3] W. Clockson and C. Mellish, Programming in Prolog, Berlin, Germany: Springer-Verlag, 1981.

[4] "A Modern Appraisal of Decision Tables" A CODASYL Report, N.Y.: ACM, July, 1982.

[5] R. Hurley, Decision Tables in Software Engineering, N.Y.: Von Nostrand Reinhold company Inc., 1983.

[6] R. Kowalski, Logic for Problem Solving, Artificial intelligence series, The Computer Science Library, N.Y.: Elsevier North Holland Inc., 1979.

[7] K. London, Decision Tables, N.Y.: D. Van Nostrand Co. Inc, 1972.

[8] J. Metzner and B. Barnes, Decision Tables Language and Systems, N.Y.: Academic Press, 1977.

[9] M. Montalbano, Decision Tables, Palo Alto CA: Science Research Associates, 1974.

[10] B. M. E. Moret, "Decision Trees and Diagrams," ACM Computing Surveys, Volume 14, Number 4, pp. 593-624, Dec., 1982.

[11] F. Pereira, "C-Prolog User's Manual," Dept. of Architecture, University of Edinburgh, Dec., 1983.

[12] P. Roussel, "Prolog: Reference Manual," Dept. of A.I., Universiteit d'Aix-Marseille, Luminy, Sep, 1975.

[13] K. Reilly, A. Salah, P. Morgan, and P. Rowe, "Multiple Representation in a Language Driven Memory Model," Proceeding of the Conference on Linguistics in Humanities and Sciences, edited by Edwin Batteistilla, Published by Indiana Univ. Linguistic Club, August 1984.

[14] A. Salah, K. Reilly, and C. C. Yang, "A Logic Programming Approach to Decision Table Definition and Implementation," Presented in the ACM Midsoutheast Conference in Gatlinburg, TN, November 1984.

[15] "UNH Prolog User Manual" N.H.: University of New Hampshire, July, 1983.

[16] D. Warren, L. Pereira, and F. Pereira, "Prolog- The Language and its Implementation Compared with LISP," Proc. Symp. on A.I. and Programming Languages, SIGPLAN Notices, Volume 12, Number 8, Aug. 1977.

[17] S. M. Weiss and C. A. Kulikowski, A Practical Guide to Designing Exert Systems, Rowman & Allanheld Publishers, 1984.

[18] R. Welland, Decision Tables and Computer Programming, U.K.: Heyden & Son Ltd, 1981.

[19] C. C. Yang, "Associative Memory Systems and Their Applications to Picture and Arithmetic Processings," Ph. D. Dissertation, Northwestern University, Evanston, Ill., 1966.

[20] C. C. Yang, Relational Databases, Prentice-Hall, Inc., Englewood Cliffs, N. J., 1986.

[21] S. S. Yau and H. S. Fung, "Associative Processor Architecture - A Survey," ACM Computing Surveys, Volume 9, Number 1, pp. 3-28, March, 1977.

Appendix A-1

Implementing a system of DTs by a uprolog program

This implementation of tables 2 and 3 is based on the representation of section 5.1 for achieving a condition policy map. A dialect of Prolog, known as uprolog [15], is used. The uprolog writes a headed Horn clause as

Predicate in head :- First predicate in body, ..., Last predicate in body.

The clauses headed by the predicate "table2" or "table3" represent table 2 or 3, respectively. This program is created and stored as a UNIX file with name Fig.1. The predicates "input2," "input3," "proceed," and "check" provide an interactive session. The underscore "_" means don't care and matches any value.

Table 2. A DT with embedded DTs and a don't-care entry

MAKE PERFORMANCE	cord good	cord poor	reo poor	reo good	ford -
COMMISSION SHOPWORK APPROVAL	1% not-needed not-req	3% 2-weeks not-req	5% 3-weeks not-req	table3(reo) not-needed not-req	table3(ford) 6-weeks required

Table 3. A DT named "table3" and having an ELSE rule

MAKE Car working? Older than 30 years?	reo y y	reo n n	ford y y	ford n y	ford n n	E L S E
COMMISSION	3%	3%	10%	3%	1%	5%

```
table2(cord, good):- commission('1%'), shopwork(not-needed),
                approval(not-required).

table2(cord, poor):- commission('3%'), shopwork(2-weeks),
                approval(not-required).

table2(reo, poor):- commission('5%'), shopwork(3-weeks),
                approval(not-required).

table2(reo, good):-input3(reo), shopwork(not-needed), approval(not-required).

table2(ford, _ ):-input3(ford), shopwork(6-weeks), approval(required).


table3(reo, y, y):-commission('3%').

table3(reo, n, n):-commission('3%').

table3(ford, y, y):-commission('10%').

table3(ford, n, y):-commission('3%').

table3(ford, n, n):-commission('1%').

table3(_ , _ , _ ):-commission('5%').  /* ELSE rule */


commission(X):-write('COMMISSION is   '), write(X), nl.

shopwork(X):-write('SHOPWORK is   '), write(X), nl.

approval(X):-write('APPROVAL is   '), write(X), nl.

input2:- write('MAKE?   '), read(Make), write('PERFORMANCE?   '),
        read(Performance), nl, table2(Make, Performance), proceed.

input3(Make):- write('Is the car working?   '), read(A1),
           write('Is the car older than 30 years?   '), read(A2), table3(Make, A1, A2).

proceed:-nl, write('Do you want "to" proceed?   '), read(Answer), check(Answer).

check(y):-input2.

check(yes):-input2.

check(Other):- !.
```

Example A-1:

This example illustrates that a function entry denoting an embedded DT, such as "input3(reo)" or "input3(ford)" in the action quadrant, can be automatically processed.

```
% uprolog

| ?- consult('Fig.1').
[Fig.1 consulted]

yes
| ?- input2.  /* cpm of table 2 */

MAKE? |: reo.  /* transaction begins */
PERFORMANCE? |: good.  /* end of transaction & invoke input3(reo) */

Is the car working? |: y.  /* cpm of table 3: transaction begins */
Is the car older than 30 years? |: n. /* end of transaction & process ELSE rule */

COMMISSION is 5%  /* result begins */
SHOPWORK is not-needed  /* result continues */
APPROVAL is not-required  /* result ends */

Do you want to proceed? |? no.

yes
| ?- table2(ford, anything).  /* invoke input3(ford) and process table3 */

Is the car working? |: n. /* cpm of table 3: transaction begins */
Is the car older than 30 years? |: y.  /* transaction ends */

COMMISSION is 3%  /* result begins */
SHOPWORK is 6-weeks  /* result continues */
APPROVAL is required  /* result ends */
anything  = _ 13;  /* anything means "don't-care" & matches any symbol */

no
| ?- halt.  /* terminate session */
```

Appendix A-2

Implementing a DT by a cprolog database

This implementation, based on the representation of section 5.2, is also an interesting and promising one for processing a single DT. Example A-2 shows an implementation based on "cprolog" [1, 20] that is the Prolog dialect (i.e., a version of Prolog with extensions plus additional programs independent of Prolog itself, which implements a relational database [1, 11, 20]). cprolog writes a headed Horn clause as

+ Predicate in head - First predicate in body - ... - Last predicate in body;

Example A-2:

This example shows creating a database and processing DT queries concerning the DT of table 4 that is in the horizontal format. The programs: DB_INIT, DB_MAKE, and DB_SCAN are independent of Prolog and are executed under the UNIX environment.

Table 4. A relation in a relational database

MAKE	PERFORMANCE	COMMISSION	SHOPWORK	APPROVAL
cord	good	1%	not-needed	not-required
cord	poor	3%	2-weeks	not-required
reo	poor	5%	3-weeks	not-required
ford	good	variable	6-weeks	required
ford	poor	variable	6-weeks	required

First of all, we use DB_INIT to initiate the underlying database.

```
% DB_INIT  /* initialize the database */
```

Secondly, we create a file, called a textfile, under the UNIX environment. The textfile created here is arbitrarily named as table4 that is displayed in the following.

```
% cat table4  /* display the textfile table4 already created */
#RELATION TABLE4 5  /* indexfile TABLE4 of arity 5 */
#KEYS 1 2 3 4 5 2  /* 1st two columns form the primary key */
#ARGS 1 MAKE  /* 1st column */
#ARGS 2 PERFORMANCE  /* 2nd column */
#ARGS 3 COMMISSION  /* 3rd column */
#ARGS 4 SHOPWORK  /* 4th column */
#ARGS 5 APPROVAL  /* 5th column */
cord, good, 1%, not-needed, not-required  /* 1st tuple */
cord, poor, 3%, 2-weeks, not-required  /* 2nd tuple */
reo, poor, 5%, 3-weeks, not-required  /* 3rd tuple */
ford, good, variable, 6-weeks, required  /* 4th tuple */
ford, poor, variable, 6-weeks, required  /* last tuple */
```

Thirdly, we use DB_MAKE to convert the textfile table4 into the indexfile TABLE4 that is a relation in the underlying database. Note that processing queries must use TABLE4.

```
% DB_MAKE table4 /* convert table4 into TABLE4 */
```

Fourthly, we use DB_SCAN to convert the indexfile TABLE4 into a balanced-tree and display TABLE4.

```
% DB_SCAN TABLE4  /* convert TABLE4 into a balanced-tree and display it */
```

Displaying of TABLE4 is omitted here.

Now we are ready to invoke the interpreter, cprolog, and the input file, input, and leave the UNIX environment.

```
% cprolog input  /* invoke cprolog and input */
```

Then query processing proceeds.

Query 1: Find all DR images for the transaction "ford" that is a single condition entry.

```
?-TABLE4(ford, *, *c, *d, *e)
 -w(*c) -w(" ") -w(*d) -w(" ") -w(*e) -line;

variable 6-weeks required  /* first result */
variable 6-weeks required  /* second result */
```

Query 2: Find all DR arguments for the transaction "variable" that is a single action entry.

```
?-TABLE4(*a, *b, variable, *d, *e) -wr(*a, *b) -line;

ford    good  /* first result */
ford    poor  /* second result */
```

Query 3: Fine all combinations of condition and action entries matching the transaction "not-required" that is a single action entry.

```
?-TABLE4(*a, *b, *c, *d, "not-required")  /* constant containing any nonalphanumeric character needs to be enclosed in a pair of double quotation marks */

?-w(*a) -w(" ") -w(*b) -w(" ") -w(*c) -w(" ") -w(*d) -line;

cord good 1%  not-needed  /* first result */
cord poor 3%  2-weeks  /* second result */
reo poor 5%  3-weeks  /* third result */

?-stop;  /* end of session */
```

APPENDIX B

A Logic Programming Perspective on Decision Table Theory and Practice

Kevin Reilly, Akram Salah, and Chao-Chih Yang

This paper studies the relationship between DTs and Prolog. Both of them have high expressive power and wide variety of applications. However, each of them has its own areas of application: DTs have been used as a tool in early stages of problem description, while Prolog is used for problem specification and implementation. Beside forming a programming foundation for the integration, the study provides an approach for software development, using DTs for problem description followed by direct implementation in Prolog.

The paper starts with an overview of basic concepts for both DTs and Logic Programming. Then definitions of DTs as functions and relations are provided. Implementations of a prototype DT problem are used to demonstrate different logic programming approaches to the problem, with comparisons among them. Several practices and conventions that are used in decision table processing, such as "Don't care" or "Else rule" are discussed. Ordering issues are also discussed. It is shown that the use of Prolog to implement DTs is more natural and easy than the use of conventional programming languages.

This paper is ready to be submitted for publication.

A
LOGIC PROGRAMMING PERSPECTIVE
ON
DECISION TABLE THEORY AND PRACTICE

Kevin D. Reilly **Akram Salah**

Department of Computer and Information Sciences
The University of Alabama at Birmingham
University Station
Birmingham, AL 35294

Chao-Chih Yang

Department of Computer Sciences
North Texas State University
Denton, TX 76203-3886

ABSTRACT

After the unfilled potential of decision tables (DTs) and disadvantages in current DT practices are noted, future promise is cited, based on providing a theoretical foundation and on conjoining DT theory and practice with logic programming (LP), Prolog, and relational databases. Overviews of basic DT and LP concepts are presented as a prelude to formal DT definitions based on ones promulgated by Codasyl's Decision Table Task Group in 1982. A balanced emphasis on functional and relational representations is established in the theory and as a precursor to presentation of prototype implementations that are compatible with the definitions and can be assessed in light of the LP theory of term and predicate data organizations. Three implementations with term (function and constant) data organizations, which adhere to the functional definition of DTs, are outlined; their origins in a larger DT processing context are delineated and their putatively automated interconnections are elaborated. Two predicate data representations, which adhere to the relational definition of DTs, are implemented as prototypes, and the impact of relational level processing is elucidated, while discussion of data organization and interconnections among implementations continues. Accent is on "tables for procedures and regulations" in Herman McDaniel's terminology. Use and performance characteristics for the implementations are mentioned. A summary on the attractiveness of the LP perspective ensues, with commentary on DT properties obtained at little cost in extra coding: capturing of precepts on ordering within DTs and don't care, facilitating DT updating and integrity constraint expression, and expediting implementation of table systems. Novel means of simulating policy maps, wide ranging DT accessing mechanisms, and a connection to relational data base theory display efficacy of the methods proposed in the paper. Culminating remarks are on how Prolog and DTs are mutually supporting methodologies offering prospect for enhancement and automation of whole lifecycle schemes.

Introduction

Decision Table (DT) work began over three decades ago as a support device for lifecycle phases such as specification and documentation. Later, in the 1960's, interest spread to automatic translation of DTs to flowchart (procedural) programming codes. More recently, growing use of graphics equipment coupled with increased emphasis on tools which span the entire software lifecycle have brought increased attention to both diagrams and tables. The popular textbook [GANE79] finds a role for them in procedural definitions within dataflow diagrams, in a "structured" approach to systems analysis, and the text [HURL83] seeks infusion of the discipline of DTs into the software engineering enterprise.

Past contributions to DT understanding have primarily stressed practice, sometimes employing rather informal methods. Despite any shortcomings in this, the ubiquitous use of DTs within the lifecycle has conferred on them a reputation for compactness, self-documentation, modifiability, handling complex logic, redundancy and completeness checking, high degree of non-procedurality, and automatic conversion to code. The notion that table forms might be the mantlepiece of a "whole lifecycle" approach has been asserted by several writers, but this prospect has not materialized.

A liability which led to the failure of this mantlepiece notion lies in the psychological realm: the very same discipline that is an advantage sometimes is a disadvantage, e.g., in forcing a programmer to think in terms of completeness, and perhaps more so, in terms of gathering a set of questions (conditions) into a grouping and their answers (actions) into another grouping with perhaps independent specification of their interconnections, a technique for which training and practice are needed [MONT73, METZ77]. Also, until quality graphics are available on a widespread basis to integrate diagrams, tables and text, adoption of DTs methods will be retarded. Finally, a liability derives from neglect of the logic roots of the method, often resulting in appearance of forms of tables which are in ways inadvisable. Neglecting the roots further creates an impression of an amorphous topic and one of little theoretical interest.

The CODASYL Decision Table Task Group report [CODA82] is a response to the informality and is an important stimulus to the present work. It is the viewpoint of this paper that the logic roots of DTs can be made the central focus, in a manner consistent with CODASYL's, but with additional notions to emphasize compatibility of theory, problem description techniques, and implementations. This viewpoint is made possible in large part because of the logic programming (LP) and Prolog phenomenon. The LP-based practices we present obviate some of the psychological problems, because of compatibility of DTs with Prolog programs. The paper characterizes selected portions of DT processing, which are crucial to the programming task, in terms of a logic foundation and puts them into a framework consistent with those of mathematical logic. Formal specifications in logic form are assumed fundamental in our view. LP, part of which is realized in the programming language, Prolog, which we adhere to in implementations, is assumed an appropriate way to achieve our objectives. General graphics issues are addressed in other parts of our work, and are not discussed here. Other features we discuss, e.g., the relational database (RDB) connection, suggest (sometimes atypical) ways to display DTs.

DTs may also be viewed as a theory of decisions, and in contemporary terms, an extension of ordinary production rules for expert systems applications [WEIS84]. The future potential of DTs probably is best characterized by the breadth of associations which can be put into the condition-action dichotomy: question-answer; query-response; test-result; stimulus-response; antecedent-consequent; transaction-activity; input-output. The variety of the ways diagrams and tables can describe relationships is an indication of their widespread applicability and reflects the fact that any table with well-defined input and output can be put into DT format. The underlying logic foundation brings some unity to this diversity and the diversity itself is ostensibly a good omen for gauging the future potential of DTs. E.g., in our work we have applied DTs in some less conventional studies, e.g., linguistic problems and expert systems, as well as conventional ones in software development [REIL84c, REIL84d, SALA86b].

Future prospects are not complete with comment on how what LP can do for DTs is complemented by what DTs can do for LP, though this matter is not fully developed in this paper. Logic programs often suffer from a kind of "sliver effect" in which isolated lines of code are

difficult to relate. A DT can be utilized to make groupings, which can be displayed as a unit. Furthermore, in LP, a specification and its implementation are ideally the same, the correctness problem then being solved. However, a good specification sometimes is a poor program and a clear specification sometimes does not work at all, e.g., in the arches problem [KOWA79a]. Specifying the logic of a problem in table form, with accent on those features for which DTs are strong, can provide guidance for a subset of practical problems. The pattern of accessing inherent in Prolog then can be correlated with a notion of a class of well-behaved DTs, correct DTs, for which specification of the table and its implementation are equivalent. The design phase of developing a program, absent only when a specification is a good implementation, often requires outside aids, and the hypothesis is that such DTs, which satisfy certain definitions, such as the ones given in this paper, may suffice in many cases. Finally, basing DT theory on LP helps forge a connection with relational database (RDB) systems and more general relational processing systems, from which concepts can be borrowed to refine and extend the notion of well-behaved DT, e.g., to cases with function entries in the actions.

The paper first overviews DT and LP concepts. Then, formal definitions for DTs are presented. Methods for implementation, their roots and properties are delineated. A review of a few well-known DT topics in the new perspectives is followed by elucidation of flexible accessing of table information and the aforementioned connection to RDB systems. Implementation techniques provide a concrete realization on accessibility of data consistent with Kowalski's observations on terms and predicates as data structures [KOWA79a]. Throughout the paper, it is assumed that implementations are to be as close to specifications as possible. Thus, the appropriate implementation technique is interpretation of tables, not compilation of them to flowchart programs.

DT Background

A DT may be defined as a functioning entity in which a dataflow from conditions to actions obtains. A query is posed to a table presentation processor (TPP), which directs testing against conditions and generates appropriate actions. A particular illustration may be helpful, using an

example table of the type Herman McDaniel would classify as in the class of "tables for procedures and regulations" [MCDA68]. A query identifying car make of cord and performance of good evokes actions of: COMMISSION is 1%; SHOPWORK is not-need; MANAGER-OK is not-required in the "car performance table" (CPT) modeled after a DT in [MONT73]:

CPT

MAKE PERFORMANCE	cord good	cord poor	reo good	reo poor	duesenberg good	duesenberg poor
COMMISSION SHOPWORK MANAGER-OK	1% no-need not-req	3% 2-weeks not-req	variable no-need not-req	5% 3-weeks not-req	variable 6-week required	variable 6-weeks required

A DT may be defined structurally in terms of four quadrants, condition stub, condition entry, action stub, and action entry. A vertical double line separates stubs from entries, and a horizontal double line separates conditions from actions. Concepts such as condition subject, condition alternatives, action subject and action alternatives, define symbols used to fill the four quadrants. A *profile* is a pair either of condition subject and condition alternatives or action subject and action alternatives. In CPT, the first profile's condition subject is MAKE, the second, PERF, and so on. A *rule* associates conditions with actions, such that, if all the conditions in a rule are satisfied simultaneously, the actions in the rule are performed. A vertically formatted DT is one in which the rules appear as columns, whereas in a horizontally formatted DT, rules are rows.

At a more detailed level, the types of data which can be supplied to the entry portion come into focus. CPT is an *extended entry* DT, wherein textual information such as words and phrases are used in the subjects and alternatives. An open vocabulary exists in this case and may include function and relation symbols, as a mechanism to invoke other tables. In a *limited entry* DT, entries come from a closed set of primitive symbols, e.g., y (yes), n (no), y! (yes by implication), or n! (no by implication), - (don't care), and x or X (perform this action). Clearly, perspective allows viewing extended entries as a generalization of limited entries and limited entries as a specialization of extended entries. Nothing precludes mixing of entry types within a table.

DTs promote easy checking for certain properties characterizing problem behavior [CODA82, HURL83, WELL81]. A DT is *complete* when each possible outcome of every condition tested is included in at least one rule. An Else rule represents all rules not explicitly in the table. It trivially completes any table. *Redundancy* occurs when a rule appears more than once within a DT. *Ambiguity* is expressed as: if conditions of any two rules are the same but the actions differ, a table is deemed to be ambiguous. Ambiguity is distinct from but closely related to *inconsistency,* which occurs when conditions of any two rules are the same but actions contradict each other. More complex forms of ambiguity and inconsistency arises with interdependent profiles [MONT73].

Each property expresses a desideratum, but is subject to qualification. Completeness may have little bearing on usefulness, and use of an Else rule to complete a table often is cited as artificial or inadvisable [CODA82]. Redundancy can be a virtue for very large tables in which external storage is needed and some rules occur more frequently than others. Emphasis, i.e., issuing repeated instructions, may decrease probability of failure to perform (essential) actions. Inconsistency and ambiguity are sometimes tolerable, e.g., during periods of change where both old and new versions of rules are kept for observation and assessment until the new versions are fully accepted. Ordering of rules and profiles within a DT is a complex topic commented upon more fully later. Recommendations on ordering and ones that a table should be complete, non-ambiguous, consistent, and non-redundant are part of the topic of "DTs and Software Engineering", i.e., the discipline of DT methodology [HURL83].

DT theory seeks to provide a formal structure for table processing, in formal logic, based on conjunctive logic. Whether a given table can satisfy a query is possible only if the query is embedded within a conjunctive normal form governing the entire table, e.g., for CPT: (MAKE is cord or reo or duesenberg) and (PERFORMANCE is good or poor) determines that a query of MAKE, reo, and PERFORMANCE, poor, can be answered. The advent of using formal logic as a programming language, popularly realized in the programming language, Prolog, suggests that the logic base of Prolog, i.e., LP, might provide an appropriate foundation for a logic-based methodology such as DT processing. Our DT theory work starts from [CODA82], but departs in

nuance and emphasis, relaxing the concern with functionality in favor of a relational view when circumstances warrant, e.g., for expressing highly flexible accessing of DT content and for connection to RDB systems. This paper contends that LP indeed is an appropriate base, and more, i.e., use of DTs in parts of problem descriptions benefits Prolog problem formulations. Combining DTs and Prolog in a software development approach may provide a systematic way to describe, design, and implement software systems.

Logic Programming and Prolog

Clausal logic, on which LP is based, is a special case of first order predicate calculus. Any expression in standard first order logic can be (re-)expressed in clausal form. Among the claims for clausal logic [KOWA79a] is that its expressions are close to natural language ones and that many problem-solving models in Artificial Intelligence and computer programming formalisms can be conveniently described in clausal form.

Features of clausal logic can be illustrated using elementary examples. A property of an individual may appear as a variable-free one-place predicate such as god(Zeus), whereas a relationship between two individuals may be represented by a variable-free two-place predicate such as: likes(John, Mary). A sentence such as "Mary likes anyone who likes her" is normally rendered by an expression involving a universally quantified variable, x, such as: likes(Mary, x) ← likes(x, Mary). The symbol, ← is read, if, in a left to right fashion. A literal reading of this clause is: "Mary likes x if x likes Mary." Clauses may be expressed using only variables, as in: mortal(x) ← man(x), which may be read as "all men are mortals." Some subtleties occur, e.g., in dealing with existential quantifiers.

A clause is formally defined as: B_1, ..., B_n ← A_1, ..., A_m, where A's and B's are atomic formulae, B_1, ..., B_n are alternative conclusions, and A_1, ..., A_m are joint conditions. A Horn Clause (HC) is a special case of a clausal form, containing only one conclusion. In theory, there is no order to processing of the A's and B's such that three forms of non-determinism (ND) occur: choice of which clause, which A_i and which B_j to process next.

Conventional programs mix the logic of problem solving with control over the order in which the information incorporated in the logic is used [KOWA79b]. There are some decided practical disadvantages to such a mix: in creating and updating programs, in distracting the programmer from the principal task of securing a proper specification for a problem, and in establishing a premature commitment to processing order. The latter disallows the processor to establish its own and putatively more appropriate orderings, including, of course, ones which the programmer has not foreseen. It also prevents the processor's option to change the order of processing when circumstances surrounding the calculation change, e.g., when the pattern of processing changes, perhaps determined by the processor itself, e.g., in terms of frequencies. Moreover, the approach offers dividends in making it easier to prove correctness of programs: an implementation may be identical to its specification and then no correctness proof is needed. A correctness proof, once and for all, for the executor, satisfying the proof criteria of first-order logic [CHAN73], governs most of the control aspects of any given implementation, so that the user's correctness task is made both quantitatively and qualitatively simpler.

Decision Table Definitions

This section provides definitions for DTs based on the CODASYL Decision Table Task Group's [CODA82] theoretical foundations, and covers: conditions and actions, properties of them, treating a DT as a function, and, more generally, as a relation. Representations of data as terms and as predicates are included. The ensemble shows how data representation choices of LP impact DT formalisms.

A *condition set,* C, and an *action set,* A, are defined analogously: $\{C_i, 1 \leq i \leq n\}$, where n is the number of conditions, each condition C_i being an ordered pair consisting of a condition subject CS_i, and a set of alternatives CA_i; and $\{A_i, 1 \leq i \leq m\}$, where m is the number of actions, A_i is composed of AS_i and AA_i values, and so on.

A *condition space,* SPACE(C), and an *action space,* SPACE(A), are defined as the complex products of n sets of condition alternatives, i.e., SPACE(C) = * $(CA_1, ..., CA_n)$, and m sets of action alternatives, i.e, SPACE(A) = * $(AA_1, ..., AA_m)$, where * denotes the complex product.

The following applies the definitions to the CPT problem:

C = {(MAKE,{reo,cord,duesenberg}),(PERFORMANCE,{good,poor})}.

SPACE(C)= {(reo good),(reo poor),(cord good),(cord poor),(duesenberg good),(duesenberg poor)}

A= {(COMMISSION, {1%, 3%, 5%, variable}),
(SHOP-WORK, {2-weeks, 3-weeks, 6-weeks, not-needed}),
(MANAGER-OK {required, not-required})}.

SPACE(A) =
{(1% 2-weeks required),(1% 2-weeks not-required),
(1% 3-week required),(1% 3-week not-required),
(1% 6-week required),(1% 6-week not-required),
(1% not-needed required),(1% not-needed not-required),
(3% 2-weeks required),(3% 2-weeks not-required),
(3% 3-week required),(3% 3-week not-required),
(3% 6-week required),(3% 6-week not-required),
(3% not-needed required),(3% not-needed not-required),
(5% 2-weeks required),(5% 2-weeks not-required),
(5% 3-week required),(5% 3-week not-required),
(5% 6-week required),(5% 6-week not-required),
(5% not-needed required),(5% not-needed not-required),
(variable 2-weeks required),(variable 2-weeks not-required),
(variable 3-week required),(variable 3-week not-required),
(variable 6-week required),(variable 6-week not-required),
(variable not-needed required),(variable not-needed not-required)}

A reader should note that elements in SPACE(C) and SPACE(A) are strings produced by complex product, thus the order of values may change without altering the meaning. The full complex product is not displayed above due to its length.

We refer to a string (reo good) or (good reo) as a *table condition entry* (TCE) which, formally, is an element in SPACE(C) appearing in a rule in a given DT. It is a string of condition entries used to access a DT. Similarly, a *table action entry* (TAE), an element in SPACE(A) appearing in a rule in a given DT, is a string of action entries.

DOM(T), the *condition domain* of a table, is a non-empty subset of SPACE(C), such that each element appears as a TCE in table T. The condition domain for CPT: DOM(CPT) = {(cord good),(cord poor),(reo good),(reo poor),(duesenberg good), (duesenberg,poor)}. Using these definitions, completeness can be succinctly specified as: DOM(T) = SPACE(C). CPT is complete since: DOM(CPT) = SPACE(C_{CPT}). If any rule is deleted from this table, it becomes incomplete.

The basic definitions apply directly to extended entry DTs, since they put no restrictions on the nature of the elements in the table. Limited entry DTs are determined by restricting condition alternatives, e.g., in a simple case, to Yes or No, i.e., CA_{ij}= {Yes,No}, and action alternatives to X ("perform action"), i.e., AA_{ij}= {X, }.

A DT as a Function

A *function* f: X → Y is a subset of ×(X,Y) such that for any two tuples (x_1, y_1) and (x_2, y_2) in ×(X,Y), if $x_1 = x_2$, then $y_1 = y_2$, i.e., f = {(x,y) | each x ∈ X has a unique y ∈ Y}. If (x,y) is in f, then y is called the value of f for x or the image of x under f. The domain of f is the set D_f = {x | (x,y) ∈ f for some y ∈ Y}. The range of f is the set R_f = {y | (x,y) ∈ f for some x ∈ D_f}. Under appropriate circumstances, a decision table is a function: each rule mapping a TCE to a TAE, i.e., DT = {(c,a) | each c ∈ SPACE(C) has a unique a ∈ SPACE(A)}. This definition implies non-ambiguity of the table, since each argument, a TCE, has a unique image, a TAE, under DT. Applying the definition of a function, the domain is D_{DT} = {c | (c,a) ∈ DT for some a ∈ SPACE(A)}. This is equivalent to the table condition domain, DOM(T). The range of the function DT is R_{DT} = {a | (c,a) ∈ DT for some c ∈ SPACE(C)}. The range of CPT is {(1% not-needed not-required), (3% 2 weeks not-required), (5% 3 weeks not-required), (variable not-needed not-required), (variable 6 weeks required)}.

A DT as a Relation

There are some advantages to viewing DTs as relations. [CODA82] identifies DTs with relations but stresses functionality. We need relations for several reasons: implementations that allow us to achieve objectives that are relational in nature, such as ambiguity in tables undergoing modification, connection to RDBs, and extended forms of accessing.

Since functions are just a special case of relations, all the tables described so far are (also) relations. The following definition incorporates the cases when the DT is relational and not functional as well as the functional ones. A DT rule can be defined as a logic implication in the form: c → a_1, ..., a_i, where c ∈ SPACE(C), a_i ∈ A for 1 ≤ i ≤ m, and m is the number of action

subjects. Other forms of implications exist, e.g., c → a, where c ∈ SPACE(C) and a ∈ SPACE(A). Or: $c_1, ..., c_n \rightarrow a_1, ..., a_m$, where $c_i, \in$ C for $1 \leq i \leq n$, and $a_j, \in$ A for $1 \leq j \leq m$. This last form is a clausal form but is not a HC. The last form is actually preferable over the first in that it facilitates flexibility in expression and updating, but it cannot be processed in Prolog, and so is not used in the sequel. The rules in CPT defined in logic implications in the first format are:

CPT = {cond(cord, good) → comm(1%), shwk(no-need), mok(not-required).
cond(cord, poor) → comm(3%), shwk(2-weeks), mok(not-required).
cond(reo, good) → comm(variable), shwk(not-needed), mok(not-required).
cond(reo, poor) → comm(5%), shwk(3-weeks), mok(not-required).
cond(duesenberg, good) → comm(variable), shwk(6-weeks), mok(required).
cond(duesenberg, poor) → comm(variable), shwk(6-weeks), mok(required).}

DTs of the form we have been discussing can also be viewed as a (single) relation. A DT is defined as a relation in which the set of attributes is the union of condition and action subjects and the alternatives for each condition or action is its domain. An element in this (DT) relation is a concatenation of a string of n condition alternatives and a string of m action alternatives. Expressed as a subset of the complex product, it becomes: $\times(*(CA_1, ..., CA_n), *(AA_1, ..., AA_m))$

A schematic problem definition using terminology used in the RDB field, i.e., attribute set, domain, relation scheme and dependencies, for CPT may appear as:

Attribute set= {make, perf, comm, shwk, mok}
Dom(make)= {reo, cord, duesenberg}
Dom(perf)= {good, poor}
Dom(comm)= {1%, 3%, 5%, variable}
Dom(shwk)= {2-week, 3-weeks, 6-weeks, not-needed}
Dom(mok)= {required, not-required}
Relation scheme= (Attribute set, Dep={make, perf --> comm,shwk,mok})

Prototype Implementations of DTs in Prolog

Among the many facets of DT processing commanding attention is the unit which processes user queries, the table presentation processor, TPP. A TPP is an interactive program which prompts the user, collects and packages query input, accesses the data of the DT, and generates responses. An example of prompting and response was given earlier. A TPP interaction of this type is assumed in each of the implementations discussed. The TPP employs two major kinds of infor-

mation: one data, namely the core information content of the table(s); the other, access code neutral to the application in cases, but engineered to exploit features of the data in others. In this latter situation, accessing code virtually becomes part of the data. Also, in predicate implementations, accessing code is as simple as stating the name of the predicate, with some of the variable arguments instantiated to (constant) values. In both of these cases the distinction between data and access code is at least somewhat blurred.

The several implementations provide opportunity to analyze ways Prolog supplies mechanisms to meet application needs. Three main Prolog implementations based on a DT as a function are shown, one of which has an important pair of special cases. Two implementations based on DTs as relations include one using logic implications and involving several predicates, and the other based on a single relation. The different implementations vary in data structure, in mapping constructs to access the structure, in purposes served and environmental contexts in which they operate, and in performance.

Implementations as Functions

When a DT is defined as a function, it has SPACE(C) and SPACE(A) as its domain and range, respectively. Both SPACE(C) and SPACE(A) are expressed as data aggregates, usually involving a concatenation function, similar to the Lisp cons, and thus the implementations are called term implementations, where the terms are either functions or constants. TPPs use program code, the mapping, to associate conditions and actions. We overview the functions implementations as: 1) both conditions and actions are lists, TCEs and TAEs, respectively, in one-to-one correspondence between elements, 2) conditions and actions are sets of TCEs and TAEs, represented in lists, with the mapping represented in a set of logic clauses, each representing a DT rule, 3) conditions and actions are organized into sets of alternatives, each incorporated into a separate predicate, with an altered mapping. Also, in the last case, we discuss possibilities of mixing alternatives with table entry schemes. How the data structure affects construction of a program, specifically, that more highly structured data requires less accessing code, and certain features of user interest, are noted.

Data as a List of Lists

In this implementation, each TCE and TAE of the defined DT is represented as a list. The collection of TCEs, DOM(T), is incorporated into a list, such that at top level the data structure is a list of lists. The n-th TAE corresponds to the n-th TCE, to keep coding simple. This organization exhibits the highest level of aggregation and the least amount of accessing code, i.e., data are lists of lists, one list for the domain and another for the range, and the mapping code is a simple two-clause recursive program. The Prolog implementation centers on a predicate, conds, which has a single argument, a list of sublists, each sublist being a TCE, and a predicate, acts, a list of TAEs, each sublist ordered to correspond to the appropriate TCE in conds, where the symbols, [and], in Prolog, perform the same task as the cons function in Lisp:

```
conds([[reo,good],[reo,poor],[cord,good],[cord,poor],[duesenberg,good],[duesenberg,poor]]).

acts([[variable ,'no need',required], ['5%','3 wks',required],['1%','no need',required],
    ['3%','2 wks','no required'],[variable,'6 wks',required], [variable,'6 weeks',required]]).
```

The last two action entries are the same; it is necessary to duplicate them to maintain association of the n-th TCE and TAE. Mathematically, conditions and actions are sets, but once an order is established in a table, e.g., for conds, this order is imposed on acts. Two clauses perform the mapping:

```
map(Tce,Tae,[Tce|_],[Tae|_]).
map(Tce,Tae,[H1|T1],[H2|T2]):-map(Tce,Tae,T1,T2).
```

The first clause states that: if the TCE of the query matches the conds list head, then the acts list head is the result. The second clause recursively calls map with tails of the conds and acts lists, in the event that the first clause fails.

The data organization here is generated directly from the DT spaces, as described in the definitions. The data terms are functions, in fact, functions of functions, with cons as the function. The need to duplicate TAEs confers a degree of clarity through the explicit correspondence in the data structures and makes accessing simple, but, the organization, in analogy to the situation in RDB theory, is subject to update anomalies which create inconsistency if one, say of two duplicates, is updated while the other is not.

An earlier version of this method was developed in Lisp, as part of a larger system in which other phases of DT processing are involved, e.g., table creation, including use of natural language input. A major theme in this larger context is one of portability, obtained in three ways, DTs being one of them [REIL84b]. A total of about nine different DT implementations spread over five different machines has been prototyped. A connection to Prolog is part of this overall picture, and compatibility of implementations is very important. Thus, the Prolog implementation here purposely mimics the Lisp one: the data structure of the Lisp code is ported to Prolog to produce analogous capabilities in the Prolog portion of the system. The stipulations on the data structure of the Lisp implementation and the ability to re-convert Prolog implementations to Lisp is part of a goal that operations be invertible. Thus, in the following pages of this paper, we are concerned with affinities among the implementations, and how these affinities express possibilities for (automated) transformation from one implementation to another. Specific applications of a combination of Lisp for natural language input, and Prolog for long-term storage and additional query capabilities, as part of a multiple knowledge representation scheme, are discussed in the [REIL84d].

Data as a Set of Lists

In this representation, the actions are implemented as a set. Mechanically, this removes duplication of TAEs and achieves a space savings. The functions describing the conditions, the cons-ing operations, become one-level ones, extending only over single TCEs, i.e., each table input argument is a TCE in the form of a list. The actions are in a predicate, acts, with arity $m \leq n$, and each TAE appears only one time. Using CPT as an example, with a conds predicate of arity 6 and an acts predicate of arity 5, consistent with the requirement $m \leq n$, and with an arbitrary order for the TAEs, we have:

```
conds([reo,good],[reo,poor],[cord,good],[cord,poor],[duesenberg,good],[duesenberg,poor]).

acts([variable,'not needed',required], ['5%','3 weeks',required], ['1%','not needed',required],
     ['3%','2 weeks','not required'], [variable,'6 weeks',required]).
```

Associated with this data organization is a one-clause-per-rule mapping organization, illustrated

for CPT as:

```
rule(Tce,Tae):-conds(Tce,_,_,_,_,_),acts(Tae,_,_,_,_),!.
rule(Tce,Tae):-conds(_,Tce,_,_,_,_),acts(_,Tae,_,_,_),!.
rule(Tce,Tae):-conds(_,_,Tce,_,_,_),acts(_,_,Tae,_,_),!.
rule(Tce,Tae):-conds(_,_,_,Tce,_,_),acts(_,_,_,Tae,_),!.
rule(Tce,Tae):-conds(_,_,_,_,Tce,_),acts(_,_,_,_,Tae),!.
rule(Tce,Tae):-conds(_,_,_,_,_,Tce),acts(_,_,_,_,Tae),!.
```

After dialog with the user, the TPP issues a call to the predicate, rule, with Tce instantiated to values obtained from the user, e.g., rule([cord,poor],Tae). Tae is instantiated when the predicate conds in one of the rule clauses is matched with the assertion, conds. Each rule clause associates one of CPT's TCEs with one TAE, consistent with functionally of the implementation. The underscore symbol, _ , matches any term in the arguments; the underscore here signifies the irrelevance of data values at its locations. The brevity of the condition and actions descriptions (no duplications) and the explicit nature of the accessing code is highly suggestive of this scheme's being conducive to separate construction of conditions and actions, with links between them fashioned at a later period. Rule ordering in this method is more flexible than in the previous implementation, i.e., clauses can be defined in any arbitrary order. However, associating TCEs with TAEs is accomplished by carefully placing variable names in the proper place in the predicate arguments of conds and acts.

In graphic form, this DT may appear as:

MAKE is cord & PERF is good	y					
MAKE is cord & PERF is poor		y				
MAKE is reo & PERF is good			y			
MAKE is reo & PERF is poor				y		
MAKE is duesenberg & PERF is good					y	
MAKE is duesenberg & PERF is poor						y
COMM. is 1% & SHWK is not-needed & MOK is not-required	X					
COMM. is 3% & SHWK is 2-weeks & MOK is not-required		X				
COMM. is 5% & SHWK is 3-week & MOK is not-required				X		
COMM. is variable & SHWK is 6-week & MOK is required					X	X
COMM. is variable & SHWK is not-needed & MOK is not-required			X			

This implementation's data structures stay close to the previous implementation's, and indeed, the latter can be reconstructed from the former. The new mapping, seemingly quite different at first glance, also can be related to the previous one. Refer to the conds of a specific rule, e.g., the fourth rule, conds(_,_,Tce,_,_,_): it states that there are five items in an aggregate which are of no concern, and that two of them are positioned anterior and three posterior to an item of interest. Similarly, just prior to successful match by the map code in the previous implementations, two items on the conds list have been bypassed in the recursion, i.e., they are no longer of concern, and three items to the right of the matching item are remaining elements on the list that have not been considered yet, i.e., there will be no concern for them once the match is consummated. There are exactly six states of the first two map code lines that successfully match the conds (and acts) predicates for CPT. These can be arrived at by macro processing [KOWA79a], the collected results of which produce analogs to the rule predicates of this (latest) implementation. Through these transformations, which are easy to automate, the new organization retains its ties to the overall DT processing system, of which, by design, it is a part [REIL84b].

Distributed Condition and Action Data

A representation using a mapping scheme similar to the one just seen adduces additional flexibility, e.g., changing one alternative for another such as 5% for 3% in comm. It also allows flexibility for changes not normally countenanced as updates, such as adding entirely new conditions and actions, since to perform these changes, redefinition of the spaces is required. For this scheme, TCEs and TAEs are abandoned in favor of separate clauses for each condition or action subject, with alternatives as arguments, according to the scheme: subject (*alternative*$_1$, ..., *alternative*$_j$), as in:

```
make(reo,cord,duesenberg).
perf(good,poor).

comm('1%','3%','5%',variable).
shwk('2 weeks','3 weeks','6 weeks','not needed').
mok(required,'not required').
```

The number of predicates representing data in a DT, increases over the previous case, to a total of m+n, where m and n are the number of condition and action alternatives, respectively. In CPT since there are two conditions and three actions, five clauses define the data. Once again, each clause corresponds to one DT rule. Following are rules representing CPT.

```
rule([X,Y],[A,B,C]):-make(X,_,_),perf(Y,_),comm(_,_,_,A),shwk(_,_,_,B),mok(_,C),!.
rule([X,Y],[A,B,C]):-make(X,_,_),perf(_,Y),comm(_,_,A,_),shwk(_,B,_,_),mok(_,C),!.
rule([X,Y],[A,B,C]):-make(_,X,_),perf(Y,_),comm(A,_,_,_),shwk(_,_,_,B),mok(_,C),!.
rule([X,Y],[A,B,C]):-make(_,X,_),perf(_,Y),comm(_,A,_,_),shwk(B,_,_,_),mok(_,C),!.
rule([X,Y],[A,B,C]):-make(_,_,X),perf(Y,_),comm(_,_,_,A),shwk(_,_,B,_),mok(C,_),!.
rule([X,Y],[A,B,C]):-make(_,_,X),perf(_,Y),comm(_,_,_,A),shwk(_,_,B,_),mok(C,_),!.
```

In display form, this DT may appear:

MAKE is cord	y	y				
MAKE is reo			y	y		
MAKE is duesenberg					y	y
PERFORMANCE is good	y		y		y	
PERFORMANCE is poor		y		y		y
COMMISSION is 1%	X					
COMMISSION is 3%		X				
COMMISSION is 5%				X		
COMMISSION is variable			X		X	X
SHOP WORK is 2-weeks		X				
SHOP WORK is 3-weeks				X		
SHOP WORK is 6-weeks					X	X
SHOP WORK is not-needed	X		X			
MANAGER OK is required					X	X
MANAGER OK is not-required	X	X	X	X		

The TPP in this case packages information from the query to form a call such as: rule([cord,poor],[A,B,C]). The instantiations of X and Y in rule([X,Y],[A,B,C]) are passed right to the make and perf predicates in the rule clauses. When a match occurs, A, B, C, are instantiated to the appropriate response, 3%, 2 weeks, and not required, respectively.

Simplification of data organization and preparation is achieved by distributing condition and action data. The data structure in this method is defined from the condition and action sets instead of spaces, at a cost of slightly more complex mapping code. Note that reliance on cons'd structures has been eliminated and access to individual data items is direct. Though the space

concept is not directly incorporated into the formulation, it may appear as if there is no restriction on adding a new alternative to a condition or an action. However, adding an alternative to a predicate changes its arity and normally causes malfunction of the code. A supervisory program, with recourse to the definition of the table, can detect this and take steps to amend the situation.

Though we do not present the details, it can be seen that a graceful gradation exists between this implementation's mapping code and that of the previous implementation, just as the latter can be derived from its predecessor. The key idea is to convert each rule, e.g., rule(Tce,Tae) :- conds(Tce,_,_,_,_,_), acts(Tae,_,_,_,_),!. of the previous implementation, into a rule of the form: rule([X,Y],[A,B,C]) :- make(X,_,_), perf(Y,_), comm(_,_,_,A), shwk(_,_,_,B), mok(_,C),!. In LP terminology, this is achieved through meta-language processing. The program code becomes data to the meta-processor and the connections of conds and acts to make, perf, comm, shwk, and mok are examined along with the dialog code to effect needed changes. Moreover, the information captured in the (user) definition of the table provides a sufficient substrate to make these transformations or to arrive at this implementation directly from the definitions. If we observe the table (display) forms, the nature of the transformation may be somewhat clearer.

Hybrid Organizations

Other organizations can also be obtained by combination of the distributed data method and the aggregated one appearing earlier. E.g., conditions can be aggregated while actions are distributed, and vice versa, in these hybrid cases. In the former one, the condition data structure resembles the predicate conds of the aggregated implementation, while the three predicates, comm, shwk, and mok, can be identified with distributed actions. A selected rule in the mapping code for such a case, using CPT, is:

```
rule(TCE,[A,B,C]):-conds(TCE,_,_,_,_,_),comm(_,_,_,A),shwk(_,_,_,B),mok(_,C),!.
```

When action data are aggregated and condition data are distributed, e.g., for CPT, conditions are defined in two predicates, make and perf, and actions are aggregated in one predicate, acts, a typical rule becomes:

```
rule([X,Y],TAE):-make(X,_,_),perf(Y,_),acts(TAE,_,_,_,_),!.
```

In all the function implementations, a distinct part of the program represents the function's domain and range, with data as either atoms or aggregates. The mapping code utilizes variables and the Prolog executor matches query values against clause heads in the mapping code, with arguments of the head predicate instantiated and passed to variables in body predicates. The latter are checked to find if they are satisfied using the input values. If so, the result is found. If no rule clause is matched, the query fails. Thus, if the DT has N rules, the expected number of clauses to be checked, on the average, is N/2. Using a list of lists as a data structure, the conditions list contains N elements and an average of N/2 checks is performed to find the desired entry. If a TCE does not exist in the table, N checks are performed before discovering failure.

Implementations as Relations

Above, explicit code has been written to access data, caused, in part, by the need to police the functionality in the essentially relational implementation environment of Prolog. For the other part, the extra code is needed because data are represented as terms, whereas when data are represented by predicates, the Prolog executor accesses the data on its own. It is the goal, therefore, of this section to exploit relationality to a degree that both data and accessing code are in predicates. Under these conditions, tables become relations. Besides properties such as already mentioned when defining a DT as a relation, we develop themes, e.g., on the notion of input-output indifference and the implications it has for flexible accessing methods, and a connection to RDB systems for enhancement of certain capabilities.

In the first relational implementation, explicit separation of conditions and actions is maintained in that the consequences of the implications are TCEs and the antecedents are compounds of action entries. Several relations are employed in this "multiple relations" approach. Later, a single relation is used for each rule; conditions-actions separation is achieved through access code.

Multiple Relations

In the definitions section we introduced the representation of DT rules as HCs. With a condition predicate as its head and a conjunction of action predicates as its body, i.e., in a form Prolog can access through its top-down processing, we have: cond(c_1, ..., c_n) :- a_1, ..., a_i. The condition predicates of a DT have the same name and arity, the latter being equal to the number of condition subjects in the DT. For the CPT example, the arguments of a cond predicate directly represent a TCE of some rule. A full implementation for CPT, including three lines of code used by the TPP for dialog with the user, is:

```
tab1(cord,good):-comm('1%'), shwk('not needed'), mok('not required').
tab1(cord,poor):-comm('3%'), shwk('2 weeks'), mok('not required').
tab1(reo,good):-comm(variable), shwk('not needed'), mok('not required').
tab1(reo,poor):-comm('5%'), shwk('3 weeks'), mok('not required').
tab1(duesenberg,poor):-comm(variable), shwk('6 weeks'), mok(required).
tab1(duesenberg,good):-comm(variable), shwk('6 weeks'), mok(required).

comm(X):-write('commission is '),write(X),nl.
shwk(X):-write('shop work is  '),write(X),nl.
mok(X):-write('manager ok is '),write(X),nl.

dt:-write('car make  ?   '),read(X),write('car performance ? '),read(Y),tab1(X,Y),proceed.
proceed:-write('Do you want to proceed ? '),read(X),chk(X).
chk(y):-dt,!.          chk(yes):-dt,!.          chk( _ ):-!.
```

The first six clauses in the program represent DT rules, with condition-action separation achieved through an (explicit) implication symbol. Upon receiving a query from a user, the TPP invokes the tab1 predicates through, e.g., tab1(reo,poor). Upon successful match, the predicates, comm, shwk, mok are invoked and results are returned to the user. Note that accessing code is very simple, expressed in the tab1(X,Y) call, essentially just citing the name of the predicate and its arguments (as variables in the code), due to the fact that in predicate implementations the Prolog executor, and not applications code, accesses the data.

This implementation bears a strong resemblance to one of the cases discussed under the heading, Distributed Conditions and Actions, specifically, the case with aggregated conditions and distributed actions. A metaprogram can readily make the changes from that implementation to this one.

A Single Relation

A DT like CPT can be defined as a single relation, containing only atomic symbols and no directional dependency among the arguments. Like RDBs, a column of the table is an attribute and a row is a tuple in the relation. We postpone more details on the RDB connection until later. A DT as a single relation is a set of assertions in LP terminology. Each DT rule is a single predicate with arity equal to the sum of the number of condition and action subjects, as:

```
tabr(cord,poor,'3%','2 weeks','not required').
.....
tabr(duesenberg,good,variable,'6 weeks',required).
```

Six tuples are employed for CPT, each one an assertion stating an association of values in the DT. The accessing portion of the program is so constructed that the condition-action dependency is incorporated into code which directs matching of the query against TCEs, i.e., against that part of the tuple we define as conditions. E.g.: the clause, dec, matches values from the query against the arguments in the predicates, tabr:

```
dec(X,Y):-tabr(X,Y,A,B,C),!,write('commission is'),write(A),nl,
          write('shop work is'),write(B),nl,write('manager ok is  '),write(C),nl.
```

If the match succeeds, action values are written out. Since each predicate-argument is a constant and with clause heads stored in a hash table, a Prolog executor matches the predicate name and argument values as a single entity, so that one Prolog match finds the desired clause (DT rule). Finding the rule effects immediate instantiation of action entry values. The worst case in both relational implementations is the same as the best and the average case, i.e., one Prolog matching operation. If variables are included in the accessing code, as they are in the discussion of flexible accessing later, the top-down processing of Prolog is interjected into the accessing method, and the situation approaches that of the function representations.

This single relation implementation is related to the multiple relations case in a very simple way. To change from the latter to the former simply involves gathering all the constant arguments into a single, arbitrarily named predicate, and establishing it as an assertion. Refer to the union operations discussed in the section, A DT as a Relation. Adjustments in accessing code are

quite minor. Since the two representations are so similar it might be expected that they would have quite similar properties, but this is only partly the case when Prolog is used for processing. As we shall see, the single relation implementation is indifferent to what the user defines as input (or output), whereas the multiple relations implementation is not. Were LP used, then the multiple relations representation could be processed bottom-up as well as top-down and the distinction between the two implementations would disappear.

Observations and Items of Additional Interest

In the remaining sections of the paper some conventional and not so conventional topics in DT processing are discussed in light of the implementations presented in this paper. Our choices are selective due to the paper's scope. Topics of ordering, don't care, updating, integrity constraints, and systems of tables are among conventional DT issues for which new insights accrue. Policy maps and flexible accessing, and connection to RDB systems are among the less conventional topics for which the novelty factor is higher. The coverage is uneven, with less for topics in which other publications are entailed, e.g., the RDB connection. A bias in favor of utilizing the relational implementations is exhibited, due to their suiting purposes so well.

Ordering in DTs and LP

Ordering is a potentially complex issue for all the objects and languages of concern to us, i.e., DTs, LP, Prolog, and conventional programming languages. The principal ordering topics for DTs center on ordering for rules, condition profiles, and action profiles. For some DT organizations, ordering is a fundamental concern, either to increase efficiency or in determining meaning. E.g., rules with highest frequency may have to be positioned first in the list of rules to overcome inefficiency, or a profile may have to assign values to variables that are used by some later profile. Some of these features complicate DT processing, and, accordingly, are either not allowed by our definitions or are proscribed by researchers such as [METZ77], where a recommendation is given that rules and condition profiles not be considered as ordered, whereas action profiles ordering is sometimes permitted.

Ordering in Prolog is associated with non-determinism (ND). HC formulations using unrestricted resolution inherit two ND forms from LP, but practical Prolog systems, especially sequentially based ones, compromise by imposing left-to-right and top-to-bottom processing order. We demonstrate a few issues in DT processing in Prolog through the multiple relations implementation, in the following fragment, wherein the layout has been shuffled relative to its previous appearance:

```
tabl(cord,poor):-comm('1%'),shwk('3 weeks'),mok('no req').
tabl(cord,good):-shwk('no need'),comm('5%'),mok('no req').
        .....
shwk(X):-write('shop work is  '),write(X),nl.
mok(X):-write('manager OK is '),write(X),nl.
comm(X):-write('commission is '),write(X),nl.
```

Prolog's ND guarantees that these clauses are processed without regard to order, when accessing is through constant values. I.e., the two clauses headed by tabl in the fragment can be interchanged without affecting the meaning for requests involving cord-poor and cord-good. When variables appear in the accessing code, as in the flexible accessing methods below, top-to-bottom processing becomes a more important factor. A request for information about car make, cord, casually interpreted as giving meaningful indications of reality, and with only one response from the TPP, may give a different "impression" if the clauses are ordered differently on different occasions or for different users. Efficiency may also be affected, e.g., if queries about cars with good performance records are more frequent than others, rules concerning them should be placed higher up in the code, for more efficient processing. A brief summary on this rule ordering matter is that, for queries containing only constants, the recommendations of [METZ77] are satisfied.

The fact that the order in which answers emerge in the two cases is different, i.e., the cord-poor case yields the comm first, and so on, whereas a cord-good case yields the shwk first, and so on, is a profile ordering issue. The fact that processing in both these cases is left-to-right, in Prolog, is due to its not adhering to ND in processing clause bodies. Were ND applied strictly in evaluating the clause body, as it is in LP theory, it would cause erratic printout, unless our write statements are modified. A brief summarizing point is that the lapse of ND in processing clause bodies felicitously leads to our implementation's being in accord with the [METZ77] recommen-

dation on ordering of actions. If LP theory applied, we would have to reorganization our writing clauses.

A remaining profile ordering issue is that of condition profiles. Using the last CPT fragment, we see that the order of the arguments in the condition predicate is arbitrary, though all the rules follow the same order. Again, the [METZ77] recommendations are met.

The single-relation and the distributed conditions and actions implementations present a similar picture with respect to rule order. The first of the function implementations, i.e., the one in which the k-th TAE corresponds to the k-th TCE, allows free choice of order for the condition profiles so that the k-th rule can correspond to any rule in the problem description. The order of (individual) actions within a TAE is also free, allowing us to change the order of actions on a rule-by-rule basis and again meet the [METZ77] stipulations.

The points on ordering have important consequences in the systems lifecycle. Most workers, especially those who advocate logic as a specification language, seek a high degree of freedom from ordering early in the lifecycle. Ordering is seen as something that may occur later in the lifecycle, with the degree of ordering dependent on the choice of implementation language. Conventional programming languages, which mix logic and control, impose a higher degree of ordering than Prolog, so that our preference for Prolog implementations is manifest. Though the issue of ordering is not widely discussed in practical literature on software development, it does have a great effect on the compatibility of specification and implementation, and accordingly on the correctness problem.

Don't Care and the Prolog Underscore

A "don't care symbol" (DCS), often, -, in DT practice, is used as a condition entry when a question is immaterial to the rule it is in. In the following fragment, the make, duesenberg, determines actions, with no reference to performance:

MAKE PERFORMANCE	....	duesenberg –	....
COMMISSION SHOPWORK MANAGER OK	....	variable 6-weeks required	....

The rules involving duesenberg of the original table are said to be consolidated in the rule just seen, i.e., where there were once two rules there are now one. Rule consolidation obviously makes a DT more compact. Consolidation is a process that can be performed automatically, and is discussed in more detail by several writers (see, e.g., MONT[73]).

The concept of don't care is readily handled in Prolog implementations through the underscore operator, available in "standard" Prolog. We can illustrate this through the following fragment:

```
tab1(duesenberg,good):-comm(variable), shwk(6-weeks), mok(req).
tab1(duesenberg,poor):-comm(variable), shwk(6-weeks), mok(req).
```

These two rules can be consolidated since they have the same actions and differ only in one condition entry. The result is:

```
tab1(duesenberg, _ ):-comm(variable), shwk(6-weeks), mok(req).
```

The symbol, _ , matches any value whatsoever, so that this clause is satisfied when invoked by a query whose first argument is duesenberg, regardless of the value of the second argument: good, poor, or for that matter, anything at all. The same actions are performed in these cases, thereby achieving required results. If it is desired to proscribe second arguments other than good and poor, an integrity constraint (see a succeeding section) can be used.

An alternative way to incorporate don't care is to use a variable, say, X, in the position where the symbol, _ , appears: tabr(duesenberg,X,variable,6-weeks,required). In CPT, the value of X is instantiated to one of the values, good or poor, in valid queries. The value of X can be used in further programs, if needed, e.g., recording each value assigned to performance in CPT, without regard to the whether or not it affects decision outcomes. These consolidation possibilities carry over to the single relation implementation, with an example consolidated rule for CPT

appearing as: rule(duesenberg,_,variable,6-weeks,required) or rule(duesenberg,X,variable,6-weeks,required).

Updating Prolog Implementations of DTs

Aspects of DT updating include: rule deletion and addition, and intra-rule changes. Some changes, e.g., addition of condition subjects or introduction of new alternatives, are not considered an update, but rather a redefinition of the DT, since the DT space is altered by them. Changes in DT space are not necessarily more difficult, but for the most part we do not discuss them here except to mention them as a factor in supervisory programs designed to enjoin certain kinds of operations. E.g., permission from system administrators might be required to change the spaces, whereas simpler changes might be performed by users. The next section, on integrity constraints, indicates how this supervisory capability may constrain attempts to access values.

When each DT rule is represented in a single clause, as it is in most of the implementations, removing a rule from the table is achieved simply by deleting a clause. Other clauses for other rules and the access code remain unaltered. Deletion of a rule obviously changes a complete table into one which is not complete. Addition of a new rule involves only inserting a new clause anywhere in the program text. The name and arity of new rules must be compatible with the table, and predicates in the body must be properly formed, for correct processing.

Changing information within a rule from one value to another value in the space, e.g., substituting a commission of 5% for 1% in the cord-poor case in CPT, is localized to the clause representing the given rule. Thus, changing is as easy as deleting or adding, since in effect it is deleting one rule and adding another. The ease of change in relational representations results from the set level processing entailed.

The changes discussed above are "static" in the sense that they might be exercised on a table stored in auxiliary storage prior to the table's use in processing. Prolog also allows changes to be made during a session, through its retract and assert commands. Such changes are limited to the session and do not affect permanent program copies. This capability has valuable applica-

tions potential, e.g., in "what if" experiments on decisions.

Data Integrity through Auxiliary Prolog Code

It is easy to construct auxiliary clauses in Prolog to aid enforcement of data integrity. Consider a clause which might be accessed prior to writing out a value for a (salesman's) commission: numeric(X) ← commission(X). In LP terminology, this clause expresses a general rule applying to any object, X, which is a commission, i.e., it is constrained in type, to numeric.

A clause which expresses a constraint on the amount of money involved in a commission might be constructed as follows: acceptable(X) ← commission(X), gt(X,100), lt(X,1000). Conjoined with the clause in the first example it requires a commission to be numeric. Acting as a general check on car make and performance, e.g., during input of a query, to enforce the specification that only particular cases of these exist, might be clauses such as:

valid-car-make(X) ← member(X,[reo, cord, duesenberg]).
valid-car-performance(X) ← member(X,[poor, good]).

Note that such auxiliary Prolog code often can be organized into another DT, creating in the process a system of tables, a topic taken up later. In DT systems information flows from DT to DT, and auxiliary clauses such as these can guarantee that tables within the system are searched only with valid input values. Checks on types can enhance processing efficiency by preventing searches that can't be successful.

Accessing and Policy Maps in the Single Relation Case

A single relation representation accommodates a broader range of queries than is usually the case in DT practice. Among them are behaviors associated with Condition Policy Maps (CPM) and Action Policy Maps (APM), cited as significant components in a program of augmentations of DT practice thought to encourage DT usage [MONT73]. In a CPM, conditions are on the left and top portions of a grid, and actions are within (internal) cells of the grid. The behavior achieved from use of a CPM is identical to what we have been discussing all along, and we need say no more on it. In an APM, actions are on the left and top and conditions are within. These

maps imply physical reformatting of the data, which has obvious heuristic value for the user.

The approach presented here, espouses simulating the behavior of these maps, leaving physical reformatting to other programs. The code fragment below illustrates how we proceed in simulating an APM in the CPT example: clauses, apdt and adec, respectively, prompt for input and access the table, tabr, where we abbreviate read to re, and write to wr:

```
tabr(cord,poor,'3%','2 weeks','not required').
.....
tabr(duesenberg,good,variable,'6 weeks',required).

adec(A,B,C):-tabr(X,Y,A,B,C),!,wr('make:'),wr(X),nl,wr('performance:'),wr(Y),nl.
apdt:-wr('commission ?'),re(X),wr('shop-work ?'),re(Y),wr('manager-ok ?'),re(Z),adec(X,Y,Z),go_on.
go_on:-nl,wr('Do you want to go_on ? '),re(X),chk(X).
chk(y):-!,dt.  chk(yes):-!,dt.  chk( _ ):-!.
```

The predicate, tabr, is accessed from adec through arguments earlier associated with actions. I.e., the dependency of actions on conditions in single relation implementation seen earlier lies only in access code, just as the dependency of conditions on actions does here. In fact the tabr predicates can be accessed by any pattern of values, i.e., conditions only, actions only, mixtures of conditions and actions, single conditions, single actions, nothing. In the last case, the entire table is printed out. This highly flexible accessing capability demonstrates well what is meant by input-output indifference. Though the DT user may associate a subset of the subjects with conditions and the remaining subset with actions, the Prolog processor is indifferent to the user view, and happily so, since the solution to a problem in the "forward" direction, i.e., from the conditions to the actions, is also the solution to a problem in "reverse" direction, i.e., from the actions to the conditions. Moreover, the situation is even better than this: queries with patterns which mix (user-viewed) input and output can be answered, with the response supplying the complementary portions of the table input and output.

The following fragment illustrates clauses useful to gain access through single actions, e.g., by entering: commission(5%), all (TCEs) appearing in a rule which includes a commission of 5% are displayed. It also has a predicate, display, which causes printout of the entire table.

```
commission(A):-tabr(X,Y,A,B,C),write(X),write(Y).
shopwork(B):-tabr(X,Y,A,B,C),write(X),write(Y).
managerok(C):-tabr(X,Y,A,B,C),write(X),write(Y).
display:-tabr(X,Y,A,B,C),write(X),write(Y),write(A),write(B),write(C),nl.
```

This flexibility of accessing, achieved with almost no cost in added coding in the single relation implementation, is shared by the distributed condition and action implementation. In the other representations, the data structure might reflect data dependency directionally, from conditions to actions, or aggregate data inconveniently, and the code commitment needed for these cases is heavier.

The Single Relation and RDBs

The single relation representation as implemented in Database Prolog (DB Prolog) [BRUY80] demonstrates the connection between DTs and RDBs. The RDB component of DB Prolog interacts with the Prolog component in such a way that, after data (functional) dependencies are supplied by the user to the RDB component, they are automatically enforced by the Prolog component. RDB tuples are DT rules for us, and this enforcement means that features such as ambiguity, inconsistency, redundancy are automatically policed in this system. When relational level processing is required by the application, e.g., all the accessing methods just delineated are needed, the user may define the entire relation as the key. The approach also allows handling of very large tables and use of the RDB management system to query DTs and manipulate them in other ways [SALA86a]. With this system we can perform virtually all the activities outlined above, the only loss being that redundant DT rules (tuples) can't be used. Furthermore, the display features of the host RDB system can be used for additional graphics output for DTs.

The RDB limitation to atomic values prohibits invoking subtables, e.g., from the action portion of a DT as done in the next section. A study aimed at relaxing this limitation has been made, through adding Prolog codes to DB Prolog system so that a DT can have a variable name, a function symbol, or a relation symbol as an action entry. The RDB "thinks" these entries are constants and our added code construes their true semantics. Rules and data are stored and manipulated in the same environment in the proposed system we call EXPRD (Extended Prolog Rule Data system), details on which are provided in [SALA86a].

Systems of Tables

Several DTs can be involved in a problem description, possibly from the first description written by the user, or as a result of a decomposition done later, perhaps through program logic. Representing several independent, non-communicating tables in Prolog requires only that we appropriately choose names and arities for predicates, to avoid name clashes within the representation. In table systems, each table is separately represented and communication is established among them [LOND72]. Effecting communication among the tables is possible in a number of ways. A natural and useful case, used in the demonstration below, is one in which an action in one table invokes accessing code for a second table. Accessing a second table from condition entries or other parts of a DT are not discussed, a reformulation of the problem often permitting these cases to be converted to or simulated by the case we treat.

For the purposes of demonstration, we modify CPT for the multiple relations implementation such that a call to accessing code occurs in an action in the three rules in which the commission was listed as variable, e.g., tab1(reo,good) :- dt2(reo), shwk('no need'), mok('no req'). The invoked access code, dt2, calls a table, tab2, in which additional conditions are presented to the user, relating to whether the engine is working and the car is older than 30 years:

```
tab2(reo,y,y):-comm('3%').                tab2(reo,y,n):-comm('5%').
tab2(reo,n,y):-comm('5%').                tab2(reo,n,n):-comm('3%').
tab2(duesenberg,y,y):-comm('10%').        tab2(duesenberg,y,n):-comm('5%').
tab2(duesenberg,n,y):-comm('3%').         tab2(duesenberg,n,n):-comm('1%').

comm(X):-write('comm is   '),write(X),nl.
shwk(X):-write('shopwork is     '),write(X),nl.
mok(X):-write('manager OK is   '),write(X),nl.

dt2(M):-nl,write('engine ok?'),read(X),write('car 30 years + ?'),read(Y),nl,tab2(M,X,Y).
dt:-nl,write('car make ?'),read(X),write('car performance ?'),read(Y),nl,tab1(X,Y),proceed.
proceed:-write('do you want to proceed ? '),read(X),chk(X).
chk(y):-!,dt.     chk(yes):-!,dt.     chk(X):-!.
```

In this example, the value of the car make is passed from the first to the second DT and affects the outcome of this table's decision, a value for the salesman's commission. Note, also, both the new and the old tables share action clauses.

This demonstration illustrates only a few basic features of communications between DTs as well as relates how a solitary table might be augmented to become part of a system of tables through addition of calls to another table. The topic of table systems is a much more extensive topic that we can discuss here and constitutes the fare of a later paper.

Conclusions

A hypothesis that DTs and LP can be usefully employed together in a systematic approach to software problems, in problem description, specification, design and implementation has led us to explore relationships between DTs and LP. Some compatibilities between them are manifest in this report. Prolog provides a natural methodology for defining and implementing DTs, facilitating expression of DT processing for such well-known operations as "don't care", consistency checking, updating, as well as for newer features such as policy maps and connection to RDB systems. Transformation of DTs into Prolog programs can be done in more than one way, to effect solutions with good properties from a DT theoretic point of view, and which reflect applications needs. That a large class of problems have already been formulated as DTs increases the importance of facile implementation of DTs in Prolog.

DTs, on the other hand, offer help in constructing logic programs because of the correspondence between DT rules and logic clauses. DTs aid the problem description process by promoting formulations which are readily implemented in Prolog. Tables help offset the sliver effect of logic programs by fostering the grouping of related clauses, which can then be displayed for inspection. Table-based problem descriptions can be assessed for completeness, redundancy, ambiguity and inconsistency. ND and non-procedurality, still new concepts in computing science, are sometimes difficult to comprehend and DTs may offer some help in the guise of disciplined restrictions on the realizations of both of them.

References

[BRUY80] M. Bruynooghe, "Prolog in C for Unix version 7: A Reference Manual," Leuven, Belgium: Katholieke Univ. 1980.

[CHAN73] C. Chang and R. Lee, "Symbolic Logic for Mechanical Theorem Proving," N.Y.: Academic Press, 1973.

[CLOC81] W. Clockson and C. Mellish, "Programming in Prolog," Berlin, Germany: Springer-Verlag, 1981.

[CODA82] "A Modern Appraisal of Decision Tables," A CODASYL Report, N.Y.: ACM, July, 1982.

[DAVI82] R. Davis, "Runnable Specifications as a Design Tool," in K. Clark and S. Tarnlund, Eds, "Logic Programming," N.Y.: Academic Press, 1982.

[GANE79] Gane, C. and T. Sarson "Structured Systems Analysis: Tools and Techniques," Englewood Cliff, N.J.: Prentice Hall, 1979.

[HURL83] R. Hurley, "Decision Tables in Software Engineering," N.Y.: Von Nostrand Reinhold, 1983.

[KOWA79a] R. Kowalski, "Logic for Problem Solving," N.Y.: Elsevier North Holland Inc., 1979.

[KOWA79b] R. Kowalski, "Algorithms= Logic + Control," Comm. ACM, 22, 7, 1979, 424-436.

[LOND72] K. London, "Decision Tables," N.Y.: Van Nostrand Co. Inc, 1972.

[METZ77] J. Metzner and B. Barnes, "Decision Tables Language and Systems," N.Y.: Academic Press, 1977.

[MCDA68] H. McDaniel, "An Introduction to Decision Logic Tables," N.Y.: John Wiley, 1968.

[MONT74] M. Montalbano, "Decision Tables," Palo Alto CA: Science Research Associates, 1974.

[PRES82] R. Pressman, "Software Engineering A Practitioner's Approach," N.Y.: McGraw-Hill Inc., 1982.

[REIL84a] K. Reilly and J. Barrett "The ASP System: The Friendly Front," Proc. 1984 ACM-SE Conf., 124-132.

[REIL84b] K. Reilly, W. Jones, J. Barrett, A. Salah, E. Strand, J. Autrey, and P. Rowe, "Software Development Study: A Simulation Environment Perspective," ISETT 1984, ISDOS Ref. M0657, 16pp.

[REIL84c] K. Reilly, A. Salah, P. Morgan, and P. Rowe, "Multiple Representation in a Language Driven Memory Model," Papers on Computational and Cognitive Sc., E. Battistella (Ed), Indiana U Linguistic Club, Aug 1984, 99-111.

[SALA85] A. Salah, C.C. Yang, and K. Reilly, "On Decision Table Properties, Representations, and Implementations," Submitted.

[SALA86a] A. Salah, K. Reilly, and C.C. Yang, "An Integration of a Rule Representation into a Relational Database System," Submitted.

[SALA86b] A. Salah and K. Reilly, "A Reduction Methodology for Differential Diagnosis Expert System," Proc. 3rd Annual Computer Science Symposium on Knowledge-Based Systems: Theory and Applications, March 1986, 15pp.

[WELL81] R. Welland, "Decision Tables and Computer Programming," Philadelphia, PA: Heyden & Son, 1981.

APPENDIX C

An Integration of a Rule Representation into a Relational Database System

Akram Salah, Kevin Reilly, and Chao-Chih Yang

This paper is based on the previous two papers. In Appendix A, a theoretical foundation for DTs was established, and in Appendix B, a comparison of the different DT-Prolog implementations showed that using a RDBS is more proper for DT processing. Here, an integration of a DT system into a RDBS is introduced.

The paper contains first a summary of DT definitions, then a comparison of a DT system with a RDBS. The comparison shows that there is a restricted type of DTs that can directly be stored and manipulated into a RDBS, while other types need extensions to a RDBS to handle them. A configuration of an extended Prolog-RDBS that accommodates storage and manipulation of a broader type of DTs is shown. The extended system, EXPRD (Extended Prolog Rule Data), uses some data relations to store information about elements used to fill entries in DTs. This information is used by the extended management system (on the meta-level) to interpret components used as DT entries. This facility enables EXPRD to handle entries of the types: variable, function symbol, and relational symbol, in addition to the normal constant entries.

The approach developed through the integration is applied, both in application and in theory in appendices D and E, respectively.

This paper has been submitted for publication and acceptance is pending.

AN INTEGRATION
of
A RULE REPRESENTATION
into
A RELATIONAL DATABASE SYSTEM

Akram Salah, Kevin D. Reilly
Dept. of Computer & Information Sciences
The University of Alabama at Birmingham
University Station, Birmingham, AL 35294

and

Chao-Chih Yang
Department of Computer Science
North Texas State University
Denton, TX 76203

March 15, 1986

ABSTRACT

Abstracted from an integrated software development environment project which calls for elements from knowledge engineering and database theory, the management system of a relational database system (RDBS) is analyzed and extended using definitions for classes of rules, so that rules can be incorporated into the database along with facts (data in relations). The new system constitutes a knowledge representation scheme similar to rule-based systems such as Ops5 and Prolog-based expert systems, but for a generalized kind of production rule based on a decision table format. The paper provides a systematic approach to rule-based system, in the form of decision tables, within a RDBS context, identifying types of rules and how some of them can be handled by a RDBS without modification, whereas others require metadata containing descriptors for elements used in building the rule base to overcome limitations imposed by the RDBS. The paper includes a configuration of a prototype system, which is an extension of a Prolog-RDBS, to accommodate rule systems. The extended system combines capabilities of a domain-independent problem solving methodology (Prolog) and those of a domain-dependent Knowledge-intensive rule representation (decision tables) as well as the storage and manipulation facilities of a RDBS such that rules, data, and logic programs can be stored in a compatible form which facilitate easy intercommunication.

1. INTRODUCTION

During the past few years, much attention has been given to knowledge representation (KR) in expert systems (ES) [12, 13, 16, 30]. In our lab, since we are concerned about programming environment [28, 29], we view knowledge as part of a broader environment together with both data, and programs (usually, programs employ knowledge rules which are applied to data items). Our goal was to find a knowledge representation that can be integrated into a system, developed before, which is a Relational Database System (RDBS) embedded into Prolog [42]. Such an integration yields a system where data, logic programs, and knowledge rules coincide in a compatible form and thus can interact easily.

Studying and analyzing rule representations such as those might appear in Ops5 or Prolog systems, show that they represent a condition-action relationship. A problem in such systems is that rules are represented in an unstructured way which makes their manipulation hard to be systematized. Decision Tables (DT), a technique used for almost thirty years to describe problems that have complex logic [15, 20, 21, 23, 40], is a structure that represent a collection of "related" condition-action pairs called decision rules. The use of a DT format in building expert system is, informally, described (p. 118 of [39]) as a compact and clear way to represent a generalization of production rules.

For the concern of our research, DTs as a rule representation fulfill two requirements: 1) it represents rules in a structured form which facilitate systematic approach to construct, store, and analyze rules; and 2) it is a tabular representation which is a similarity with a relation in a RDBS. Thus, it was chosen as a candidate to be integrated in our system. Unfortunately, DTs did not have a widely accepted formal definition such that the comparison between a DT system and a RDBS can be based on theory. However, in a previous study [35], a DT set-theoretic definition [5] has been studied, analyzed, and generalized as well as many properties of DTs have been formulated in the light of the new definitions. Here, we proceed from this definition to show an integration of a DT system into a RDBS.

This paper is concerned mainly with showing how, in theory and practice, a RDBS can be used to accommodate storage and manipulation of rules represented in the form of DTs. We overview basic concepts of a DT, as it is presented in the set-theoretic definition [35], as a prelude to a comparison between a DT system and a RDBS. The comparison shows that a RDBS can be used to store and manipulate a limited type of DTs. It also shows that a broader type of DTs can be accommodated with extensions to a RDBS and its management system.

To extend the RDBS: (1) relations containing information about the DT-system itself (meta-data) are added to the system; (2) code, making use of the meta-data, is added to the management system to deal with the more complex items. A RDBS with these extensions, EXPRD, is presented in this paper.

In the subsequent section of this paper, we, first, show a DT rule representation then, formally, overview DT basic concepts and, finally, discuss similarities and differences between RDBS and DTs. In Section 3, we present an extended Prolog database system (EXPRD), a system that stores and manipulates rules (in DTs) as well as data in a Prolog environment. Section 4 is a conclusion.

2. BASIC CONCEPTS

In this section, we first, informally, show rule representation as if-then statement, OPS5, and as decision rules. Then, a formal definition of DTs, as it is defined in [35], is overviewed. Finally, a DT system is compared with a RDBS. We assume the reader is familiar with RDBS theory represented by such texts as [8, 36, 41, 42]. Specifically, we freely use the concepts of attribute names, domains, relational schemes and instances, functional dependencies and multi-valued dependencies, and normal forms. We also assume that a reader is familiar the conventional graphical structure of a DT [15, 20, 21, 23, 40].

2.1 Rule Representation

Rule systems are systems that are composed entirely of conditional statements called *rules*. These rules are usually stated as if-then statments, i.e., condition-action, and interpreted as: if all the conditions in a rule are true, simultaneously, then the action(s), in the same rule are consequently true. Here we show three representations of the same rules, taken from a diagnostic expert system, to demonstrate DT rules compared to other rule representations.

Rules such as ones shown below are used to describe differential diagnostic for rheumatic diseases. First, as they have been acquired, i.e., if-then form.

.....

Rule: 67
If a patient has 2 major symptoms
    and the RNP antibody is Positive
then the patient probably has Mixed Connective Tissue Disease

Rule: 68
If a patient has 4 major symptoms
    and the RNP antibody is Positive
    and he doesn't have Positive SM antibody
then the patient definitely has Mixed Connective Tissue Disease

.....

Secondly, as they are represented in an OPS5 system. Note that each rule is represented separately with no particular arrangement except that the conditions are on LHS of -> and the actions are on the RHS.

.....

```
(p diag67
    (rheumatic ^majorsym 2  ^required  RNP-Positive)
        -> (make MCTD probable))

(p diag68
    (rheumatic ^majorsym 4  ^required RNP-Positive  ^excluding SM-Positive)
        -> (make MCTD definite))
```

.....

Thirdly, the same two rules as they appear in a DT. Note that the condition-action relationship is still preserved, yet subjects under question are abstracted and placed in the left part of the table, called the table stub. Also note that a set of rules, that have the same subjects, are represented in the same DT, yet each in a separate column.

major symptoms required excluding	o	o	4 RNP positive SM positive	2 RNP positive none	o
Mixed Connective Tissue Disease	o	o	definite	probable	o

2.2 Decision Table Definitions

A DT is formally defined in [5, 35] as a triple (C, A, R) where C and A are finite nonempty sets and R is a relation. C is called the ***condition set,*** and is described: C = $\{C_1, C_2, \ldots, C_n\}$, where each C_i is called a condition and n $\geq$ 1. Each condition C_i is an ordered pair, a ***condition subject,*** CS_i, and a finite nonempty set, $CA_i = \{CA_{i1}, CA_{i2}, \ldots, CA_{is_i}\}$, where each CA_{ij} for $1 \leq j \leq s_i$ is called an ***alternative*** (for condition C_i). CA_i is the ***set of alternatives*** for condition C_i. A condition alternative, CA_{ij}, using logic terminology [17], is a term, (i.e., a constant, a variable, a function symbol), or a relation symbol. Condition alternatives often are constants.

A in the triple, is called the ***action set,*** and is defined analogously to the condition set. That is, A = $\{A_1, A_2, \ldots, A_m\}$, consists of m actions, m $\geq$ 1, where an action is an ordered pair, an ***action subject,*** AS_i, and a finite nonempty set, $AA_i = \{AA_{i1}, AA_{i2}, \ldots, AA_{it_i}\}$, where each AA_{ik} for $1 \leq k \leq t_i$ is called an ***alternative.*** An action alternative, AA_{ij}, is a term, i.e., a constant, a variable, a function symbol, or a relation symbol. An action alternative, AA_{ik}, behaves the same as a condition alternative.

R is a set of entities called ***decision rules*** and is defined as a relation between cspace and aspace, where cspace, denoting the ***condition space,*** is the complex product of the n sets of condi-

tion alternatives, and aspace, denoting the *action space,* is the complex product of the m sets of action alternatives in DT. That is, where * denoting complex product and x denoting Cartesian product

$$\text{cspace} = *(CA_1, CA_2, \ldots, CA_n) = \{(CA_{1j_1} \; CA_{2j_2} \ldots CA_{nj_n}) \mid 1 \leq j_x \leq s_x \text{ and } 1 \leq x \leq n\}; \text{ and}$$

$$\text{aspace} = *(AA_1, AA_2, \ldots, AA_m) = \{(AA_{1k_1} \; AA_{2k_2} \ldots AA_{mk_m}) \mid 1 \leq k_y \leq t_y \text{ and } 1 \leq y \leq m\}.$$

R, by definition, is a subset of the Cartesian product of cspace and aspace, i.e.,

R *subset* x(cspace, aspace)

$$R \textit{ subset } \{((CA_{1j_1}, \ldots, CA_{nj_n}, AA_{1k_1}, \ldots, AA_{mk_m})) \mid 1 \leq j_x \leq s_x, 1 \leq x \leq n, 1 \leq k_y \leq t_y, 1 \leq y \leq m\}$$

An element in the set of alternatives for a condition C_i (or for an action A_j) is called an *entry* for condition subject CS_i (or for action subject AS_j) if it is used in a decision rule. A *DT system* consists of a set of DTs, each DT in the system being defined as above. A DT is identified by its triple (C, A, R), and while it is not strictly necessary, it is useful that each table in a system has a name.

2.3 DTs and Relational Databases Compared and Contrasted

In this section we discuss analogies between: a relation in RDBS and a DT; a tuple in a relation and a decision rule in a DT; and an instance of a relation scheme in RDBS and a set of decision rules existing in a DT. Interspersed among the analogies are differences in terms of types of subjects (conditions or actions) and alternatives.

Attributes in RDBS, names associated with columns in the table representing a relation, are analogous to subjects in a DT, names associated with either a condition or with an action.

A difference exists: subjects in a DT system have a type, i.e., either condition or action, in fact by most definitions, including ours, they must. In RDBS, attributes do not have any particular type. We illustrate this distinction in an example at the end of this section.

A value for an attribute used in a database relation has to be a member in the associated domain for this attribute, i.e., value(A_i) ϵ D_{A_i}. Similarly, an entry in a DT for a condition subject CS_i (an action subject, AS_j) has to be an element in the set of alternatives for this condition subject, i.e., entry(CS_i) ϵ CA_i (entry(AS_j) ϵ AA_j).

The difference between a domain of an attribute in a RDBS and a set of alternatives in a DT system is in the types of values. A domain of an attribute contains only atomic values whereas a set of alternatives may contain a term, (i.e., a constant, a variable, a function symbol), or a relation name (i.e., in a call to a relational DT).

A relation scheme designates a relation and the attribute names in this relation, i.e., R(A_1, A_2,...., A_n), and is a skeleton of a table representing the relation. A similar DT concept can be postulated: a DT name, together with the condition subjects and action subjects used in the table, i.e., DTNAME(CS_1, ..., CS_n, AS_1, ..., AS_m) where n and m are the number of condition subjects and the number of action subjects in DT, respectively, designates a DT.

A relation in a RDBS is defined as a subset of the complex product of the domains of its attributes. Each element in a relation R, which is also an element in the complex product, is called a tuple. A set of tuples existing in R is called an instance of the relation scheme of R. Similarly, the condition space of a DT is a the complex product of the sets of alternatives for the conditions in DT, and the action space of a DT is the complex product of the sets of alternatives of the actions in DT. A DT is a subset of the Cartesian product of its condition space and action space. Thus, a difference lies in this concept as a result of the predefinition of conditions and actions in DTs.

The integrity constraints in a RDBS scheme are explicit. A similar concept exists in a DT system, though expressed differently, and leads to concepts of what we call non-ambiguous and ambiguous DTs. A DT is said to be nonambiguous if R specifies a function, i.e., each argument in R (an element in cspace) determines a unique value of R (an element in aspace). Thus, a nonambiguous DT defines a constraint equivalent to that in a RDBS and can be expressed as a functional dependency: {CS_1, ..., CS_n} -> {AS_1, ..., AS_m}, where n and m are the number of

condition and action subjects, respectively.

An ambiguous DT is defined as a DT where an argument in R (an element in cspace) determines more than one value in aspace. An ambiguous DT may sometimes define a multi-valued dependency, $\{CS_1, ..., CS_n\}$ ->-> $\{AS_1, ...,AS_m\}$.

A simple illustration reflecting the points just made based on the following example from C.J. Date [8] (see Fig.1.a). Date shows a partitioning of this relation into two 4NF relations as shown in Fig.1.b.

CTX	COURSE	TEACHER	TEXT
	Physics	Green	Basic Mechanics
	Physics	Green	Principles of Optics
	Physics	Brown	Basic Mechanics
	Physics	Brown	Principles of Optics
	Math	Green	Basic Mechanics
	Math	Green	Vector Analysis
	Math	Green	Trigonometry

Fig.1.a: Sample tabulation of relation CTX.

CT	COURSE	TEACHER
	Physics	Green
	Physics	Brown
	Math	Green

CX	COURSE	TEXT
	Physics	Basic Mechanics
	Physics	Principles of Optics
	Math	Basic Mechanics
	Math	Vector Analysis
	Math	Trigonometry

Fig.1.b: Sample tabulation of two projections of CTX: CT, and CX.

COURSE	Physics	Math
TEACHER TEXT	DT1(Phy) DT2(Phy)	DT1(Math) DT2(Math)

COURSE	Phy	Math
TEXT is Basic Mechanics	x	x
TEXT is Principles of Optics	x	
TEXT is Vector Analysis		x
TEXT is Trigonometry		x

COURSE	Phy	Math
TEACHER is Green	x	x
TEACHER is Brown	x	

Fig.2: A DT representation of the CTX relation.

This partition can be realized for a DT only if we relax the condition-action dichotomy. Then, a DT definition can be based on complex product. Recall, however, that our definition fixes the cspace and aspace at problem definition time. A different partition, based on different principles is seen in Fig.2. The partition in Fig.2 is based on an assumed goal that the TEXTs and the TEACHERs occur in an identifiable grouping within a table. They appear as subjects in the subordinate tables.

RDBS relations are restricted to contain only atomic values. If a DT contains only constant entries, it can be stored as a data relation with tuples representing decision rules and accessed analogously to query processing in a database. If, however, an entry in a decision rule is a variable name, a function symbol, or a relation symbol, some additional processing is needed. If an entry is defined as a variable name or a function symbol, its evaluation returns a value which is used as an entry in the decision rule. The case of an entry expressed as a relation symbol is more complex because it represents non-deterministic behavior [17].

The location of a non-constant entry in a DT makes a difference. Though permitting non-constants in condition entries allows some complex problems to be expressed in an elegant form, it complicates the processing of a DT system. We restrict our representation to condition entries that are constants and action entries can be any type, i.e., constants, variable names, function symbols, or relation symbols.

3. CONFIGURATION OF THE EXTENDED SYSTEM

In Section 2, we reviewed basic concepts of RDBS and DTs and identified some similarities and differences. Since database systems are restricted to atomic values, and DT are not, we must find a means to extend the capabilities of a RDBS to handle DTs. Once we accomplish this, a rule-system, represented as a DT system, can be stored in a RDBS potentially, facilitating rule querying, analysis, and processing. In this section, we provide an overview of a prototype extended system, EXPRD, designed to store and manipulate facts and rules in a single system. EXPRD is based on a Prolog relational database management system [2] with added capabilities, at the Prolog level, to manage the DT ingredients. It manages a widely used kind of DTs, with action entries of any type but with constant condition entries.

Besides managing data relations representing facts, EXPRD manages two additional components: (1) a meta-data component, EXMD, which contains descriptions of DT entities, e.g., DT names, condition subjects, action subjects, types of values; (2) a rule base, EXDT, which is a DT system constructed according to the specification in EXMD. In Section 3.1, we describe relations in EXMD and how they are used. In Section 3.2, we describe relations in EXDT and how EXPRD creates them. Then, in 3.3, we discuss how EXPRD uses meta-data to handle evaluation of non-constant entries.

3.1 EXMD: Meta-Data Component

Meta-data is represented in four data relations. The schemes for these relations are independent of any particular DT system. They are an integral part of EXPRD and cannot be deleted. Instances of these relations must be defined, as needed, by a user to describe a particular rule system. EXPRD uses the information in the meta-data relations: a) to construct relations to store DTs; b) to enforce integrity constraints, when adding a new decision rule or updating an existing one; c) to recognize and evaluate non-constant entries during rule-query processing. Following is a description of the four relations: DTABLES, DTSUBJECTS, ALTERNATIVES, and VARIABLES.

First, DTABLES, with arity 4, contains one tuple for each DT in an application system. Its scheme is

DTABLES (TABLENAME, NCONDITIONS, NACTIONS, TTYPE).

TABLENAME is a character string denoting a DT name; NCONDITIONS and NACTIONS are integers representing number of conditions and number of actions in this DT, respectively; and TTYPE is a type for this DT. The type can be, "n" for non-ambiguous, or "a" for ambiguous.

The relation DTABLES acts as the top level for a rule base. Contents of DTABLES are supplied by a user as an initial step to define a DT system. EXPRD creates a relation for each tuple in DTABLES. During processing, EXPRD greets an attempt to access a DT whose name does not exist in DTABLES with an error message.

Second, DTSUBJECTS, with arity 3, contains one tuple for each subject in a DT system. Its scheme is

DTSUBJECTS (TABLENAME, SUBJECT, STYPE).

TABLENAME is a character string denoting a DT name; SUBJECT is a character string representing a subject; and STYPE is its type (either "c", for a condition or "a" for an action) when an action subject in a DT is redefined as a condition subject in another DT, the subject appears twice in DTSUBJECTS but with different table names and different types.

Third, the relation ALTERNATIVES stores values and types of alternatives in a DT system. It contains one tuple for each alternative. Its scheme is

ALTERNATIVES (SUBJECT, ALTERNATIVE, ATYPE).

SUBJECT is a character string denoting a condition or an action subject; ALTERNATIVE is an alternative for this subject; and ATYPE is the type of this alternative: either "c" for constant, "v" for variable, "f" for formula, or "t" for the name of a DT.

Note in this relation: 1) a subject has to exist in the relation DTSUBJECTS prior to its definition in ALTERNATIVES; 2) an alternative is determined by its name together with its

subject. The relation ALTERNATIVES is essential in keeping types of alternatives. EXPRD refers to it to recognize non-constant entries.

The fourth relation, VARIABLES keeps names and values for variables which are used as entries in DTs. It has three attributes: a table name, a variable name, and its current value. The scheme for the relation VARIABLES is

VARIABLES (TABLENAME, VNAME, VALUE).

This is the only relation in the meta-data whose instances are supplyed by EXPRD itself. When an alternative is defined with a type "v" (variable name), this alternative is stored in VARIABLES, in addition to in ALTERNATIVES. When an action evaluates a variable name, the value is stored in VARIABLES. If a decision rule accesses an entry defined as a variable name, the relation VARIABLES is searched for its value.

3.2 EXDT: RULE BASE COMPONENT

EXDT is the component of EXPRD that stores the rule base. As discussed before, a rule base is represented as a set of DTs, each DT stored in coded form as a relation in EXDT.

When a user wants to define a system of DTs, EXPRD directs him to provide information to fill the meta-data relations. Upon defining a DT name and its condition and action subjects, EXPRD creates a relation with a scheme:

$$\text{TABLENAME}(CS_1, \ldots, CS_n, AS_1, \ldots, AS_m)$$

where CS_i, AS_i are the condition and action subjects, n and m are the number of conditions and number of actions, respectively. EXPRD defines a key for this relation depending on the type (stored in DTABLES) given to the DT by the user. If a DT is given the type non-ambiguous (a function), then its key is the condition subjects. If a DT is given the type ambiguous, then the key is the condition and action subjects, i.e., an all-key relation.

When a user finishes defining all his DTs, a relation for each DT is created and ready to store decision rules. With addition of each decision rule, EXPRD checks entries in this rule against values defined previously (stored in ALTERNATIVES). If the rule is valid, EXPRD stores it in the relation representing the proper DT. If not, an error message is returned to the user.

3.3 EVALUATING NON-CONSTANT ENTRY

EXPRD recognizes and interprets the elements of a DT. The input to this process is the tuple retrieved from the DT and the output is the tuple after the evaluation of non-constant entries. This process can be described by the following algorithm, where the above mentioned tuple is called Tablename.

INPUT: Tablename(e_1, ..., e_k)

OUTPUT: Tablename(f_1, ..., f_k)

given the scheme: Tablename($SUBJECT_1$,..., $SUBJECT_k$):

where k is m+n, and n and m are the number of condition subjects and the number of action subjects, respectively.

```
Process eval
 For each entry e_i for SUBJECT_i, 1 ≤ i ≤ k;
     get type from ALTERNATIVES (SUBJECT , e_i, type);
       if type = "c", then f_i = e_i;
       if type = "v", then
          get value from VARIABLES (DT, e_i, value), f_i = value;
       if type = "f", then execute(e_i, value), f_i = value;
       if type = "t", then subtable(e_i, DT(e_1, ..., e_k), value), f_i = value;
 end;
```

```
Process subtable (tablename, DT(e_1, .., e_k), value)
  bagof(DTSUBJECTS (tablename, SUBJECT, c), SUBJECT, LIST1);
  bagof(DTSUBJECTS (DT, SUBJECT, TYPE), SUBJECT, LIST2);
  get difference(LIST1, LIST2, DIF);
  if DIF=0, then call execute(eval, v1);
      else message "provide values for elements in DIF", execute(eval, v1);
    value = v1;
end;
```

"execute(name, value)" is a Prolog program that execute a Prolog clause "name" and return its value in "value". "bagof" [2, 4] is built-in Prolog program that constructs a list of values from a Prolog predicate. "difference" is a program that performs the set difference between two lists.

4. CONCLUSION

In this paper, basic concepts of a DT have been overviewed in an effort to obtain insight into the nature of rule bases. An extended management system for a Prolog-RDBS emerges from the analysis. The new system constitutes a knowledge representation scheme with generalized production rules. The paper's systematic approach to DT problems identifies an approach to decompose a DT problem using arguments from RDBS theory as guidelines. The system devised achieves a number of desirable features such as: a view of a RDBS such that data relations exist in a compatible form with rules; a methodology for merging (rule) knowledge bases with large RDBSs; a generalized system in which a RDBS is considered a special case; and an introduction of modularity within rule sets owing to grouping of related rules into a table.

The approach presented in this paper is used, both theory and application, in rule-based systems and expert systems. In [33], a formal approach similar to the one used for DTs is used to characterize a rule-based system. In [34], the approach together with the extended system is used to implement a reduction methodology for differential diagnosis expert system by adding another extension to EXPRD's management system.

In conclusion, we can say that the devised system, EXPRD, has a great expressive power, partially inherited from its components' and mostly from providing a compatible media for different types of components all of them are used in expert systems. Prolog furnishes a domain-independent problem-solving methodology. RDBS has strong formal basis as well as it provides a storage media that is easy to manipulate and retrieve. DTs does not only represent a domain-dependent knowledge-intensive system, but also it represents them in a structured formal way. The extended system, besides its expressive power, provides a systematic approach to construct, store, manipulate, process queries, and apply programs equally on both rules and data.

5. REFERENCES

[1] H. Berghel, "Simplified Integration of Prolog with RDBMS," Data Base, Volume 16, Number 3, Spring 1985.

[2] M. Bruynooghe, "Prolog in C for Unix Version 7: A Reference Manual," Belgium: Katholieke Univ. 1980.

[3] H. Cantrell, J. King, and F. King, "Logic Structure Tables," Communications of ACM, Volume 4, Number 6, pp. 272-275, June, 1960.

[4] W. Clockson and C. Mellish, "Programming in Prolog," Berlin, Germany: Springer-Verlag, 1981.

[5] "A Modern Appraisal of Decision Tables" A CODASYL Report, N.Y.: ACM, July, 1982.

[6] E. Codd, "A Relational Model of Data for Large Shared Databanks," Comm. ACM, Volume 13, No. 6, 1970, pp 377-387.

[7] E. Codd, "Relational Databases: A Practical Foundation for Productivity," Comm. ACM, Volume 25, No. 2, February, 1982.

[8] C. Date, "An Introduction to Database Systems," 4th Edition, Addison Wesley Inc., CA, 1985.

[9] R. Fikes and T. Kehler, "The Role of Frame-Based representation in Reasoning," Comm. of ACM, Vol 28, No. 9, September 1985, pp. 904-920.

[10] H. Gallaire and J. Minker "Logic and Databases," N.Y.: Plenum Press., 1977.

[11] H. Gallaire, J. Minker, and J. Nicolas, "Logic and Databases: A Deductive Approach," ACM Computing Surveys, Volume 16, Number 2, pp. 153-185, June, 1984.

[12] M. Genesereth and M. Ginsberg, "Logic Programming," Comm. of ACM, Vol 28, No. 9, September 1985, pp. 933-941.

[13] F. Hayes-Roth, "Rule-Based Systems," Comm. of ACM, Vol 28, No. 9, September 1985, pp. 921-932.

[14] P. Horstmann, "Expert Systems and Logic Programming for CAD," VLSI DESIGN, November, 1983, pp 37-46.

[15] R. Hurley, "Decision Tables in Software Engineering," N.Y.: Von Nostrand Reinhold company Inc., 1983.

[16] R. Kowalski, "Logic for Data Description," in [14], pp 77-103.

[17] R. Kowalski, "Logic for Problem Solving," Artificial intelligence series, The Computer Science Library, N.Y.: Elsevier North Holland Inc., 1979.

[18] A. Lew, "Decision Tables for General-purpose Scientific Programming," Software Practice and Experience, Volume 13, September, 1981, pp 181-188.

[19] A. Lew, "On the Emulation of Flowchart by Decision Tables," Comm. ACM, Volume 25, No. 12, December, 1982, pp 895-905.

[20] K. London, "Decision Tables," N.Y.: D. Van Nostrand Co. Inc, 1972.

[21] J. Metzner and B. Barnes, "Decision Tables Language and Systems," N.Y.: Academic Press, 1977.

[22] B. McMullen, "Structured Decision Tables," SIGMOD Notices, Volume 19, No. 4, April, 1984, pp 34-43.

[23] M. Montalbano, "Decision Tables" Palo Alto CA: Science Research Associates, 1974.

[24] B. M. E. Moret, "Decision Trees and Diagrams," ACM Computing Surveys, Volume 14, Number 4, pp. 593-624, Dec., 1982.

[25] H. Parde, "A Computational Approach to Approximate and Plausible Reasoning with Applications to Expert Systems," IEEE Trans. on Pattern Analysis and Machine Intelligence, Vol PAMI-7, No. 3, May 1985.

[26] F. Pereira, "C-Prolog User's Manual," Dept. of Architecture, University of Edinburgh, Dec., 1983.

[27] R. Pressman, "Software Engineering: A Practitioner's Approach," McGraw-Hill Software Engineering and Technology Series, 1982.

[28] K. Reilly, W. Jones, J. Barrett, A. Salah, E. Strand, E. Autry, and P. Rowe, "Software Development Studies: A Simulation Environment Perspective," ISETT 1984, Ann Arbor, Michigan, 1984.

[29] K. Reilly, A. Salah, P. Morgan, and P. Rowe, "Multiple Representation in a Language Driven Memory Model," Proceeding of the Conference on Linguistics in Humanities and Sciences, edited by Edwin Batteistilla, Published by Indiana Univ. Linguistic Club, August 1984.

[30] P. Rosenbloom, J. Laird, J. McDermott, A. Newell, and E. Orciuch, "R1-Soar: An Experiment in Knowledge-intensive Programming in Problem-solving Architecture," IEEE Trans. on Pattern Analysis and Machine Intelligence, Vol PAMI-7, No. 5, Sep 1985.

[31] A. Salah, K. Reilly, and C. C. Yang, "A Logic Programming Approach to Decision Table Definition and Implementation," Presented in the ACM Midsoutheast Conference in Gatlinburg, TN, November 1984.

[32] A. Salah, K. Reilly, and C. C. Yang, "A Restatement of Decision Table Problem in Logic Programming," Technical Report, UAB, C&IS, December, 1984.

[33] A. Salah and K. Reilly "A Reduction Methodology for a Differential Diagnosis Expert System," Proceedings of the Third Annual Computer Science Symposium on Knowledge-Based Systems: Theory and Applications, April 1986.

[34] A. Salah and C.C Yang "Rule-Based Systems: A Set-Theoretic Approach," Proceedings of the Third Annual Computer Science Symposium on Knowledge-Based Systems: Theory and Applications, April 1986.

[35] A. Salah, C. C. Yang, and K Reilly, "On Decision Table Properties, Representations, and Implementations," Submitted to IEEE trans. on Software Engineering, June, 1985.

[36] J. Ullman, "Principles of Database Systems," MD: Computer Science Press, Inc., 1983.

[37] J. Ullman, "Implementation of Logical Query Languages for Databases," Proceeding of 1985 ACM-SIGMOD Conference, June 1985.

[38] "UNH Prolog User Manual," N.H.: University of New Hampshire, July, 1983.

[39] S. M. Weiss and C. A. Kulikowski, "A Practical Guide to Designing Expert Systems," Rowman & Allanheld Publishers, 1984.

[40] R. Welland, "Decision Tables and Computer Programming," U.K.: Heyden & Son Ltd, 1981.

[41] G. Wiederhold, "Database Design," Second edition, N.Y.: Mcgraw-Hill, 1983.

[42] C. C. YANG, "Relational Databases", Prentice-Hall, Inc., Englewood Cliffs, N.J., 1986.

APPENDIX D

Rule-Based Systems: A Set-Theoretic Approach

Akram Salah and Chao-Chih Yang

This paper uses the set-theoretic approach, which is used to define a DT system, to represent a rule-based system. The purpose of this paper is to show that the set-theoretic DT definitions can apply to the terminology of rule-based systems. It also shows that when a rule-based system is represented in this form many problems can be solved easily.

The paper starts by specifying the known general structure of a single rule. Then the paper introduces a set-theoretic definition of basic concepts for a rule. Among those basic concepts are conditions, consequences, actions, and conclusions. Then a decision situation is defined as a relation between conditions and consequences. Many characteristics of a decision situation, such as a scheme, spaces, and domains are also defined. These characteristics are then used to define properties for a rule-based system. A discussion follows, showing that the set-theoretic approach solves the problems of a theoretical size limit and rule-system comparison. It also provides a systematic approach to the acquisition and construction of a rule-base system.

A version of this paper has been presented at, and is published in the proceedings of, The Third Annual Computer Science Symposium on Knowledge-Base Systems: Theory and Applications, Columbia, SC, March-April 1986.

RULE-BASED SYSTEMS: A SET-THEORETIC APPROACH

Akram Salah

Department of Computer & Information Sciences
The University of Alabama at Birmingham
Univ. Station, Birmingham, AL 35294

Chao-Chih Yang

Department of Computer Sciences
North Texas State University
Denton, TX 76203-3886

March 1986

ABSTRACT

Basic concepts of a Rule-based system are defined in a set-theoretic formulation and used as a formal approach to deal with other notions in such a system. These concepts comprise an extensive list including new perspectives into widely known concepts as well as new notions. Properties that characterize a rule-based system are discovered and formulated in terms of these concepts showing new insights for structural and behavioral study of constructing new systems as well as for analysis and evaluation of existing ones. A discussion is included to show how some problems in a rule-based system, such as theoretical size limits, automatic construction, and system evaluation can be easily developed in the new approach. The set-theoretic approach not only provides solid grounds for formulation of a syntax and the characteristics of a rule-based system, but also provides a systematic approach to design, construct, and analyze a rule-based system.

1. Introduction

Since experts tend to express most of their problem-solving techniques in terms of a set of if-then structures, known as production rules, a rule-based system is a method of choice for building knowledge intensive expert systems [11, 21]. Most of the existing rule-based systems are built for particular applications in an unstructured manner and and have no formal or theoretical basis. Some difficulties had been cited when such informal approaches are employed. First, a rule-based system has no theoretical size limits. The number of rules within a system grows, usually rapidly, creating a large system that is incomprehensible and difficult to handle. Secondly, human beings (experts or computer specialists) still must do all the work of defining, specifying, and following, to its logical consequence, each chain of condition and action. Thirdly, there are no criteria for analysis that can be used as grounds for evaluation of a rule-based system or for comparing a system with another [6, 27]. This is due to the lack of formal conceptualization of the process of constructing such systems from their basic components and the lack of definitions of properties that characterize the structure or behavior of a rule-based system.

This paper approaches the formalization of a rule-based system as part of a broader study of representation and management of systems that integrates programs, rules, and data models [1, 26]. The study is based on investigating a diverse of theories and practices: Production Systems [7, 21], Decision Tables [3, 13, 14, 15, 16, 28], Logic, Logic Programming [8, 10], Relational Database (Systems and Theory) [4, 9, 29] as well as many existing expert systems, to find analogies and differences in their expressive powers and make use of advantages in one model to overcome limitations in another. The final goal is to find a representation that is general enough to express a wide variety of rule-based systems being independent from any particular application such

that it can be formulated in an acceptable theoretical basis. Through this study, a set-theoretic definition for a rule-based system has been evolved from an integration of concepts from decision tables, logic, and databases systems. This definition is presented here.

In Section 2, basic concepts, such as a condition, a consequence, an action, and a conclusion, are defined in a formal way. Then, the notion of a decision situation is defined together with concepts, such as condition space, consequence space, decision situation scheme, rules, and many other related concepts. Finally, a rule-based system is defined. Illustrative examples are used to demonstrate these concepts when necessary.

In Section 3, properties that characterize a rule-based system are defined and formulated in set-theoretic forms in terms of the basic concepts. Among these properties are completeness, redundancy, and ambiguity.

Section 4 contains a discussion of the definitions and how they facilitate a formal view of many concepts in rule-based systems. We show how these definitions furnish theoretical size limits for a rule-based system. We also discuss using system properties to evaluate rule-based systems.

It is concluded that the set-theoretic approach is a powerful formal way to formulate a syntax of a rule-based system. Problems, such as size limits and construction methodology, are clearly formulated. The construction of a rule-based system can be partially automated when it is viewed in this approach.

2. Definitions for a Rule-Based System

We first informally overview the rule structure and then provide the formal set-theoretic definitions. A rule is typically expressed in the form:

$$C_1, C_2, \ldots\ldots, C_n \rightarrow Q$$

where C_i for $1 \leq i \leq n$ are atomic formulae called *conditions* and Q is called a *consequence* of the conditions. Such a rule is interpreted as if C_i for all $1 \leq i \leq n$ are simultaneously true, then Q is also true.

In the rest of this section, we provide a formal view of rules and other related concepts. Since conditions and consequences are the building blocks of rules, we define them first and then from these definitions we proceed to define a decision situation and a rule-based system.

2.1 A Condition

A *condition* is denoted by a pair (CS, CA) where CS is a *condition subject,* usually represented as a character string, and CA is a finite set of *alternatives* for this subject. A condition subject can be viewed as an attribute or a field-name while a set of alternatives can be seen as the domain of values for an attribute. At any time, a condition may have a state. A *state* of a condition (CS, CA) involves its subject CS and an alternative in CA.

For example, a condition may be (body-temperature, {high-fever, low-fever, normal}) in one diagnostic system, or (body-temperature, {96, 97, 98, 99, 100, 101, 102, 103, 104}) in another system.

Rule-based systems may have different representations, our definition covers most of them, in an abstract form. Specifically, some systems denote a condition state in a predicate form with a predicate name being the condition subject, CS, and an alternative, ca $\in$ CA, as the predicate argument, i.e., CS(ca). Other systems distinguish a condition state as an attribute followed by a value, i.e., ^CS ca. For example, a condition state may be represented in a system as "body-temperature(normal)", while the

same condition state is represented as "^body-temperature normal" in another system. It should be clear that differences between these two representations are due to the syntax of the languages but not in their semantics.

2.2 A Consequence

Similarly, a *consequence* is denoted by a pair (QS, QA) where QS is a *consequence subject,* usually represented by a character string, and QA is a finite set of *alternatives* for this subject. Also, a consequence subject, QS, can be viewed as an attribute or a field-name while a set of alternatives, QA, can be seen as the domain of values of an attribute. At any time, a consequence may have a state. A *state* of a consequence (QS, QA) involves its subject, QS, and an alternative in QA.

A consequence can be any one of two cases, a conclusion or an action, depending on its state. A *conclusion* is a consequence such that its state is a final one, i.e., no more processing is needed and its state can be outputted as a result. An *action* is a consequence such that its state is not a final state, i.e., there is more processing needed to reach a final state. Some systems use either a final conclusion or an intermediate conclusion instead of a conclusion or an action, respectively, but we prefer to use a conclusion and an action.

An example of a consequence, in a diagnostic system, would be: (leukaemia, {definite, probable, possible}) or (rheumatic-heart-fever, {positive, perform(Anti-striptolysin-A-titre)}). Note that in the second consequence, it is a conclusion if the first alternative is used or it is an action if the second alternative is used.

2.3 A Decision Situation

A decision situation is simply a situation involving a problem where a decision has to be made. In fact, when an expert describes his decision making criteria, a collection of situations is usually expressed where each situation has some rules expressing what to be done in different cases. For example, in a medical diagnostic system, a decision situation can be whether performing a specific test or not. Of course the decision depends on some conditions. If the conditions are satisfied, then a test is recommended and otherwise it is not.

We can say, in general, that: 1) a decision situation is characterized by a set of conditions and a set of consequences; 2) a decision situation contains a collection of rules, each rule describes what to be done in one condition state.

Formally, a *decision situation* is a triple (C, Q, R) where C is a set of conditions, Q is a set of consequences, and R is a set of rules. The following numbered remarks entail the elements in the triple.

1. The *set of conditions,* C= $\{C_1, C_2, ..., C_n\}$, is a set of n conditions for $n \geq 1$. Each condition C_i, for $1 \leq i \leq n$, is denoted by a pair (CS_i, CA_i), as defined in 2.1.

2. The *set of consequences,* Q= $\{Q_1, Q_2, ..., Q_m\}$, is a set of m consequences for $m \geq 1$. Each consequence Q_j, for $1 \leq j \leq m$, is denoted by a pair (QS_j, QA_j), as defined in 2.2.

3. The *condition space,* denoted by Cspace, is the complex product of the sets of condition alternatives in C, i.e.,

$$Cspace = *(CA_1, CA_2, ..., CA_n)$$

where * denotes complex product. An element in Cspace is represented by a string

of n condition alternatives.

4. The *consequence space,* denoted by Qspace, is the complex product of the sets of consequence alternatives in Q, i.e.,

$$\text{Qspace} = *(QA_1, QA_2, ..., QA_m)$$

where * denotes complex product. An element in Qspace is represented by a string of m consequence alternatives.

5. A *decision situation scheme* is a description of the structure of a decision situation (C, Q, R), and is denoted by:

$$CS_1, CS_2, ..., CS_n \rightarrow QS_1, QS_2, ..., QS_m.$$

The concept of a decision situation scheme is similar in many ways to other notions, such as a frame and a prototype, which are used in some existing rule-based systems [6, 27].

6. A *rule,* r, is defined as a mapping from the condition space to the consequence space. A rule is typically expressed in the form:

$$r: c \rightarrow q$$

where the left-hand side, $c \in$ Cspace and the right-hand side, $q \in$ Qspace.

7. In a decision situation (C, Q, R), R is a set of rules, r. Thus, R can be defined as a relation between the condition space and the consequence space. That is, R is a subset of the Cartesian product of Cspace and Qspace, i.e.,

$$R \subseteq x(\text{Cspace}, \text{Qspace})$$

where x denotes Cartesian product.

8. Since R does not necessarily involve all elements in Cspace, we define the *domain* of R in a decision situation as a subset of Cspace such that each element in the domain contributes as the argument in a rule in R. It is denoted by domain(R).

9. Some systems restrict consequences to be actions [7] in the form of calls to programs or procedures. In such systems if more than one action appear on the right-hand side of a rule, it is considered as an ordered chain (a sequence) of actions. Since this is a special case, we do not restrict our definitions to be only actions or only conclusions.

2.4 A Rule-Based System

A *rule-based system* is defined as a finite set of decision situations, each of which is defined as a triple as stated in 2.3. For a rule-based system, the following should be noted:

1. Condition sets in different decision situations in the same rule-based system are not necessarily disjoint, i.e., a condition may occur in more than one situation. Similarly, consequence sets are not necessarily disjoint.

2. Sometimes a consequence (or a set of consequences) in a decision situation appears as a condition (or a set of conditions) in another decision situation in the same rule-based system. In this case, a consequence is interpreted as an action, which means that there are more processings needed to reach a final conclusion.

3. A decision situation does not necessarily have a name since it can be distinguished by its triple (C, Q, R). However, sometimes, especially in a system, naming may simplify searching and/or matching processes.

3. Properties of a Decision Situation

We refer to a decision situation here as a situation and denote it by its triple, i.e., (C, Q, R).

3.1 An Empty Decision Situation

A decision situation (C, Q, R) is said to be *empty* if R, the set of rules, is the empty set. An empty decision situation still contains some information such as a set of conditions, a set of consequences, the condition space, the consequence space, and a scheme. An empty decision situation represents the structural characteristics of a situation but not its behavioral ones. Some systems use the term *specification* for what we called here structural characteristics.

3.2 A Complete Decision Situation

A *complete* decision situation is one in which the relation R is complete, i.e., for every element in the condition space of the situation there is an image in the consequence space. Thus, a situation (C, Q, R) is said to be complete if

$$\text{domain(R)} = \text{Cspace}.$$

The size of a condition space in a situation can simply be calculated by multiplication of the cardinalities of the sets of condition alternatives in the situation, i.e.,

$$|\text{domain(R)}| = |\text{Cspace}| = \prod_{i=1}^{i=n} |CA_i|.$$

Note that an empty decision situation and a complete one form theoretical size limits for a decision situation. We will collaborate on this point in our discussion section.

3.3 A Redundant Decision Situation

A situation (C, Q, R) is said to be *redundant* if a rule occurs more than once in R. Since R is defined as a set, which contains no duplicates by definition, redundancy is not allowed. However, a rule may practically appear more than once in a rule-based system during construction by mistake or as a result of updating a rule such that it coincides exactly with an existing one. If, by any means, redundancy has been detected in any instance of a rule-based system, it should be eliminated.

Redundancy has a big effect on structural and behavioral characteristics of a rule-based system. It is easy to define, however, its detection is not that simple since a deeper view of rule implications complicates its formulation. To clarify this point, we informally differentiate between two levels of redundancy, *mechanical* (or *structural)* level and *logical* (or *behavioral)* level.

Mechanical redundancy can be detected during rule construction or updating. It happens when the same rule explicitly occurs more than once in the same instance of a decision situation. This form is simply formulated and can easily be detected and eliminated. A supervising program can be employed to check the rule duplicates during construction and updating.

Logical redundancy occurs in a rule-based system when a rule explicitly exists in a situation while it can be derived from a set of other rules existing in the same system. This form of redundancy is more complicated than the simple mechanical one since it is not local in one situation and its detection depends on inference rules that are not defined yet. This point is not completely solved in our definitions, though the formal approach provides a strong grounds to its formulation and investigation.

3.4 An Ambiguous Decision Situation

First, we do not consider ambiguity as an unfavorable property, but we view it as a property. The term is used here in a fashion different from its use in languages where it implies inclarity. Informally speaking, a situation is said to be ambiguous if it has more than one consequence with respect to a given set of conditions.

In our opinion, ambiguity is a property that may exist in a decision-making problem. Thus, a rule-based system should be able to furnish a representation such that if it exists in the real life problem in one way, it has to be represented in the rule-based system in the same way. Ambiguity is improper only if a rule-based system is not consistent with the specification of the real-life decision-making problem, i.e., the real life problem is non-ambiguous but the rule-based system is ambiguous, or vice versa.

Formally, we say that a decision situation (C, Q, R) is *non-ambiguous* if the relation R is a function, and is *ambiguous* otherwise. A function (a one-one or many-one relation) has a unique image for each of its arguments.

Note that this definition covers ambiguity as a property in a situation. It should be clear that there is a difference between ambiguity as a property in building a rule-based system and ambiguity in processing a rule-based system. The latter case is one where the conditions at some point of processing are not sufficient to derive a consequence. This is a case in which methods of inexact reasoning could be employed. These methods may possibly result multiple consequences due to approximation techniques but not due to ambiguity in the rule-based system itself.

4. Discussion

In Sections 2 and 3 of this paper, we presented a set-theoretic formulation of a syntax of a rule-based system, concepts, and characteristics. In this section, we discuss the significance of such a formal approach to the conceptualization of rule-based systems. We also discuss how the definition provides a more compatible relationship between rule-based systems and data systems. We form the discussion as remarks for easier reading.

1. It is easy to see that the set-theoretic formulation of conditions and consequences such that each of them involving a domain (the set of alternatives) provides a better view of conditions and consequences themselves as well as all other concepts derived from them. In a rule-based system, if we have two conditions C_1 and C_2 defined as shown here, we definitely can answer the question of whether they are equivalent or not by comparing not only their subject names but also their sets of alternatives.

2. The concept of defining a scheme for a decision situation which represents its structure provides a conceptual level characterization of a rule-based system. Now a rule-based system can be studied on a conceptual (structural) level as a set of situation schemes without having to follow any specific values for conditions and actions.

3. The concept of a decision situation provides a reliable form of modularization of a rule-based system. Each situation can be viewed as a module. When a rule-based system is processed, only the decision situations under consideration can be activated or looked-up.

4. The set-theoretic formulation provides theoretical size-limits for a rule-based system. A rule-based system is a finite set of decision situations. Each decision situation is

limited to either empty or complete. This furnishes size limits for a rule-based system that can be clearly determined from each situation's condition and consequence definitions. Evenmore, completeness and emptiness are formulated in a way that can be easily checked automatically.

5. The construction of a rule-based system can be expressed systematically. This means that programs can be employed to help the user to build his/her rule-based system. In fact, we have already developed a program that provides a computer-aided construction of a rule-based system. The program asks the user for his situations, conditions, and consequences. Then, for each condition and consequence, the program accepts a subject and a set of alternatives. If the user defines any rule, the program checks for validity of the condition and consequence states. A user may also ask queries about the existing rules, conditions, or consequences.

6. The definitions provide grounds for comparison beyond the naive view of a number of rules or the structure of one or more rules. This means that when two systems are viewed through this definition they can be studied and analyzed in a more precise form.

5. Conclusion

The paper provides a set-theoretic view of a rule-based system and its characteristics, and shows that this view provides new insights for study and analysis of such systems. Many problems of existing rule-based systems can be resolved, others are precisely characterized when the set-theoretic approach is adopted. The fact that the definitions are set-theoretic adds the advantage of being compatible with database systems which are also defined in set-theoretics.

Future extensions for this work include using these definitions as a basis to establish criteria for: knowledge acquisition since formal construction methods can be established; computer analysis of an existing system using properties defined here; and modularization of a system using the definition of a situation. This can be a start of changing the approach to rule-based systems from being practices for particular applications to be a branch of computer science with established theoretical foundation.

6. REFERENCES

[1] H. Berghel, "*Simplified Integration of Prolog with RDBMS,*" Data Base, Volume 16, Number 3, Spring 1985.

[2] H. Cantrell, J. King, and F. King, "*Logic Structure Tables,*" Communications of ACM, Volume 4, Number 6, pp. 272-275, June, 1960.

[3] "*A Modern Appraisal of Decision Tables*" A CODASYL Report, N.Y.: ACM, July, 1982.

[4] E. Codd, "*Relational Databases: A Practical Foundation for Productivity,*" Comm. ACM, Volume 25, No. 2, February, 1982.

[5] C. Date, "*An Introduction to Database Systems,*" 4th Edition, Addison Wesley Inc., CA, 1985.

[6] R. Fikes and T. Kehler, "*The Role of Frame-Based representation in Reasoning,*" Comm. of ACM, Vol 28, No. 9, September 1985, pp. 904-920.

[7] C. Forgy, "*OPS5 User's Manual,*" Department of Computer Science, Carnegie-mellon University, Pittsburgh, PA 15213.

[8] H. Gallaire and J. Minker "*Logic and Databases,*" N.Y.: Plenum Press., 1977.

[9] H. Gallaire, J. Minker, and J. Nicolas, "*Logic and Databases: A Deductive Approach,*" ACM Computing Surveys, Volume 16, Number 2, pp. 153-185, June, 1984.

[10] M. Genesereth and M. Ginsberg, "*Logic Programming,*" Comm. of ACM, Vol 28, No. 9, September 1985, pp. 933-941.

[11] F. Hayes-Roth, "*Rule-Based Systems,*" Comm. of ACM, Vol 28, No. 9, September 1985, pp. 921-932.

[12] P. Horstmann, "*Expert Systems and Logic Programming for CAD,*" VLSI DESIGN, November, 1983, pp 37-46.

[13] R. Hurley, "*Decision Tables in Software Engineering,*" N.Y.: Von Nostrand Reinhold company Inc., 1983.

[14] K. London, "*Decision Tables,*" N.Y.: D. Van Nostrand Co. Inc, 1972.

[15] J. Metzner and B. Barnes, "*Decision Tables Language and Systems,*" N.Y.: Academic Press, 1977.

[16] M. Montalbano, "*Decision Tables*" Palo Alto CA: Science Research Associates, 1974.

[17] H. Parde, "*A Computational Approach to Approximate and Plausible Reasoning with Applications to Expert Systems,*" IEEE trans. on Pattern Analysis and Machine Intelligence, Vol PAMI-7, No. 3, May 1985.

[18] R. Pressman, "*Software Engineering: A Practitioner's Approach,*" McGraw-Hill Software Engineering and Technology Series, 1982.

[19] K. Reilly, W. Jones, J. Barrett, A. Salah, E. Strand, E. Autry, and P. Rowe, "*Software Development Studies: A Simulation Environment Perspective,*" ISETT 1984, Ann Arbor, Michigan, 1984.

[20] K. Reilly, A. Salah, P. Morgan, and P. Rowe, "*Multiple Represention in a Language Driven Memory Model,*" Papers on Computational and Cognitive Science, edited by Edwin Batteistilla, Published by Indiana Univ. Linguistic Club, August 1984, 87-94.

[21] P. Rosenbloom, J. Laird, J. McDermott, A. Newell, and E. Orciuch, "*R1-Soar: An Experiment in Knowledge-Intensive Programming in a Problem-Solving Architecture,*" IEEE trans. on Pattern Analysis and Machine Intelligence, Vol PAMI-7, No. 5, September 1985.

[22] A. Salah, K. Reilly, and C. C. Yang, "*A Logic Programming Approach to Decision Table Definition and Implementation,*" Presented in the ACM Midsoutheast Conference in Gatlinburg, TN, November 1984.

[23] A. Salah, K. Reilly, and C.C Yang, "*A Restatement of Decision Table Problem in Logic Programming,*" Technical Report, UAB, C&IS, December, 1984.

[24] A. Salah, C. C. Yang, and K. Reilly, "*On Decision Table Properties, Represenations, and Implementations,*" Submitted to IEEE trans. on Software Engineering, June, 1985.

[25] A. Salah and K. Reilly, "*A Reduction Methodology for a Differential Diagnosis Expert System,*" Proceedings of the Third Annual Computer Science Symposium on Knowledge Based Systems: Theory and Applications, April 1986.

[26] A. Salah, K. Reilly, and C.C Yang, "*An Integration of a Knowledge Representation in Rules into a Relational Database System,*" Technical Report, UAB, C&IS, Dec 1985, and Submitted to IEEE trans. on Pattern Recognition and Machine Intelligence, March 1986.

[27] S. M. Weiss and C. A. Kulikowski, "*A Practical Guide to Designing Expert Systems,*" Rowman & Allanheld Publishers, 1984.

[28] R. Welland, "*Decision Tables and Computer Programming,*" U.K.: Heyden & Son Ltd, 1981.

[29] C. C. YANG, "*Relational Databases*", Prentice-Hall, Inc., Englewood Cliffs, N.J., 1986.

APPENDIX E

A Reduction Methodology for a Differential Diagnosis Expert System

Akram Salah and Kevin Reilly

This paper uses the approach of DTs and RDBS to extend a rule representation that is used in differential diagnosis expert systems. The paper emphasizes the role of a management program in an expert system. It deals with a problem addressed by Weiss & Kulikowski in their book about expert systems. Instead of using a large number of simple rules, a reduction algorithm can be added to the management system such that it applies a single generalized rule to the given conditions. This method decreases the number of rules stored in an expert system.

The paper first defines the problem and characterizes its use through a user-program dialog. Then the paper shows, through an example, how the reduction methodology applies. A general reduction algorithm is then introduced, together with a set of DTs that are employed by the algorithm. Finally, there is a discussion showing the advantages of the reduction methodology compared to the simple, naive, approach.

This paper has been presented at, and published in the proceedings of, The Third Annual Computer Science Symposium on Knowledge-Base Systems: Theory and Applications, Columbia, SC, March-April 1986. Also this paper has been selected by the conference referees to be considered for publication in a special issue of the International Journal of Man-Machine Studies, edited by Prof. J. C. Bezdek.

A REDUCTION METHODOLOGY FOR A DIFFERENTIAL DIAGNOSIS EXPERT SYSTEM

Akram Salah
Department of Computer & Information Sciences

and

Kevin D. Reilly
Department of Computer & Information Sciences
Department of Biostatistics and Biomathematics

The University of Alabama at Birmingham
Univ. Station, Birmingham, AL 35294

ABSTRACT

Knowledge representation issues and problems are being studied in our lab in a Prolog-centered programming and database environment. Work presented in this paper reflects part of our interest in systems that combine rules with relational databases (rdb) in a compatible form in this environment. In a previously submitted paper, we present a comprehensive data and rule management system in which production rules as well as data are stored and manipulated. Here, we concentrate on rule management in a context of a differential diagnostic expert system characterized by the ability to establish a diagnosis on the basis of a subset of a collection of symptoms. Our approach to the problem provides a more effective solution than the use of a simple production rule representation.

Production systems are simple structures used to describe an expert's reasoning rules, typically expressed in the form

$$A_1 \,\&\, A_2 \,\&\, ...\&\, A_n \rightarrow D$$

where $\{A_1,...,A_n\}$ is a set of conditions and D is concluded if the conjunction of conditions holds. In the medical diagnosis context, conditions are mainly symptoms or observations, and the conclusion is a specific disease or class of diseases. In many differential diagnostic systems, decision structures more complicated than basic production rules may arise. This point is made clearly in expert systems development treatise by Weiss and Kulikowski, whose ideas have also contributed to our solution approach.

The case of concern to us in this paper is one in which a conclusion is derived from a subset of the conditions, specifically, when any k out of n observations (or symptoms), where $k < n$, establish the diagnosis. If we used a simple solution, e.g., represented each possible subset in a single production rule, then a large number of

rules would ensue. An alternative method is what we call here the "reduction" method. It uses an algorithm together with a set of decision tables to reduce a larger diagnostic problem successively to smaller ones. The algorithm is defined for the general case and examples provide illustrations of its use.

An outline of the paper follows. First, the problem is characterized with respect to its input-output transformation through presentation of a sample dialog between user and system.

Second, a set of decision tables is introduced for a specific diagnostic problem. These together with the reduction algorithm provide a concrete demonstration of application of the algorithm and help motivate the next part of the paper in which the reduction algorithm is formulated more abstractly, for the general case.

Third, the reduction algorithm is defined recursively. The input is a subset of observations, determined by the user but with the number of observations at the start selected such that if all observations (symptoms) are true (positive) the diagnosis is established and the algorithm terminates. If all observations are negative, it is possible in some, but not all, cases to terminate with the diagnosis rejected. In those cases which we cannot reject and in cases when some observations are true and others false, the algorithm reduces the problem to a simpler one of the same type and proceeds recursively to a solution.

Fourth and finally, advantages of the methodology are presented. The method decreases the number of rules expressed in a diagnostic production system relative to the naive formulation of the problem; the steps needed to do this (currently, partially implemented) may be viewed as a form of meta-level processing of the rule system. The efficiency of the production system is increased, again relatively, since it almost invariably permits derivation of conclusion from examination of only a subset of the observations. If the problem description evolves over time into a bigger one, as is frequently the case with diagnostic problems, needed changes are minor: the algorithm itself does not change though the tables used may. The methodology does not depend on the number of observations involved in the problem description nor on any other attribute of the specific diagnostic situation.

Though several of our ideas are motivated by medical diagnosis applications and we freely utilize the terminology from this field in our description, the basic methodology is not restricted to any particular field of application. The development of the algorithm and the methodology that surrounds it contributes directly to our understanding of expert systems for differential diagnosis, and indirectly to a potential theoretical basis for expert systems, to include exploitation of meta-knowledge and problem reduction methodologies.

INTRODUCTION

Much recent research focuses on computer systems which facilitate methodologies that simulate experts' decision-making strategies [7, 21, 22]. The most popular current way to create such systems is to incorporate large amounts of domain-specific knowledge acquired from experts. Other alternatives, competing and complementary, exist and often include a general knowledge component. A recent effort, the R1-SOAR system [17] involves such a combination of domain-dependent knowledge-intensive methodology, embedded within a general domain-independent problem solving capability. The methodology permits hand-crafted and automatic elaboration of production rules in the face of poor performance of weak methods to increase efficiency. A form of generalization is implicit in the process of creating new rules in that a so-called chunking activity involved in the creating act is focused on sub-goals so that goals which share these subgoals are the actual (generalized) learners.

In this paper we consider one fundamental part of a system which shares several design aspects with the work just reviewed. Our whole system is incorporated into a Prolog [1, 3, 9] framework with an embedded relational database system, together with some meta-language constructs which constitute a management system that stores simple production rules along with conventional data of an rdb [22]. The management system allows such operations as enabling and disabling parts of rules, like in the algorithm presented below. Efficiency and generalization are served through such mechanisms. This paper centers on a part of the overall apparatus under development, specifically, on that part which incorporates the reduction algorithm stated in the title. The approach we take here starts with simple production rules and generalizes them to the case of primary interest.

In knowledge intensive expert systems, since experts often express their

decision-making processes in sets of if-then rules, rule bases are devised [6]. Rules are represented in a structure typically in the form:

$$A_1 \& A_2 \& ..... \& A_n \rightarrow \text{C}.$$

Such a rule is interpreted as: if A_1 and A_2 and ....and A_n are true, simultaneously, then, consequently, C is true. The left-hand side of a rule contains a conjunction of atoms called *conditions* and the right-hand-side is called a *conclusion* or a *consequence* [20].

If an expert expresses his decision-making process explicitly as a collection of if-then rules, then each of them can be represented directly as stated above. A problem arises if an expert expresses rules in a less explicit way, or in some form such as a function over a set of rules. Then it is the responsibility of the rule acquirer, whether man or machine, to decide upon an explicit representation or to provide management (control) code to process the rules.

PROBLEM

The problem arises in a differential diagnosis expert system where conditions are either symptoms, observations, or test-results gathered by a physician to be used to derive a conclusion, which in this case is a disease or a class of diseases. Rules used to derive such conclusion would typically be in the form:

$$O_1 \& O_2 \& .....\& O_n \rightarrow \text{D}.$$

where each O_i, for $1 \leq \text{i} \leq \text{n}$, is an observation and D is a disease or class of diseases. Note: from here on in this paper, we refer to any atom on the left-hand side of a rule as an observation (an observation can be a test result or a symptom) and the right-hand side as a disease. So this rule is read as if all observations, 1 through n, exist simultaneously in a patient, then this case can be diagnosed as D.

Our problem arises when a group of rules is expressed within a single if-then statement, with a subset operation applied over the group. Our particular concern is: given a set of n observations and a conclusion D such that if any k-subset of observation out of n holds, $k \leq n$, then D can be concluded, i.e.,

$$\text{Any k observations out of } \{O_1 \ \& \ O_2 \ \& \ .....\& \ O_n\} \rightarrow \text{D}.$$

Actually, this is a generalization of a rule application. The special case in which k=n defines the "normal" rule structure, i.e., the case where all the conditions have to be satisfied to derive the consequence. A formal view of this problem exhibiting still further differences between this formulation and the usual one is as follows. In a rule system, there is a set of conditions for each decision situation, each condition having a domain of values such that the left-hand-side of any rule represents an element in the cartesian product of the domains of these conditions [20]. The case here may be conceptualized as having one condition with one domain of observations, say O with length n, such that if any k-subset of O with $k < n$ occur simultaneously, then the diagnosis is established. This expresses a set of rules each one having a condition part, $c \in O^k$, where $k < n$, and the same consequence, D, which is the disease under consideration.

How the user and system interact is fairly easy to comprehend. After invoking the Prolog system and relevant loading operations, the system presents a series of questions to ascertain names for the tests being conducted, if these are not already known to the system. Further prompting requests results for each test in the battery. Properties include simple error detection, e.g., when a test claimed to have been performed is irrelevant to diagnosis. More interesting is when the information provided by tests is insufficient to confirm or reject the diagnosis. The system notifies the user of this state of affairs and continues seeking additional information.

Give me a test you performed: Arthralgia
...
What is the result of Arthralgia (p=positive, n=negative) p
...
** Diagnosis is POSITIVE **
** Chronic polyarthritis > 6 wks is a significant factor **
...
What is the result of Synovial fluid inflammatory (p=positive, n=negative) p
...
These results are not sufficient to establish a diagnosis.
...
What is the result of Subcutaneous nodules (p=positive, n=negative) p
...
** Diagnosis is NEGATIVE **
** Two positive symptoms are noted **
...
Do you wish a trace of this dialog ? No
...

To represent this situation within an expert system, we have several options. We examine two of them to set the stage for subsequent comparison:

1) The single rule (any k out of n) is re-expressed as a set of simple rules. Each such simple rule contains k observations on its left-hand side and D on its right-hand side. Needless to say, the resulting number of rules consumes much memory space, complicates the search when the system is applied, and reduces the efficiency of the system.

2) The production system conceptualization is extended such that if the "any k out of n" formulation is expressed, it can be handled automatically.

Note that this problem, expressed as we have done here, differs from those representations of "inexact reasoning" or uncertainty [14, 17] in which subsets of conditions are used to derive a consequence, e.g., probabilistically, fuzzily, or using weighting schemes.

METHOD

We denote a problem as "a k/n diagnostic problem" when diagnosis is dependent on n observations such that if any k out of them are found to exist in a case, then the diagnosis is established. In one of the examples described in [23] diagnosis of rheumatic diseases such as Mixed Connective Tissue Disease (MCTD) involves ten observations. If any 4 out of these 10 exists in a patient, then he definitely has a rheumatic disease. Using the introduced terminology, we say that this is a 4/10 diagnostic problem. Due to a postulated need to track the decision-making process or due to constraints on the number or choice of observations, this problem cannot be solved by a simple loop seeking k successes over the n observations, though such a simple case can be viewed as a limiting case of the problem in hand. A management system employs analysis such that the simple cases can be derived from a more complex ones, reminiscent of the R1-SOAR system [17].

To illustrate the method, first, we use a specific example of a 4/7 diagnostic problem. Then, we generalize the method and evaluate it relative to other methods.

T1	p	p	p	p	p	p	p	p	n	n	n	n	n	n	n	n
T2	p	p	p	p	n	n	n	n	p	p	p	p	n	n	n	n
T3	p	p	n	n	p	p	n	n	p	p	n	n	p	p	n	n
T4	p	n	p	n	p	n	p	n	p	n	p	n	p	n	p	n
Diagnosis Established:	x															
Diagnosis Rejected:																x
Further tests:		1	1	2	1	2	2	3	1	2	2	3	2	3	3	

Table 1: used for any 4/7 problem.

In this table test results are p=positive, n=negative.

To solve the diagnostic problem

1. pick any 4 observations

2. name them temporarily T1, T2, T3, and T4

3. check Table 1, with the results of these 4 observations
   (p = positive or n = negative).

4. the possible cases are:

   a- if result of all 4 are p, then the diagnosis is definitely established;
   b- if only 3 are p, then we need to check 1 more out of the remaining 3;
   c- if only 2 are p, then we need to check 2 more out of the remaining 3;
   d- if only 1 is p, then we need to check 3 more out of the remaining 3;
   e- if all are n, then, hypothetically, we need to check 4 out of the remaining 3; which is impossible, so reject the diagnosis.

Case a is self explanatory, there are 4 observations, all of them exist, thus the diagnosis is established. In case e, we can reject the diagnosis since total number of observations are 7, 4 of them already checked and failed, the remaining observations are 3. Since to establish a diagnosis 4 observations need to exist, then it is impossible to establish a diagnosis from this situation (4/3).

Cases, b, c, or d, a diagnosis is not established due to insufficiency of the input information. Instead of restarting the problem, we can define a new *reduced* problem such that we check only the remaining observation. Now we need to check either 1/3, in case b, 2/3, in case c, or 3/3 in case d.

The method either establishes a diagnosis from the information provided or uses the information to reduce the problem to a smaller problem of the same nature. The new problem can be solved recursively by the same methodology. In preparation for the generalization to the general k/n case, the reader is asked to reflect on the 3/7 diagnosis problem.

THE GENERAL ALGORITHM

To solve a k/n diagnostic problem using the reduction methodology:

1. pick any k symptoms
2. name them temporarily T1, ..., Tk
3. check decision table(k) [see below], with results for the k symptoms.
4. the output of the decision table is δ.
5. The problem now is δ/R where R= n-k.
6. For any δ/R diagnostic problem:

a) if $\delta = 0$	Diagnosis is POSITIVE; terminate.
b) if $\delta > R$	Diagnosis is NEGATIVE; terminate.
c) if $\delta \leq R$	go to step 1 (with k=δ, n=R) for further reduction.

The set of tables that follow depict the situation in an over-simplified form to make it easier to focus on the steps of the reduction method. In realistic cases actions may involve reports back to the user on the rules that are fired, auxiliary calculations, e.g., of a statistical nature, or other options. In such cases a table action portion would include additional information besides the number of remaining tests that are depicted in this set of tables. It should be noted that use of tables to describe the algorithm does not necessarily imply that implementation by tables is mandated. Indeed, we have already alluded to the possibility that in a very special case a simple looping construction might suffice. If tables are used in the implementation, they need not always be stored, i.e., there are cases in which they can be generated. More complicated situations exist in which human guidance is required.

T1	P	P	P	P	P	P	o	o	N	N	N	N	N	N	N	N
T2	P	P	P	P	N	N	o	o	P	P	P	P	N	N	N	N
o	o	o	o	o	o	o	o	o	o	o	o	o	o	o	o	o
Tk	P	N	P	N	P	N	o	o	P	N	P	N	P	N	P	N
δ	0	1	a	b	c	d	o	o	t	u	v	w	x	y	z	k

Table(k): used for any k/n problem. δ is the number of further tests.
The numbers, a, b, ..., are defined once k is known. P is positive, etc.

o
o
o

T1	P	P	P	P	P	P	P	P	N	N	N	N	N	N	N	N
T2	P	P	P	P	N	N	N	N	P	P	P	P	N	N	N	N
T3	P	P	N	N	P	P	N	N	P	P	N	N	P	P	N	N
T4	P	N	P	N	P	N	P	N	P	N	P	N	P	N	P	N
δ	0	1	1	2	1	2	2	3	1	2	2	3	2	3	3	4

Table(4): used for any 4/n problem.

T1	P	P	P	P	N	N	N	N
T2	P	P	N	N	P	P	N	N
T3	P	N	P	N	P	N	P	N
δ	0	1	1	2	1	2	2	3

Table(3): used for any 3/n problem.

T1	P	P	N	N
T2	P	N	P	N
δ	0	1	1	2

Table(2): used for any 2/n problem.

T1	P	N
δ	0	1

Table(1): used for any 1/n problem.

COMMENTARY

We may cite several advantages of the reduction methodology: 1) guaranteed recursive reduction until a solution is reached; 2) the number of rules to be checked is less than using the naive rule approach, e.g., in the 4/18 case (see Table 2) where an average of 15 rules is matched to find a diagnosis, compared to 1530 and 36720 in the naive cases; 3) tables shown above can be used for any diagnostic problem k/n, regardless of the value for n; 4) the growth of number of rules in limited since all the decision tables are complete [21] and thus there is no possibility of adding rules to any of them, however the number of tables employed vary from one problem to another depending on the value of k in a problem.

As can be seen, the number of rules in the reduction method depends on the length of the subset that establishes the diagnosis rather than the length of the domain of observations. This is an important property of the reduction methodology since in naive approaches the number of rules grow exponentially with the length of the set of observations. The following table shows the number of rules that can be generated if sets of simple rules are used. In Table 2, we show the number of rules (and its growth) in the reduction method for a 4/n diagnostic problem, compared with the number of rules in two of the simple rule generation approaches. We assume that simple rule generation approach uses either combinations or permutations of conditions.

Problem ID	using Permutation	using Combination	using Reduction
4/10	5020	210	31
4/18	73440	3060	31
4/35	1256640	52360	31

Table 2: number of rules used to build a knowledge base

In [23], it is stated that a problem similar to what we have been intimating was detected during implementing an expert system for diagnosis for rheumatic diseases. As the expert system evolved, the number of its (physician) users increased and consequently the number of observations known to the system increased. It is mentioned that the expert system started with a 4/10 diagnostic problem and was extended to 4/18, and eventually to 4/35. A simple treatment of the problem, see Table 2, would make the number of rules increase exponentially with any increase of the number of observations.

A final point to be noted about the reduction method is that it does not depend on any particular application. The algorithm was developed for an expert system for differential diagnosis of rheumatic diseases, but it can be used in any other rule representation of the same nature.

ENVIRONMENT

The system that we employ for representing the methodology of this paper is based on an extension of a previously defined system called EXPRD [22]. EXPRD (EXtended Prolog Rule Data system) is an integration of a Prolog system, a relational database system and a decision table system. EXPRD is designed to be independent from any particular application. It stores and manipulates program procedures, rule systems, and data tuples in one environment, managed by the same management system, which assures compatibility among its components.

This system is used to store or generate decision tables such as those appearing in this paper. Prolog programs expressing the reduction algorithm are part of the management program. Sets of observations are stored in the system as database relations. An interactive dialog prompts the user to provide the proper information for the diagnostic problem and invokes the reduction algorithm. If a decision is reached, the

program advises the user whether the diagnosis is established or rejected. If the information is not sufficient to establish the diagnosis, the program prompts the user to provide more information.

A general philosophy in dealing with rule systems emerges from our methodology: to employ a management system in which rules are dealt with as one of the components. Such management system can be viewed as a meta-rule program that helps a user (or an expert-system administrator) to build, manipulate, query, and analyze a rule system.

CONCLUSION

In this paper we discuss a reduction methodology for differential diagnosis expert system. Though the method is self-contained in the sense that it solves a well-defined problem, a broader view of the situation sees it as part of an environment for rule management. The environment conceptualization emphasizes use of meta-level processing to manipulate rule-like representations. Given "a k/n diagnostic problem", an extended form of rule, the management program is designed to generate a set of simple rules or employ the reduction methodology to reduce the problem to a smaller problem of the same nature. Employing management programs on the meta-level facilitates a global view for expert systems, allowing operations such as generation, reduction, or analysis of rules.

REFERENCES

[1] M. Bruynooghe, "Prolog in C for Unix Version 7: A Reference Manual," Belgium: Katholieke Univ. 1980.

[2] H. Cantrell, J. King, and F. King, "Logic Structure Tables," Communications of ACM, Volume 4, Number 6, pp. 272-275, June, 1960.

[3] W. Clockson and C. Mellish, "Programming in Prolog," Berlin, Germany: Springer-Verlag, 1981.

[4] "A Modern Appraisal of Decision Tables" A CODASYL Report, N.Y.: ACM, July, 1982.

[5] R. Fikes and T. Kehler, "The Role of Frame-Based representation in Reasoning," Comm. of ACM, Vol 28, No. 9, September 1985, pp. 904-920.

[6] F. Hayes-Roth, "Rule-Based Systems," Comm. of ACM, Vol 28, No. 9, September 1985, pp. 921-932.

[7] R. Hurley, "Decision Tables in Software Engineering," N.Y.: Von Nostrand Reinhold company Inc., 1983.

[8] R. Kowalski, "Logic for Data Description," in [14], pp 77-103.

[9] R. Kowalski, "Logic for Problem Solving," Artificial intelligence series, The Computer Science Library, N.Y.: Elsevier North Holland Inc., 1979.

[10] K. London, "Decision Tables," N.Y.: D. Van Nostrand Co. Inc, 1972.

[11] J. Metzner and B. Barnes, "Decision Tables Language and Systems," N.Y.: Academic Press, 1977.

[12] B. McMullen, "Structured Decision Tables," SIGMOD Notices, Volume 19, No. 4, April, 1984, pp 34-43.

[13] M. Montalbano, "Decision Tables" Palo Alto CA: Science Research Associates, 1974.

[14] H. Parde, "A Computational Approach to Approximate and Plausible Reasoning with Applications to Expert Systems," IEEE Trans. on Pattern Analysis and Machine Intelligence, Vol PAMI-7, No. 3, May 1985.

[15] K. Reilly, W. Jones, J. Barrett, A. Salah, E. Strand, E. Autry, and P. Rowe, "Software Development Studies: A Simulation Environment Perspective," ISETT 1984, Ann Arbor, Michigan, 1984.

[16] K. Reilly, A. Salah, P. Morgan, and P. Rowe, "Multiple Representation in a Language Driven Memory Model," Papers on Computational and Cognitive Science, Indiana U. Linguistics Club, edited by Edwin Batteistilla, Published by Indiana Univ. Linguistic Club, August, 1984, 87-94.

[17] P. Rosenbloom, J. Laird, J. McDermott, A. Newell, and E. Orciuch, "R1-Soar: An Experiment in Knowledge-intensive Programming in Problem-Solving Architecture," IEEE Trans. on Pattern Analysis and Machine Intelligence, Vol PAMI-7, No. 5, Sep 1985.

[18] A. Salah, K. Reilly, and C. C. Yang, "A Logic Programming Approach to Decision Table Definition and Implementation," Presented in the ACM Midsoutheast Conference in Gatlinburg, TN, November 1984.

[19] A. Salah, K. Reilly, and C. C. Yang, "A Restatement of Decision Table Problem in Logic Programming," Technical Report, UAB, C&IS, December 1984.

[20] A. Salah and C. C. Yang, "Rule-Based Systems: A Set-Theoretic Approach," Proceedings of the Third Annual Computer Science Symposium on Knowledge Based Systems: Theory and Application, April 1986.

[21] A. Salah, C. C. Yang, and K Reilly, "On Decision Table Properties, Representations, and Implementations," Technical Report, UAB, C&IS, June 1985. Submitted to IEEE trans. on Software Engineering, January 1986.

[22] A. Salah, C. C. Yang, and K Reilly, "An Integration of a Knowledge Representation in Rules into a Relational Database System," Technical Report, UAB, C&IS, Dec 85, Submitted to IEEE trans. on Pattern Matching and Machine Intelligence, March 1986.

[23] S. M. Weiss and C. A. Kulikowski, "A Practical Guide to Designing Expert Systems," Rowman & Allanheld Publishers, 1984.

[24] R. Welland, "Decision Tables and Computer Programming," U.K.: Heyden & Son Ltd, 1981.

[25] C. C. YANG, "Relational Databases", Prentice-Hall, Inc., Englewood Cliffs, N.J., 1986.

[26] V. Dahl, "Logic Programming as a Representation of Knowledge," IEEE Computer, October, 1983, pp 106-111.

GRADUATE SCHOOL
UNIVERSITY OF ALABAMA IN BIRMINGHAM
DISSERTATION APPROVAL FORM

Name of Candidate Akram I. Salah

Major Subject Computer and Information Sciences

Title of Dissertation An Integration of Decision Tables and Relational Databases into a Prolog Environment

Dissertation Committee:

Kevin D. Reilly, Chairman — Fouad H. Fouad

Chao-Chih Yang

Warren D. Jones

Pradip Bey

Director of Graduate Program Warren D. Jones

Dean, UAB Graduate School Anthony Ramoa

Date 6/6/86

PS-1428