SPARC/E(V.2) AN ELEUSIS RULE GENERATOR AND GAME PLAYER

by

R. S. Michalski
H. Ko
K. Chen

Reports of the Intelligent Systems Group, ISG 85-11, UIUCDCS-F-85-941, Department of Computer Science, University of Illinois, March 1985.

File No. UIUCDCS-F-85-941

SPARC/E(V.2)
An Eleusis Rule Generator and Game Player

By

Ryszard Michalski
Heedong ko
Kaihu Chen

Internal Report

Intelligent Systems Group
Department of Computer Science
University of Illinois at Urbana-Champaign

This work was partially supported by the National Science Foundation under grant NSF DCR 84-06801.

March 1985

Table of Contents

1. Introduction

1.1. Background

The type of inductive learning investigated here is **part-to-whole generalization**. For example, given a set of fragments of a scene, the problem is to hypothesize the description of the whole scene. Such a description is used to predict the rest of the scene. Examples of such prediction problems include medical diagnosis, stock market forecast, and estimating the damage to corn from cutworm infestation.

Unlike the prediction problem in letter sequence extrapolation, this work involves objects in the sequence characterized by many relevant attributes. Further complexity is introduced by relationships among these attributes. For example, the pattern may involve the recurrence of certain attributes depending on the attributes of objects preceding it at some arbitrary distance in the past. The Eleusis card game, which models a process similar to scientific discovery, is identified as a domain with the above mentioned complexities and chosen as a domain of investigation.

This research represents an attempt in developing a program that provides problem-specific performance together with ease of modification for application to different problems. Generality is obtained by adhering to a knowledge discipline--the program is constructed as a layered learning system in which the top-most layers use problem-specific knowledge, and the bottom-most layers use only general induction knowledge (See Figure 1). The SPARC/E(V.2) program is such a program specialized in playing the Eleusis game autonomously. To apply the program to closely related problems, the top two Eleusis-oriented layers may be removed and replaced by new layers which perform functions peculiar to the new problem. To apply the program to vastly different problems (which do not involve sequentially ordered data), all but the bottom-most layer may need to be rewritten. Expert-level performance is achieved by permitting the upper layers to make extensive use of domain-specific knowledge in whatever form is convenient.

1.2. Structure of this Report

This report discusses three main topics. First, the problem of describing a sequence of events is investigated. The possible types of descriptions are defined and basic techniques for discovering these

5	User Interface	Most Problem Specific
4	Eleusis Knowledge	
3	Segmentation	
2	Sequential Analysis	
1	Basic Induction	Most General

Figure 1. Layered Structure of Eleusis Program.

descriptions are detailed. The second major topic is the methodology of knowledge layers. The detailed design of the SPARC/E(V.2) program is presented. Lastly, the Rule Generator and Game Player elements of SPARC/E(V.2) are discussed and examples are given to demonstrate its strengths and weaknesses.

2. The Theory of Inducing Descriptions of Sequential Event Sets

This chapter presents the theoretical background and the basic algorithms used to develop the SPARC/E(V.2) tool.

2.1. Events and Sequences of Events

This research seeks to construct a tool which can find plausible descriptions of a sequence of events. Imagine, for example, that some process is occurring in time--a process which we do not understand. We wish to understand the process by describing it in a way which permits us to predict the future course of the process from its past history. We want this to be a plausible description--conceptually simple and in accord with our knowledge of the problem at hand. In order to develop such a description, we could take regularly spaced "snapshots" of the process. We could measure, at each snapshot, the state of the process in terms of a set of variables which we believe are relevant or which may improve our understanding.

These measurements form a sequence of **events** which merely **represent** the original process. Since events are symbolic entities, they are amenable to manipulation by a computer. Formally,

> Definition 1: An **event** is a symbolic description of a set of measurements taken of some process, situation, or occurrence.
>
> Definition 2: A **sequential event set (sequential e-set)** is a set of events which are arranged in a totally ordered sequence. **Time-series events** are events whose ordering is based on the order in which they occur in time.

There may be many different representations of events. An event may be as simple as a single number, as elaborate as a graph or predicate logic description. The specific representation chosen for this research is a vector of symbols known as a canonical VL_1 complex. A canonical VL_1 complex is equivalent to an ordered n-tuple of symbols. Each symbol describes some measurement taken of the original process. (A definition of VL_1 appears below).

There can also be many different types of sequences of events. For example, time-series events need not be equally spaced in time. Sometimes **negative events** are available which indicate incorrect extensions of the sequence of events. In some cases, errors may be present in the data. Errors can be of three types: errors of ordering, of measurement, and of membership in the sequence. Ordering errors manifest themselves as out-of-sequence events. Measurement errors involve events which do not accurately represent the actual processes being described. Membership in the sequence is a form of classification error in which events have been included in or excluded from the sequence incorrectly. For the purposes of this research, the events comprising the sequence are considered to be equally spaced and error-free. The algorithms presented in this report work best when negative events are available, but satisfactory performance can be obtained without negative events.

It is beyond the scope of this report to handle sequential event sets which contain noise. Although many researchers have been criticized for ignoring noise, it was felt that there were plenty of difficult problems to solve in sequential data analysis without introducing noisy events as an additional feature. Error handling can be incorporated to some extent within the knowledge layer programming methodology. For example, errors of measurement can often be detected by using a knowledge-based preprocessing layer to filter them out. This approach is taken to some extent in Meta-DENDRAL [4,5,6] and in BASEBALL [34,35]. Noisy data admit many more plausible descriptions than error-free data. In order to develop plausible descriptions of noisy data, either more search or more problem knowledge is required. It is an open question as to how such problem knowledge can be kept carefully separated from the general-purpose knowledge of the induction program and yet still be used effectively to eliminate noisy descriptions.

2.2. The Description Language VL_1

The techniques and notation of VL_1 are used heavily in this report. VL_1 (Variable-valued Logic 1 [26,29,30]) is an extension of the propositional calculus (zeroeth-order logic) which uses the concept of a **selector** as the basic building block for propositions.

> Definition 3: A **selector** consists of a variable, a set of values called a **reference**, and a relation defined between the variable and the set of values.

Syntactically, a selector is written as

[variable relation reference]

An example of a selector is:

[suit = clubs, diamonds]

The variable is **suit.** The reference is {clubs, diamonds}, and the relation is =. This selector indicates that the suit variable may take on either of the values **clubs** or **diamonds.**

[size > 10]

This selector indicates that size must take a value greater than 10.

In any particular VL_1 system, each variable is defined to have an explicit set of values called its **domain.** All values which appear in the reference of a selector must be taken from the domain. For example, the domain of suit is {clubs, diamonds, hearts, spades}.

Each variable in a VL_1 system is also given a domain type which specifies the permitted generalizations of the variable. For example, the **interval** domain type indicates that any reference can be generalized by closing the interval between the smallest and the largest elements of the reference. Thus, the selector

[value = 2,5]

may be generalized to

[value = 2,3,4,5]

if **value** has an interval domain. Domain types have the very important function of providing problem-specific knowledge to the inductive program. In addition to interval domains, the SPARC/E(V.2) program supports:

(1) nominal domains. All elements are unrelated and no plausible generalizations exist.

(2) cyclic interval domains. The elements in a cyclic domain are circularly ordered so that end-around intervals are permitted. (Example: Card values are sometimes considered to be circular so that J Q

K A 2 is a straight).

Intervals, both cyclic and normal, are denoted in the reference by writing the endpoints of the interval separated by two dots. Thus, [value=2,3,4,5] is written as [value=2..5], and [value=J, Q, K, A, 2] is written as [value=J..2].

Both events and descriptions can be conveniently represented by conjunctions of selectors called **complexes.**

> Definition 4: A **complex** is a conjunction of selectors. It is written by placing selectors directly adjacent to each other:
>
> [suit = clubs, diamonds][value < 3]
>
> (This conjunction describes the cards {AC, 2C, AD, 2D}). A **canonical complex** is a complex in which all variables are present, and all selectors have the = relation and a single value in the reference.

In the context of sequential data analysis, we use a subscripting notation to indicate the ordering of various events. The subscript zero on a variable indicates that that variable refers to the current event of interest. A subscript of one refers to the event immediately preceding; a subscript of two, to the event before that; and so on. For example, [color1=red][value0>6] indicates that the color in the preceding event was red and the value in the current event is greater than 6. We also introduce so-called **difference** and **sum** variables. The variable dvalue01 has a value equal to value0-value1. And the variable svalue01 takes on value0+value1.

We noted above that a canonical VL_1 complex is equivalent to an n-tuple of symbols. To use such a representation, all of the variables in a VL_1 system must be placed in some order. Then the elements in the n-tuple provide the references for each variable in that order. Thus, if we order the card variables as value followed by suit, the pair (10, clubs) is equivalent to the canonical VL_1 complex [value=10][suit=clubs].

2.3. Descriptions and Predictions

How can a sequential event set be described? We seek descriptions which permit us to predict the future behavior of the sequence from past events.

> Definition 5: A **prediction** concerning an event E, is a description, D, of the set of possibilities for E along with some specification of the likelihood of each possibility. We write
>
> $$D >\!\!-\!\!> E$$
>
> when a description predicts an event.

In traditional fields, a statistical prediction specifies the possible values of some variable along with a probability distribution function which indicates the probability of each possible value. In the present work, a prediction is a logical description which subsumes all possibilities for the event in question. For example, a prediction that the next card will be red is merely the description [color0=red] along with the understanding that this is a perfect description (probability 1). There are two fundamental types of descriptions for sequential event sets which allow us to predict the future course of the sequence: **look-back** descriptions and **periodic** descriptions. A lookback description is a function, F, of the most recent events, which predicts the next event. If

$$S = \langle E_1, E_2, E_3, \ldots, E_n \rangle$$

is a sequence of events, then F can be applied to the **lb** most recent events prior to any E_i in order to predict E_i:

$$F(E_{i-lb}, E_{i-(lb-1)}, \ldots, E_{i-2}, E_{i-1}) >\!\!-\!\!> E_i$$

lb is called to lookback parameter. It indicates how far into the past it is necessary to look back in order to predict the next event. In a simple Markov process, for example, a lookback parameter of 1 is all that is ever required. An example of a lookback description is the function

$F(x) = x+1$

which describes the sequence

<1, 2, 3, 4, 5, 6, 7, 8, 9, 10>

by predicting the next value in the sequence as a function of the previous value: $F(E_i) >\!\!-\!\!> E_{i+1}$.

A **periodic description** is a periodic function which describes each event in the sequence as a function of the position of that event in the sequence. For example, the periodic description

$P(x) = x \bmod 4$

describes the sequence

<1, 2, 3, 0, 1, 2, 3, 0, 1, 2, 3, 0>

since

$P(i) >\!\!-\!\!> E_i$.

Since P is a periodic function, the function has a period or length, T, after which it repeats. The **phase** of an event is its relative position within the period. All events in the same phase have the same prediction. The sequence:

<2C, 4H, 7C, AH, 6C, JH>

may be described by the periodic function P:

P(i) = [suit0=club] if i mod 2 = 1

P(i) = [suit0=heart] if i mod 2 = 0.

All of the clubs are in the first phase of the period, and all of the hearts are in the second phase. A convenient way to specify the periodic function P is simply to list the descriptions of each phase as an ordered n-tuple. We could rewrite the above function as

P: ([suit0=club], [suit0=heart])

where it is understood that **[suit0=club]** describes the first phase, and **[suit0=heart]** the second.

2.4. Description Models

Induction is the process of finding plausible and useful descriptions of events. One approach to induction is to identify models which specify the form of plausible descriptions. Induction then becomes the two step process of first fitting data to a model and second evaluating the fit to assess the plausibility and utility of the resulting description. Such techniques have long been used in traditional regression analysis where the model is usually some specific regression polynomial. Statistical tests for goodness-of-fit have been developed for such models.

Definition 6: A **model** prescribes the specific functional or syntactic form for a description.

Examples of description models are the decision tree used by Hunt [18], and the disjunctive normal form used by Michalski [23,25,30]. In a numerical sequence, a model might specify that the description is to be a lookback description in which the prediction is a linear function of the value of the previous number in the sequence:

$$F(x) = ax + b.$$

In this model, the a and b parameters need to be determined from the data. Obviously, the models used by a program carry a good deal of implicit problem-specific knowledge. It is important that a general inductive tool permit modification and manipulation of the models chosen.

Three models have been identified for use in SPARC/E(V.2):

(1) Periodic conjunctive model. This model specifies that the description must be a periodic description in which each phase is described by a single VL_1 complex. Example:

Period ([color0=red], [color0=black])

describes an alternating sequence of red and black cards.

(2) Lookback decomposition model. This model specifies that the description must be a lookback description in the form of a disjunctive set of if-then rules:

[color1=red] => [value0<5] v [color1=black] => [value0>=5].

The left-hand sides, or condition parts, of the rules must only refer to events prior to the event to be predicted (subscripts 1, 2, etc.). The right-hand sides provide predictions for the next event in the sequence given that the condition part is true. The decomposition model requires that the left-hand sides be disjoint—that only one if-then rule be applicable at any time. Furthermore, the right-hand sides should also be disjoint.

(3) Disjunctive Normal Form (DNF). This lookback model requires only that the description be a disjunction of VL_1 complexes. An example is:

[dsuit01 = 0] V [dvalue01 = 0]

which indicates that either the suit of the current card must be the same as the suit of the previous card, or the value of the current card must be the same as the value of the previous card.

From a logic standpoint, any decomposition rule (and many periodic rules) can be written in disjunctive normal form. The periodic and decomposition models are useful not because of their theoretical expressiveness or power, but because they assist in locating plausible descriptions quickly. The space of all DNF descriptions is very large and difficult to search.

2.5. Descriptions Based on Segmentation

Very often sequences of events are best described in a hierarchical fashion as a sequence of subsequences. For example,

S = <3, 4, 4, 5, 5, 5, 6, 6, 6, 6, 7, 7, 7, 7, 7>

is best described as a sequence of subsequences. Each subsequence is a string of identical digits. The length of each subsequence is one longer than its predecessor. The digit used in the subsequence is one larger than the digit used in the previous subsequence. In VL_1, this can be indicated by the two part description:

string = [dvalue01=0] (A)

[dvalue01=+ 1][dlength01=+ 1] (B)

Statement (A) defines a subsequence to be a string of adjacent events satisfying the constraint that their values must remain constant (dvalue01=0). The sequence is **segmented** into strings of maximal length satisfying this **segmenting condition.** This yields, in this example, the derived sequence

S' = < (3,1), (4,2), (5,3), (6,4), (7,5)>

In S' we have used the n-tuple representation for VL_1 events. The first value in each event is the digit used in the corresponding string of events in S. The second value specifies the length of the corresponding string in S.

Once the sequence has been segmented, a DNF description, statement (B), can be written. In (B) dvalue01 and dlength01 refer to the values and lengths of the events in sequence S'.

Any of the description models listed in section 2.4 can be applied to a sequence after it has been segmented. The discovery of such segmented descriptions requires both the discovery of the segmentation condition and the discovery of the description of the segmented sequence.

2.6. Discovering Descriptions--VL_1 Induction Algorithms

How can these descriptions be discovered? In this section we outline the basic algorithms used to discover descriptions in SPARC/E(V.2). The general approach is to choose a segmentation condition, a value for the lookback parameter, and a model. Then one of the VL_1 induction algorithms described in this section is called to fit the data to a model and assess the quality of the fit. The VL_1 algorithms are provided with events which have been developed by **transforming** the original sequence. As an example, consider the sequence of cards, S, shown in Figure 2. Assume, for the moment, that no segmentation condition is applicable and that we are considering a lookback parameter of 1. This sequence of events can then be transformed into the VL_1 events listed in Table 1.

Notice, using Table 1, that [dcolor01=1] for all events (i.e. color always changes from one card to the next). The VL_1 induction algorithms seek to discover exactly this sort of description.

The variables listed in Table 1 are called **derived variables** because they are derived from the original sequence. The events are **derived events.** The events in Table 1 are unordered. The original ordering of the sequence has been made explicit through the difference variables (dvalue, dsuit, and dcolor).

S = <2C, 10D, 3S, AD, JC, 6H, 6C>.

Figure 2. Example Sequence of Cards.

value1	suit1	color1	value0	suit0	color0	dvalue01	dsuit01	dcolor01
2	C	black	10	D	red	8	+1	1
10	D	red	3	S	black	-7	+2	1
3	S	black	A	D	red	-2	+2	1
A	D	red	J	C	black	10	+3	1
J	C	black	6	H	red	-5	+2	1
6	H	red	6	C	black	0	+2	1

Table 1. Transformed VL_1 Events

Color is derived using knowledge of the characteristics of the cards.

In generating Table 1, value was given an interval domain, suit a cyclic interval domain (with the suits ordered as clubs, diamonds, hearts, spades, clubs, ...), and color a nominal domain (red or black). The difference variables reflect these domain types. Dvalue01 takes on values from -12 to $+12$, but dsuit01 takes values 0, 1, 2, and 3. Differences for cyclic interval domains are computed as values modulo n, where n is the size of the domain. Thus, the difference between clubs and hearts is + 2 ((0-2) modulo $4 = 2$). Dcolor01 is an example of a difference on a nominal variable. Dcolor is 0 if color0=color1, and 1 otherwise.

Table 1 could be used to discover DNF and decomposition descriptions with a lookback of 1, but it would not be useful for discovering periodic descriptions or descriptions with other lookbacks. Different derived variables and different events are required for discovering descriptions which fit different description models.

2.6.1. The Aq Algorithm

Much work in induction has been conducted by Michalski and his collaborators. Most of this work is based on the Aq algorithm [23,25,30] which was originally developed in the context of switching theory. This algorithm accepts as input a set of positive events and a set of negative events. Each event is a canonical VL_1 complex. Aq considers each VL_1 variable to be a variable in a multiple-valued logic covering problem. By developing a **cover** of the positive events against the negative events, Aq produces a description which is satisfied by all of the positive events and by none of the negative events. (A description **covers** an event if the event satisfies the description). The process of developing a cover involves partially computing the complement of the set of negative events and intelligently selecting complexes which cover positive events. The final cover may be a single complex or a disjunction of complexes. Aq seeks to develop a disjunction with the fewest number of complexes possible, but the algorithm is only quasi-optimal. It is capable, under certain conditions, of giving an upper bound on the distance from optimality of the solution it produces.

The algorithm proceeds in best-first fashion by the method of disjoint stars. A positive event, e1, is chosen and a star is built about e1. A **star** is an approximation to the set of all prime implicants which cover e1 (and are in the complement of the set of all negative events). The best complex in the star, lq, is chosen to form part of the solution. All events covered by lq are removed from further consideration. The process of choosing a positive event, e1, building a star about e1, and selecting the best element of the star is repeated. However, the new e1 must not have been covered by any element of **any previous star.** In this manner, disjoint, well-separated stars are built, and lq's are selected. The process repeats until all events have been covered by at least one star. Some clean-up operations are required in the case where some positive events were covered by some star, but by no lq.

The process of building a star about an event e1 is simple. The disjunction of all negative events is complemented and then multiplied out, one event at a time. After each event is multiplied out, the set of intermediate products (so-called partial stars) is trimmed according to a user-specified preference criterion, and only the **MAXSTAR** best elements are retained. The final star has at most **MAXSTAR** elements in it.

Note that all of the steps mentioned (complementation, multiplication, etc.) are being performed on variables which can take on a **set** of values. This is a multiple-valued covering process.

The strengths of the algorithm include

(1) flexibility - The user can specify the preference criterion which determines which lq is chosen from each star and which partial stars are retained during the star-building process. The algorithm can be modified to develop strictly disjoint complexes in the solution.

(2) optimality - If no trimming is performed, the algorithm provides a measure of how many additional complexes appear in the solution above and beyond the minimum necessary for the optimal solution.

(3) quasi-optimality - The algorithm performs well even when the stars must be severely trimmed.

Unfortunately, the Aq algorithm is not sufficient for discovering all of the description models required for describing sequential event sets. Aq was designed to develop descriptions in disjunctive normal form with the fewest number of complexes. This is only one of the description models discussed above. Neither periodic nor decomposition descriptions can be easily discovered using Aq (although equivalent descriptions are sometimes discovered as DNF descriptions). Aq tends to develop a "lopsided" cover. Since it proceeds in a best-first manner, it selects the largest prime implicant first and smaller prime implicants later. Descriptions involving symmetry (decomposition descriptions in particular) tend to be overlooked. Thus, although the Aq algorithm is powerful and useful, it does have some limitations which prevent it from completely solving the discovery problems of sequential event sets.

Two other algorithms have been incorporated into the SPARC/E(V.2) program: the decomposition algorithm and the periodic algorithm.

2.6.2. The Decomposition Algorithm

This algorithm seeks to fit the data to a decomposition model. It accepts as input a set of positive events and a set of negative events. In addition, some variables are designated as "left-hand side" variables (i.e. variables which describe previous events). A decomposition model seeks to explain the events in terms of the values of "left-hand side" variables. A decomposition description for the events in Table 1 would be [color1=red] => [color0=black] v

[color1=black]=> [color0=red].

This description **decomposes on** color1. It breaks the description of the sequence into two if-then rules. The => can be interpreted as an implication. The decomposition algorithm takes advantage of the constraints that both the left-hand and right-hand parts of the if-then rules must be single VL_1 complexes and that the left-hand sides must be disjoint.

The decomposition algorithm starts by performing a trial decomposition on each possible left-hand side variable. A trial decomposition for a given variable is formed by creating a complex for each possible value of the given variable (This basic idea was suggested to me by R. S. Michalski). All events covered by the given value of the given variable are merged together to form a complex. (The references of corresponding selectors are unioned). For example, using the events of Table 1, trial decompositions could be performed on value1, suit1, and color1 to yield the complexes shown in Table 2. The general idea is to form trial decompositions, choose the best decomposition, and break the problem into subproblems, one for each if-then rule in the selected decomposition. The algorithm can then be applied recursively until a consistent description has been developed.

Table 2 shows the raw trial decompositions. These are very specific and very poor descriptions. They must be processed further before a decision can be made as to which decomposition is best and should be further investigated. Three processing steps are applied to the trial decompositions.

The first processing step involves interval (and cyclic interval) variables such as value1. These variables often have many values and trial decompositions based on them are very uninteresting and implausible. (An SPARC/E(V.2) rule with 13 separate cases would be impossible to discover!). An attempt is made to close intervals on the left-hand side of the trial decomposition. Imagine, for example, that some sequence is well-described by the decomposition:

[value1<8] => [color0=red] v [value1>=8]=> [color0=black]

A trial decomposition would involve up to 13 different complexes for value1. The first processing step attempts to detect that all if-then rules below [value1=8] should be combined into one if-then rule, and

On value1:

[value1=a] => [value0=j][suit0=C][color0=B][dvalue01=10][dsuit01=+ 3][dcolor01=1]
[value1=2] => [value0=10][suit0=D][color0=R][dvalue01=8][dsuit01=1][dcolor01=1]
[value1=3] => [value0=A][suit0=D][color0=R][dvalue01=-2][dsuit01=+ 2][dcolor01=1]
[value1=6] => [value0=6][suit0=C][color0=B][dvalue01=0][dsuit01=+ 2][dcolor01=1]
[value1=10]=> [value0=3][suit0=S][color0=B][dvalue01=-7][dsuit01=+ 2][dcolor01=1]
[value1=j] => [value0=6][suit0=H][color0=R][dvalue01=-5][dsuit01=+ 2][dcolor01=1]

On suit1:

[suit1=C] => [value0=6,10][suit0=D,H][color0=R][dvalue01=-5,8][dsuit01=+ 1,2][dcolor01=1]
[suit1=D] => [value0=3,J][suit0=C,S][color0=B][dvalue01=-7,+ 10][dsuit01=+ 2,3][dcolor01=1]
[suit1=H] => [value0=6][suit0=C][color0=B][dvalue01=0][dsuit01=+ 2][dcolor01=1]
[suit1=S] => [value0=a][suit0=D][color0=R][dvalue01=-2][dsuit01=+ 2][dcolor01=1]

On color1

[color1=R] => [value0=3,6,J][suit0=C,S][color0=B][dvalue01=-7,0,10][dsuit01=+ 2,3][dcolor01=1]
[color1=B] => [value0=a,6,10][suit0=H,D][color0=R][dvalue01=-2,-5,8][dsuit01=1,2][dcolor01=1]

Table 2. Trial Decompositions.

that all if-then rules above [value1=7] should be combined into another if-then rule. The algorithm operates by computing distances between adjacent if-then rules and looking for sudden jumps in the distance measure. Where a jump occurs (a local maximum), the algorithm tries to split the domain into cases.

The distance computation is a weighted multiple-valued Hamming distance. The weights are determined by taking user-specified plausibilities for each variable and relaxing these weights according to the discriminating power of each variable (taken singly). For instance, if a right-hand side variable is irrelevant in some if-then rule (i.e. its reference contains all possible values so that it is a **don't care** selector), then its weight is reduced to zero. The distances between adjacent if-then rules are computed and local maxima are located. If there is one maximum, the interval is split there, and two if-then rules are created. If there are two maxima, three if-then rules are created. If there are more than two maxima, the smaller maxima are suppressed.

Similar techniques are used for cyclic interval domains.

Once the cases have been determined, each trial decomposition is next processed by applying the domain-specific rules of generalization to the selectors on the right-hand sides of the if-then rules. Intervals are closed for interval variables and cyclic interval variables. Special domain types are defined for difference variables (variables derived by subtracting two other variables). The rules of generalization for difference variables attempt to find intervals about the zero point of the domain. Thus, [dvalue01=-3,1,2] would be generalized to [dvalue01=-3..+3]. One-sided intervals away from zero are also created: [dvalue01=3,4,6] would be generalized to [dvalue01>0]. These generalizations are only performed if the reference contains more than one value. Corresponding to the trial decompositions of Table 2 we get the generalized trial decompositions of Table 3. The notation [variable = *] is used when a variable can take on any value from its domain (i.e. it is irrelevant).

The third processing step examines the different if-then rules and attempts to make the right-hand sides of the rules disjoint by removing selectors which have overlapping references. Table 4 shows the results of this step.

On value1

```
[value1=a..4] => [value0=a..j][suit0=C,D][color0=*][dvalue01<>0][dsuit01<>0][dcolor01=1]
[value1=5..k] => [value0=3..6][suit0=C,S][color0=*][dvalue01<=0][dsuit01=2][dcolor01=1]
```

On suit1

```
[suit1=C]  => [value0=6..10][suit0=D,H][color0=R][dvalue01<>0][dsuit01=1,2][dcolor01=1]
[suit1=D]  => [value0=3..J][suit0=C,S][color0=B][dvalue01<>0][dsuit01=2,3][dcolor01=1]
[suit1=H]  => [value0=6][suit0=C][color0=B][dvalue01=0][dsuit01=2][dcolor01=1]
[suit1=S]  => [value0=a][suit0=D][color0=R][dvalue01=-2][dsuit01=2][dcolor01=1]
```

On color1

```
[color1=R] => [value0=3..J][suit0=C,S][color0=B][dvalue01=*][dsuit01=2,3][dcolor01=1]
[color1=B] => [value0=a..10][suit0=H,D][color0=R][dvalue01<>0][dsuit01=1,2][dcolor01=1]
```

Table 3. Generalized Trial Decompositions.

On value1

[value1=a..4] => TRUE
[value1=5..k] => TRUE

On suit1

[suit1=C] => TRUE
[suit1=D] => TRUE
[suit1=H] => TRUE
[suit1=S] => TRUE

On color1

[color1=R] => [suit0=C,S][color0=B]
[color1=B] => [suit0=D,H][color0=R]

Table 4. Trial Decompositions With Overlapping Selectors Removed.
(Irrelevant selectors are omitted)

At this point, the algorithm has identified the rule fairly well. Now the best decomposition can be selected. The selection process uses a set of cost functions which measure characteristics of each trial decomposition. The cost functions are:

(1) Count the number of negative examples that are incorrectly covered by this decomposition.

(2) Count the number of cases (if-then rules) in this decomposition.

(3) Return the user-specified plausibility for the variable being decomposed on.

(4) Count the number of null cases for this decomposition (e.g. [value1=4] is a null case in Table 2).

(5) Count the number of "simple" selectors in this decomposition. A simple selector can be written with a single value or interval in the reference (e.g. [value01>4] is a simple selector). After applying the generalization rules (as in Table 3) all selectors except those with nominal variables are necessarily simple.

The cost functions are applied in an ordered fashion using the functional sort algorithm developed by Michalski [26]. The trial decomposition with the lowest cost is selected. The lowest cost decomposition in Table 4 is the decomposition on color1. The other decompositions are completely overgeneralized.

The algorithm does not always proceed as indicated above. The user can request that the best trial decomposition be selected after performing only the first post-processing step, or after the second post-processing step has been completed. In fact, it is recommended that the best decomposition be selected after the second step.

Once the best trial decomposition has been selected, it is checked to see if it is consistent with the events (covers no negative events). If it is, the composition algorithm terminates. If it is not, the problem is decomposed into separate subproblems, one for each if-then rule in the selected decomposition. Then the algorithm is repeated to solve these subproblems. (The subproblems are solved simultaneously, not independently).

The strengths of the decomposition algorithm are

(1) speed - The algorithm locates good decompositions quickly.

(2) aptness - The algorithm locates descriptions which fit the decomposition model very well.

The weaknesses of the algorithm are

(1) inability to produce alternatives - This is a best-first algorithm which only returns one description. Often it is desirable to have a learning algorithm which returns a set of possible descriptions.

(2) restricted model - The algorithm was designed for a specific model. The generality of this model has not yet been demonstrated.

2.6.3. The Periodic Algorithm

The periodic algorithm is really just a modified version of the decomposition algorithm designed for discovering descriptions which fit the periodic model. A parameter is provided to the algorithm which indicates the number of phases to expect in the description. Each phase is treated in somewhat the same way as the different if-then cases are treated in a trial decomposition. First, the events in each phase are combined to form a single complex (by forming the union of references of corresponding selectors). For the sequence S in Figure 2, the results are shown below. Note that no difference variables or variables describing previous events are included in these derived events.

phase 1: [value0=10,a,6][suit0=D,H][color0=R]

phase 2: [value0=3,j,6][suit0=C,S][color0=B]

If these complexes are consistent with the negative examples, then the references are generalized according to the domain types of the variables:

phase 1: [value0=a..10][suit0=D,H][color0=R]

phase 2: [value0=3..j][suit0=C,S][color0=B]

If these generalized complexes are still consistent, selectors with overlapping references (overlapping with selectors in other phases) are removed:

phase 1: [suit0=D,H][color0=R]

phase 2: [suit0=C,S][color0=B]

If these complexes are still consistent, they are returned as the final description.

Both the periodic and the decomposition algorithms go through these post-processing steps until the description becomes inconsistent. When this occurs, the algorithm backs up and returns the version of the description before it was overgeneralized to become inconsistent. In some cases, the star generation process of the Aq algorithm is invoked to attempt to extend the description against negative examples.

2.7. Relationship to Statistical Methods

There are many direct parallels between the previous discussion of sequential data analysis and the traditional area of time-series analysis.

Time-series events occur in many systems: the economy, the factory, the environment. Techniques have been developed to predict the future course of the time-series and to determine the appropriate amount of feedback required to control the system. The same sorts of descriptive models discussed above exist in traditional areas — the event forms and the inductive techniques differ drastically.

There are two primary approaches to time-series analysis: **regression** methods and **spectral** methods. Regression methods attempt to explain the behavior of a particular variable (the dependent variable, y) in terms of the previous behavior of a set of variables (the independent variables, x_i). If the past behavior of the dependent variable is a function of itself, the system is called **autoregressive.** Regression-based descriptions are the statistical counterparts of the lookback models described above. To fit data to a regression model, the user must specify a particular model, the regression polynominal. Often the form of the regression polynomial is suggested by theory within the field of application. The technique of least-squares regression is applied to estimate the constant parameters of the regression polynomial. If certain assumptions hold, a measure of goodness-of-fit (total explained variance) can be obtained.

Spectral methods attempt to describe the behavior of a particular variable by analyzing its frequency spectrum. This is the continuous frequency counterpart of the discrete periodic models described before. Fourier analysis is used to determine the frequency components that make up the "waveform" of the dependent variable. The independent variable is time.

Here are some examples:

Economic time-series. Let us examine the series

$$S = \langle D_1, D_2, D_3, \ldots, D_k \rangle$$

where each D_i is an ordered pair, $D_i = (Y_i, X_i)$.

Let

Y_i = demand for beef at time i,

X_i = supply of beef at time i.

Economic theory predicts that the demand for beef is a function of the recent values for supply. The form of the regression polynomial is

$$Y_i = B_0 + B_1 X_{i-1} + B_2 X_{i-2}.$$

Using the data in S, the coefficients B_0, B_1, and B_2 can be estimated. The goodness-of-fit of the model can be tested.

Plant management. Imagine a plastics factory where some of the key ingredients are water, oil, and heat. Let

$$S = \langle E_1, E_2, \ldots, E_n \rangle.$$

Where each $E_i = (y_i, u_i, v_i, w_i)$:

y_i = output per minute of plastic at time i

u_i = input of water (per minute) at time i

v_i = input of petroleum base at time i

w_i = temperature of the reaction chamber at time i.

In order to predict the future production of the plant, we want to describe y_i in terms of previous values of u, v, and w. Water is believed to have a parabolic effect on plastic output. The regression polynomial looks like this:

$$y_i = B_0 + B_1 y_i + B_2 u_i + B_3 v_i + B_4 w_i$$

Using linear regression, we can estimate the coefficients B_0 through B_4 from the data. The regression polynomial need only be linear in the coefficients.

An autoregressive sequence might have the form:

$$y_i = B_0 + B_1 y_{i-1} + B_2 y_{i-2}.$$

Box and Jenkins [3] describe techniques for estimating the degree of autocorrelation (the lookback parameter) from the data. Such techniques permit the researcher to base the form of the model as well as the specific content of the model on the data. Few such heuristics exist in logical sequential data analysis.

3. The Methodology: Knowledge Layers

In this chapter we describe the programming methodology used to develop the SPARC/E(V.2) program. The steps of the methodology are illustrated by indicating how they were applied to SPARC/E(V.2). The knowledge layer methodology has been very useful in designing the SPARC/E(V.2) program.

3.1. Description of the Methodology

The goal of any programming methodology is to enhance the quality and performance of the program and improve the productivity of the programmer. The knowledge layer methodology seeks to

- simplify the programming process by providing a framework (knowledge layers) for problem decomposition,
- develop general learning programs which are easily adapted to solve related learning problems,
- develop learning programs with sufficient power to solve the problems at hand.

A program designed using the knowledge layer concept is built of distinct layers roughly like an onion (Figure 3).

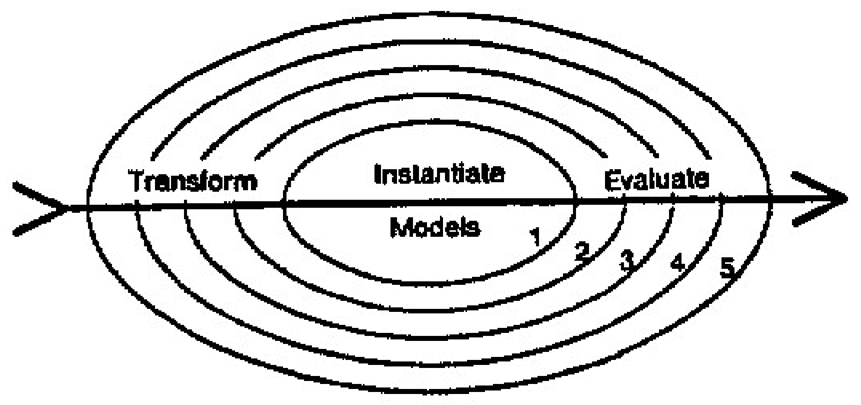


Figure 3. The Knowledge Layer Scheme.

Each layer has access to a specific body of knowledge. Each layer may invoke the next layer within it and may examine the information returned by that layer. The outermost layer interacts with the user of the system to solve a specific class of problems. The innermost layer is the most general. It uses only very general knowledge and algorithms to accomplish its task. The layering is reflected in the generality of the knowledge used at each level, in the scope of variables at each level, and in the flow of control from one level to the next. The knowledge used at each level must all be of the same degree of generality, appropriate to the function of that layer. The variables in that layer can be accessed by outer layers, but not by inner layers. Subroutine calls may only be directed at routines in the current layer or within inner layers. If this discipline is adhered to, the outer layers can easily be removed and replaced by layers better-suited to a particular task.

In order to apply the methodology, it is easiest to proceed by the following steps:

Step 1.

Identify the input representations. What kinds of data must the program accept? Do these data contain errors? Are negative examples available? How should the data be described?

Step 2.

Identify output representations. What kinds of output descriptions must the program produce? How can these be represented? What description models should be used?

Step 3.

Identify the basic algorithms needed to accomplish the learning task. Most learning in non-trivial environments requires three basic operations: interpretation, generalization, and evaluation. Furthermore, after generalized descriptions have been developed, they must be evaluated to assess their plausibility within the domain in question. This step (step 3) involves determining how the generalization process will take place. A few learning algorithms may be chosen from the many general-purpose algorithms currently in use. Alternatively, new algorithms may be required. These should be designed to use only general knowledge.

Step 4.

Identify the transformations required to prepare the input events for the general-purpose algorithms identified in Step 3. This step solves the interpretation portion of the learning problem.

Step 5.

Identify the evaluations and transformations necessary to convert the descriptions produced by the general induction algorithms into the desired output descriptions identified in Step 2. This step solves the evaluation portion of the learning problem.

Step 6.

Identify the knowledge needed to perform the tasks defined in steps 3, 4, and 5. What knowledge is needed to generalize the events? What knowledge is required to perform the transformations on the input data? What knowledge is required during evaluation? This is a very difficult step to perform because knowledge has a way of entering programs quietly and implicitly. It may help to imagine applying the program to different but related problems.

Step 7.

Decompose the program into layers according to the knowledge and tasks performed in each layer. In this step, corresponding functions of interpretation and evaluation are identified and grouped together in layers according to the knowledge required for each function. The layers are designed to surround the basic generalization functions and span the distance from these general-purpose algorithms to the special-purpose problem the program is intended to solve.

3.2. Applying the Methodology to SPARC/E(V.2)

3.2.1. Description of SPARC/E(V.2)

3.2.1.1. Description of the Game

Eleusis was invented over a period of years by Robert Abbott. It is an inductive game in which players attempt to discover a secret rule known only to the dealer. The secret rule describes a sequence of cards which are "legal." Players attempt, in their turns, to extend the sequence by playing one or more cards. The sequence of cards which has thus far been played is arranged in a layout (see Figure 4).

The layout has a main line which contains all of the correctly played cards in sequence. Incorrect cards are placed in side lines below the main line card which they follow. In a turn, a player may play a string of from one to four cards. If the cards are correct, the dealer places them in the proper positions on the main line.

```
mainline: 3h  9s  4c  jd  2c 10d  8h  7h  2c  5h
sidelines:    jd      ah  as          10h
              5d      8h 10s         (10s
                      qd               9s  <- string played incorrectly
                                       4s
                                       2s)
```

Figure 4. Sample Eleusis Layout

If any one of the cards is incorrect, the entire string is placed on a side line below the last legal card. The string of cards is overlapped so that players examining the layout can recall that only one of the cards in the string need be wrong.

The goal of the game is to get rid of all of one's cards. When a player plays correctly, he or she gets rid of the cards so played. If a player makes errors, the dealer deals additional cards equal in number to double the number of cards played by the player.

The secret rule is invented by the dealer at the start of each round. What prevents the dealer from choosing an impossibly difficult rule? Besides the dealer's natural desire to have an interesting game, the scoring for each round is contrived so that the dealer gets a score equal to the difference between the best and the worst scores for that round. Thus, the dealer is encouraged to choose rules of intermediate difficulty. The rules should stump some players but not others. In this way a large point spread can be created. There are additional rules for the game and the reader is advised to refer to appendix 4. This program should

- suggest possible rules to describe the layout,
- evaluate rules suggested by the player, and

- suggest possible cards to play from the player's hand.

Previous work on Eleusis has been done by Barto and Prager. Their work is limited to basic induction tasks. The rule model which they demonstrated in [2] was a decomposition model with a lookback parameter of 1.

3.2.1.2. Typical Rules and Rule Models

Here are some examples of secret rules (after Abbott[21]):

- R1 "If the last card was a spade, play a heart; if last card was a heart, play diamonds; if last was diamond, play clubs; and if last was club, play spades."
- R2 "The card played must be one point higher than or one point lower than the last card."
- R3 "If the last card was black, play a card higher than or equal to that card; if the last card was red, play lower or equal."
- R4 "Play alternating even and odd cards."
- R5 "Play strings of cards where each string is one card longer than the previous string and where a string is an ascending sequence of cards starting with an Ace."
- R6 "The sum of the values of the last card and the current card must be less than 16."

Where values are mentioned, Ace is usually understood to be 1, Jack 11, Queen 12, and King 13.

The rule models for these rules are precisely the description models introduced in Chapter 2. Rules R1 and R3 are decomposition rules with a lookback parameter of 1. Rule R2 is a DNF lookback rule (a degenerate form of a disjunction) since it expresses the value of one card in terms of the values of the previous card. R5 is also a DNF rule based on segmenting the sequence into strings of ascending cards. R4 is a periodic rule. R6 is a DNF rule, but instead of cards being related by differences, they are related by a sum.

3.2.1.3. Representing Eleusis Rules

Although the VL_1 descriptions introduced in Chapter 2 are sufficient for representing the Eleusis rules described above, it was felt that the user of the SPARC/E(V.2) program would prefer a more elegant and clear representation language. Therefore, VL_{22} was developed as a description language for rules which describe sequences of events. VL_{22} is a successor of VL_{21}[21]. Both languages are subsets of a very extensive description language, VL2 [21,27].

VL_{22} is an extension of first-order predicate logic which uses a VL_{22} selector as the basic building block for well-formed formulas. VL_{22} selectors are a bit more complex than VL_1 selectors:

[function (variable-list) relation function (variable-list) operation value-list]

Variables in the variable-lists refer to specific cards or strings. The same subscripting convention used in VL_1 is used in VL_{22} to indicate the order of the cards. For example, card0 refers to the current card; card1, to the card before card0; etc. Functions applied to these variables take on values from explicitly defined domains (exactly like VL_1 variables). Difference and sum variables are not needed in VL_{22} since functions can (optionally) appear in the reference. The operations required to express Eleusis rules are plus, minus, and plus-or-minus. Each VL_{22} expression is assumed to be universally quantified over the entire event sequence (with the implicit condition that card0 is adjacent to card1, card1 to card2, etc.). Table 5 shows the VL_{22} equivalents of the Eleusis rules listed above. Note that the dummy variable **string** is used to describe a string of cards in a segmented rule. Subscripts are applied the strings as well as to cards.

3.2.1.4. Plausible Rules

In order to discover Eleusis secret rules, we must first define what we are looking for. Induction is the process of selecting plausible descriptions from the space of all possible descriptions. In SPARC/E(V.2), we are searching for plausible rules to describe the layout. Abbott gives some guidelines for forming good Eleusis rules, and these quidelines can be used to define characteristics of plausible rules.

First of all, conceptual simplicity is important. Complex Eleusis rules will not score well for the dealer because no one will be able to guess them. Even apparently trivial rules are quite difficult for people to guess. Secondly, some rules permit many cards to be legal at many points. Abbott observes that rules which, on the average, permit fewer than one-fourth of the deck to be played are usually easier to

R1 [suit(card0)= suit(card1)+ 1]

R2 [value(card0)= value(card1)+ -1]

R3 [suit(card1)=black] → [value(card0)>=value(card1)] V
[suit(card1)=red] → [value(card0)<=value(card1)]

R4 Period ([parity(card0)=even], [parity(card0)=odd])

R5 string = [value(card0)=value(card1)+ 1] :
[length(string0)=length(string1)+ 1]

R6 [value(card0) <= - value(card1)+ 16]

Table 5. VL_{22} Descriptions of Eleusis Rules.

discover than rules which typically allow half the cards to be played. A rule which permits any card to be played any time is quite difficult to discover because no negative examples are ever produced. Thirdly, most dealers arrange the rule so that every card is playable at some time during the game.

These plausibility constraints can be used to evaluate rules produced by the general induction algorithms. To measure conceptual complexity, we can count the number of selectors in the rule. Other syntactic measurements, such as measuring the number of values in a reference, can be used to approximate conceptual complexity. The size of the set of legal cards can be deduced from the VL_1 description of the rule. An estimate of the average size of the set of legal cards can be developed and used to test the plausibility of the rule.

3.2.2. Design Steps

Here are the steps of the design methodology applied to the design of the SPARC/E(V.2) program:

- Identify Input Representations. The input representations for the SPARC/E(V.2) program are symbols of the form '2c' or 'jd' representing cards in a card deck. The input is entered in order of play. Each string of cards (one to four cards in length) is entered using a "card" command along with the judgment of the dealer:

card 2c 3d:y;

This command indicates that a player played two cards and the dealer pronounced them correct. The input is stored in the program as a linked list in the form of the layout.

- Identify Output Representations. Rules produced by the program are written in VL_{22} as described above. The rules fit the three description models described in Chapter 2: periodic, decomposition, and DNF.
- Identify Generalization Algorithms. The three algorithms presented in Chapter 2 are the algorithms used in the inner-most layer of the SPARC/E(V.2) system. Each algorithm is designed to fit unordered VL_1 events to one of the description models. Each algorithm produces a description in the form of a disjunction of VL_1 complexes.
- Identify Interpretation Steps. Four interpretation steps can be identified. The first step is to convert cards to canonical VL_1 complexes containing the suit and value of each card. Thus, 2c becomes [suit=clubs][value=2].

The second step is to derive additional variables which may lead to plausible descriptions of the layout. Color and parity (the value of the card modulo 2) might be added at this point. Also, some indication of whether the card is a faced card or has a value which is a prime number might be desired. In the second step we could transform [suit=clubs][value=2] to [suit=clubs][value=2][color=black][parity=even][faced=false][prime=true].

The third step involves segmenting the layout. As discussed in chapter 2, many interesting and plausible descriptions are based on segmentation. In SPARC/E(V.2), we segment the layout into strings of maximal length which satisfy a segmentation condition. Although it might be possible to develop some techniques for inferring the segmentation condition from the data, we have chosen to use a hypothesize and test approach. The program is provided with a list of segmentation conditions. It attempts to segment the layout with each condition and then evaluate how plausible the segmentation is. For example, if the segmented layout has nearly the same number of events as the original layout, then it is very unlikely that the layout is well-described using that segmentation condition. Conversely, if the whole layout satisfies the segmentation condition so that only one segmented event is produced then this is not a plausible segmentation either (invariant properties of the entire layout are discovered by other procedures).

The final transformation step involves making the order of the events explicit in the events and removing the order from the sequence. Once a model and a value for the lookback parameter have been chosen, it is easy to develop events like those of Table 1 which contain descriptions of the current card, the preceding cards, and relationships between them. Thus, this step computes sum and difference variables.

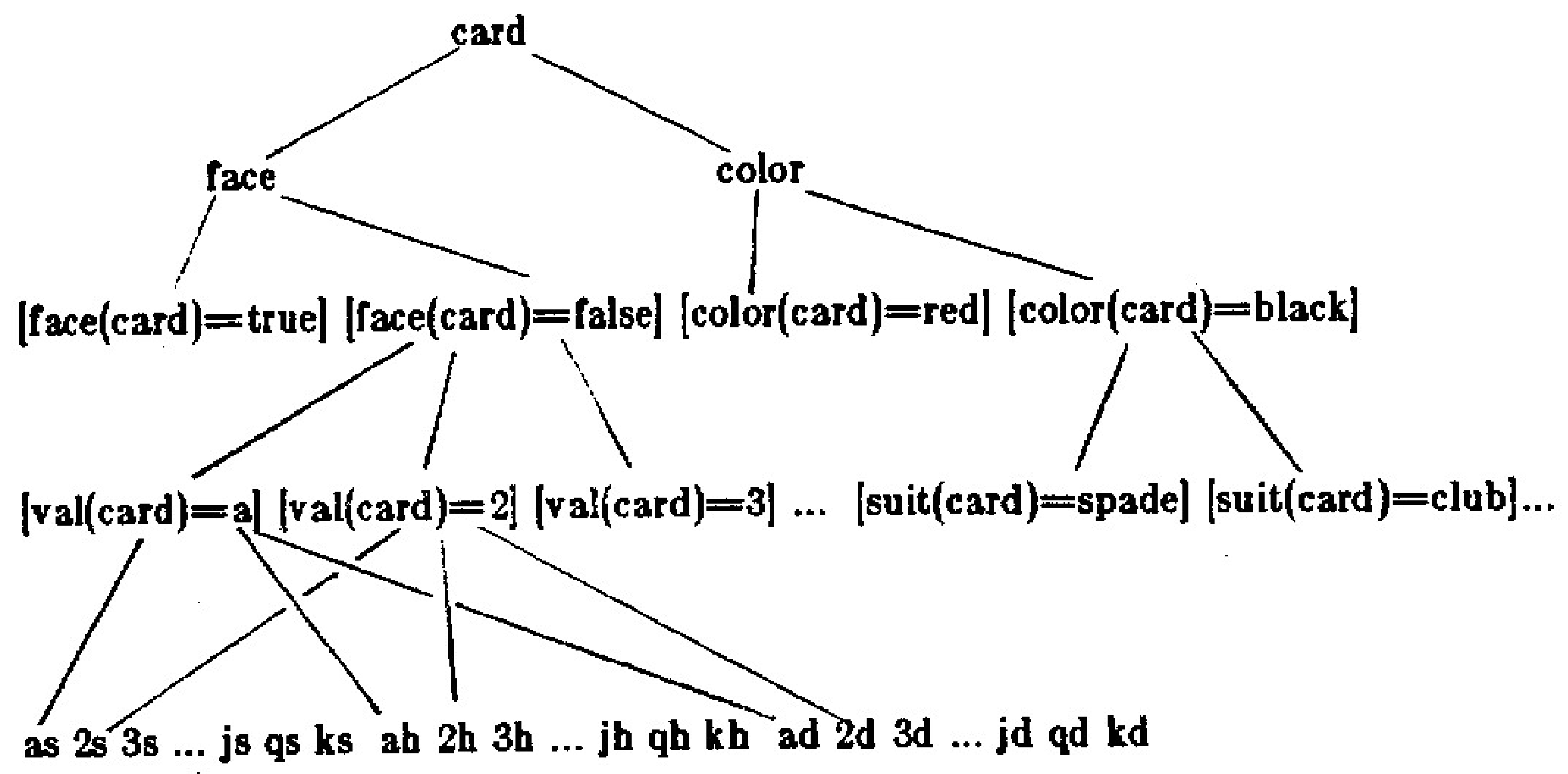
Once the events have been processed by these transformation steps, they are ready to be generalized using the VL_1 induction algorithms of the inner-most layer.

- Identify Evaluation Steps. Three evaluation steps can be identified. The **first step** examines rules developed by the VL_1 induction algorithms and filters them to remove redundant information. For example, it often happens that the VL_1 induction algorithms develop descriptions like:

  [face1=false] ⇒ [value0>=j] [value0>value1] [face0=true] &

  [face1=true] ⇒ [value0<=10] [value0<value1] [face0=false]

  The selectors [value0>value1] and [value0<value1] are redundant because [face1=false] implies [face0=true] is the same as [value0>value1], but logically correct, statements. The VL_1 induction algorithms cannot remove these selectors since the algorithms are not aware of the order of the events. There is another redundancy. [face0=true] and [value0>=j] are redundant. This redundancy was caused because VL_1 induction algorithm are not aware of the structural relationships between selectors. The filtering is done in layer 2 right after level 1 learning element discovers all the rules.



The structural information in Eleusis domain is shown above and is used in the filtering procedure as follows:

```
for each rule in the rulebase do
  for each complex in the rule do
      take two selectors based on the same primitive from the complex and call them s1 and s2;
      if s1 = s2 then keep more abstract selector
      else if s1 contains s2 then drop s1
          else if s2 contains s1 then drop s2;
```

Selector, s1, is more abstract than s2 if s1 is defined in terms of s2. It makes sense to consider those descriptors for redundancy only if they are compatible. For instance, it does not make sense to compare face descriptor with color descriptor or to compare face with dvalue descriptor. The first problem is easily solved by testing subset relationship. The second problem is solved by only comparing descriptors base on same primitives. For instance, both face and value descriptors are comparable because they are based on the same primitives, cards, but not face and dvalue.

The **second evaluation** step is required when the layout has been segmented. Using a segmentation condition, the end of the layout cannot be successfully segmented. For example, if we had the sequence:

$$S = <3, 4, 4, 5, 5, 5, 6, 6, 6, 6, 7, 7>$$

we would not want to create an event for the sevens. Such an event would indicate that there was a string of sevens of length 2. If the VL_1 induction algorithms received such an event, they would not be able to discover that the length of a string always increases by 1. Thus, the segmentation process must always leave the end of the layout unsegmented. However, when a description is developed, it might in fact be inconsistent with the unsegmented portion of the layout. If the sequence had looked like

$$S = <3, 4, 4, 5, 5, 5, 6, 6, 6, 6, 7, 7, 7, 7, 7, 7, 7>$$

then the VL_1 induction algorithms would incorrectly describe the sequence. Each description produced by the VL_1 induction algorithms must be checked to verify that it is consistent with the tail end of the layout.

The **third evaluation step** involves assessing the plausibility of the descriptions in terms of Eleusis. The complexity of each description must be measured (approximately). The average size of the set of legal cards must be measured in accordance with the plausibility criteria mentioned above. It must not be possible to reach a dead-end while playing according to the rule. Lastly, the description must be checked to see that it is consistent with negative string plays. Recall that, in Eleusis, if a player plays a string of cards (2, 3, or 4 cards), and any one of the cards is in error, the entire string is placed on a side-line below the main line. Although this information cannot be used during rule discovery, it is necessary to check each description developed by the VL_1 induction algorithms to see that it is consistent with these negative string plays. At least one of the cards in each negative string must be illegal according to the description.

Once the descriptions have passed through all of these evaluation steps, they must be converted to VL_{22}. In the SPARC/E(V.2) program, the discovered rules are maintained in a rule base along with rules which the user may have entered into the system. The rule base is consulted when the player wants to play a card.

- Identify Knowledge Requirements. First, we list the knowledge required for the interpretation steps, then for the generalization step, and finally for the evaluation steps. The first interpretation step (converting 2c to a VL_1 complex) merely requires knowledge of the card notation. The second step (adding color, parity, etc. to the events) requires knowledge of the definitions of the added variables. The user is able, in the SPARC/E(V.2) program, to enter the definition of a new variable as a VL_{22} complex. For example, color can be entered as:

```
define color = red [suit(card0)=hearts, diamonds],
                black [suit(card0)=spades, clubs];
```

In fact, this is a set of production rules.

$[suit(card0)=hearts] \vee [suit(card0)=diamonds] \rightarrow [color(card0)=red]$

$[suit(card0)=spades] \vee [suit(card0)=clubs] \rightarrow [color(card0)=black]$

The action of the program when the situation matches, is to assert the consequence into the layout.

The segmentation process requires knowledge of the ordering of the layout. The program must know how to compute the difference variables between adjacent events in order to determine that they satisfy the segmentation condition. This in turn requires knowledge of the domains of the variables. The segmentation process must also know how to segment negative events properly. Violation of either the segmentation condition or the segmented rule can cause a card to be illegal. However, at the time the layout is being segmented in preparation for rule discovery, only the segmentation condition is known.

The last interpretation step requires knowledge of the ordering of the layout so that the unordered events may be developed. Knowledge of how to compute sum and difference variables is obviously needed. This in turn requires knowledge of the domains and domain types of the variables. The last interpretation step prepares events for a specific model with a specific lookback parameter, so this information must be available.

The generalization steps require knowledge of the domains and domain types of the variables. In particular, the domain type-specific rules of generalization must be available during the generalization process. The decomposition algorithm requires knowledge of which variables are left-hand side variables. The algorithms also have a good deal of knowledge available in their cost functionals. The cost functions must measure the plausibility of the descriptions under development.

Knowledge required to evaluate the rules and remove irrelevant variables includes knowledge of the ordering of the events and knowledge of each rule model. The process which removes redundant difference variables must understand the relationship between the difference variables and the variables from which they were derived.

Knowledge required to test the tail end of the segmented layout for consistency is precisely identical to the knowledge required to initially segment the layout.

Knowledge required to estimate the average size of the set of legal cards includes knowledge of the relationships between variables and knowledge of how segmentation interacts with the description models. The process of verifying that each negative string play contains a bad card requires little knowledge beyond the knowledge of how negative string plays are handled in SPARC/E(V.2). The conversion of a VL_1 rule to a VL_{22} rule is a straightforward syntactic manipulation.

- Decompose the System into Layers. Figure 5 indicates the layers of the SPARC/E(V.2) system and the functions of each layer. The top-most layer (Layer 5) provides the menu driven user interface. It also has strategic knowledge of the game so as to act as an independent player. It also performs

Layer	Function	Knowledge Used
5	User Interface	Cards, VL22 syntax, Menus
4	Eleusis	Color, parity, negative strings, plausible Eleusis rules
3	Segmentation	Ability to segment layout, check tail end of segment, plausible segmentations
2	Sequential Analysis	Models and parameters, ordering in the layout sum and difference variables
1	Basic Induction	Domains and domain types, basic algorithms, cost functions

Figure 5. Architecture of the SPARC/E(V.2) System.

the first interpretation step by converting cards into VL_1 complexes.

Layer 4 contains Eleusis-specific knowledge including all of the knowledge of the relationships between variables and knowledge of negative string plays. This layer performs the second interpretation step by expanding the input events to contain all possible variables which might be relevant. This layer removes negative string plays from the layout and uses them later to evaluate the descriptions returned from layer 3. Layer 4 also converts VL_1 descriptions to VL_{22}. Knowledge of plausibility in SPARC/E(V.2) is used in this layer to evaluate descriptions according to the average number of cards playable under the rule.

Layer 3 performs all functions relating to segmentation. It segments the layout according to a list of possible segmentation conditions and evaluates each. Those which it finds to be promising it hands to layer 2 for further discovery. It evaluates descriptions returned from layer 2 to guarantee that they are consistent with the tail end of the layout.

Layer 2 performs the function of removing order from the layout. It computes the unordered VL_1 events including the sum and difference variables. For each model, it develops a specific set of events and passes them to layer 1 for generalization. Layer 2 filters the resulting descriptions to remove redundant selectors.

Layer 1 performs the basic generalization tasks described in Chapter 2. It implements the three VL_1 induction algorithms discussed above.

3.2.3. Other Functions

In addition to discovering plausible Eleusis rules, the SPARC/E(V.2) program in its interactive mode, provides other valuable services to its user. First, it permits the user to enter Eleusis rules in VL_{22} form. It checks those rules for consistency with the layout and adds them to its VL_{22} rule base. Second, it permits the user to enter the cards that s/he has in her or his hand. When the user issues an EVALUATE command, each rule in the VL_{22} rule base is processed against the layout to determine which cards are currently legal according to that rule. Each card in the player's hand which is currently legal is marked. The user can display this information in order to choose a card to play, or s/he can ask the system to suggest a card to play. The system plays according to two strategies. The conservative strategy is to play a card which is legal under as many rules as possible. The discriminant strategy is to play a card which will eliminate implausible rules from further consideration.

These additional functions require two major additions to the SPARC/E(V.2) program. Besides rule discovery, the program needs the ability to check a rule against the layout to determine if it covers all of the mainline and is consistent with all of the negative examples on the sidelines. This is called the Critic function. The program also needs the ability to determine which cards are legal extensions of the layout according to a given rule. This is called the Performance Element function.

3.3. Previous Sequential Data Analysis

Most prior work in sequential data analysis has sought to induce plausible grammars (or equivalently, automata) which could generate or extrapolate a sequence of events [17,40]. Grammars have advantages:

- Grammars provide a natural representation for segmented descriptions. A particular grammar rule can be used recursively by a new grammar rule. This solves the segmentation problem.
- Grammars are well-understood mathematically.

Grammars also possess disadvantages which make them difficult to use for describing sequential event sets:

- Grammars describe sequences of events by generating them. It is difficult to write a grammar which merely constrains the possibilities at a point. For example, to write a grammar which permits the next event to have an even value, we must write:

```
even -> 2 | 4 | 6 | 8 | 10 | 12

S -> even | S even
```

Furthermore, in order to extend a series, the start symbol, S, must be reduced to terminal symbols. All possible extensions of a series must be generated in order to develop a prediction.

- Grammars do not correspond to the way people describe sequential events. The grammar above describes a sequence of even numbers. I think people tend to describe such a sequence logically as

  $$x \quad \text{even}(x) \qquad x \text{ in the sequence.}$$

  It is important that a computational tool produce descriptions which are conceptually simple and in accordance with human-based descriptions. In the game Eleusis, typical rules are much more easily expressed in logic than as a grammar.

- Grammars lack many useful operations. Unadulterated grammars use only juxtaposition. Even in the augmented grammars used in general production systems, juxtaposition plays a major role. Yet a good description of a sequence of events is event-centered. The characteristics of the next event are described in terms of its immediate environment. For example, in R1 (Table 5) if the previous card is a club, we must play a diamond. In R4, the position of the next card in the layout determines whether it must be odd or even. These event-centered descriptions are very clear, and they make it very easy to compute legal extensions of the sequence. Such descriptions have grammatical counterparts, but these counterparts are rarely as succinct and clear.

4. SPARC/E(V.2) Rule Generator and Game Player

The program can be conceptually divided into two parts: a Rule Generator that generates plausible rules from a layout, and a Game Player that is equipped with all the necessary user interfaces to enable it to play the game with minimum assistance from human users. The Rules Generator encompasses the algorithms discussed in section two and section three, and essentially includes layer 1 throughout layer 4 of the of the knowledge layers. The Game Player constitutes the fifth layer of the system that sits on top of the Rule Generator. Its functions is to allow users to communicate with the program through menus, and maintain the progress of a game throughout a game session. In this section, the SPARC/E(V.2) Rule Generator and Game Player will be described.

4.1. The Implementation

The SPARC/E(V.2) program (Sequential PAttern ReCognition, Eleusis Version 2) is an extension of SPARC/E(V.1), originally written in Pascal on a CYBER 175 (Control Data Corporation) by Thomas Dietterich. The program was later transcribed to Berkeley Pascal running under 4.2 BSD UNIX operating system, which was where all the extension took place.

4.2. SPARC/E(V.2) Rule Generator

To show the function of the Rule Generator, the result of several sample runs without the Game Player interface are discussed in detail in this section. These example are based on actual games played by the authors.

It is intended that the program be run with a standard, relatively conservative, set of parameters. If the user of the program is dissatisfied with the results obtained using those parameters, then they may be changed to increase the power of the program. Ideally, the program would make such decisions based on knowledge of what constitutes good Eleusis rules. It would be very nice, for example, if the system demonstrated satisfactory behavior. It could examine the simplest (and computationally cheapest) possibilities first and then move on to more complex description possibilities if the simple ones did not work. The system would stop searching as soon as it found a few plausible rules.

However, at present, the user of the program indicates a space of possibilities by setting parameters and the program searches that space and returns all plausible rules found. Five segmentation conditions were given to the program to investigate. These are:

[value(card0)=value(card1)]

[suit(card0)=suit(card1)]

[value(card0)=value(card1)+ 1]

[parity(card0)=parity(card1)]

[color(card0)=color(card1)].

For segmented rules, the system was told to investigate only a degenerate form of the periodic model. This degenerate period has a lookback of one and only one phase. Such a degenerate periodic description can be used when a single conjunctive description of the layout is desired. The program was given the following relevant descriptors:

Color: Color of the card.

Face: True if the card is a faced (picture) card, false otherwise.

Prime: True if the card has a prime value, false otherwise.

Parity: Takes the value of even or odd.

Mod3: Takes on the value of the card modulo three. This is an example of a "noise" descriptor since it is very unlikely that it will be involved in any plausible descriptions.

Lenmod2: Takes on the value of the length of a subsequence, modulo 2.

4.2.1. Example 1

Below is the layout for the first example rule. It is a very simple rule and the program discovers three equivalent descriptions for it:

```
main line  jc   ad   qh   10s  qd   9h   qc   7h   qd   9d   qc   3h   kh
           kc   5s                  4s        10d
                7s

main line  4c   kd   6c   jc   8d   jh   7c   jd   7h   jh   6h   kd
```

The program discovered the following descriptions of this layout:

```
rule  1: lookback: 1 nphases: 0 decomp
         [face(card1) =false] => [value(card0) >value(card1)] v
         [face(card1) =true] => [value(card0) <value(card1)]
                                [face(card0) =false]

rule  2: lookback: 1 nphases: 1 periodic
         period([face(card0) <>face(card1)])

rule  3: lookback: 1 nphases: 2 periodic
         period([value(card0) >=-value(card1)+20],
         [value(card0) =3..10][value(card0) =-value(card1)+5..17])
```

Rule 1 expresses the rule as a decomposition rule with a lookback of 1. Most periodic rules which have disjoint phases can be expressed as decomposition rules. Rule 2 expresses the rule as a single conjunction. This is possible because face vs. non-face is a binary condition and there are precisely two phases to the rule. Rule 3 expresses the rule in the "natural" way as a periodic rule of length 2.

4.2.2. Example 2

This example shows what happens when the phases of a period are not strictly disjoint. Recall that the program seeks symmetrical, disjoint descriptions for the phases of a period and for the if-then cases of a decomposition rule. The rule intended by the dealer was "play a periodic rule where the first phase may be either a spade or a heart, and the second phase may be either a diamond or a heart." The layout for the game was:

```
main line  9s   4d   kh   3d   ks   5d   as   2d   kh   6h
                9c   4c   5c   qs                  6s
                7c

main line  qs   ah   ah   10d  7s   7h
                          jc
                          jd
```

The program discovered the following rules using the decomposition and periodic rule models:

```
rule  1: lookback: 0 nphases: 1 periodic
      string = [parity(card0) =parity(card1)] :
      period([length(string0) =1,2,5])

rule  2: lookback: 1 nphases: 2 periodic
   period([value(card0) >=-value(card1)+ 6]
         [suit(card0) =hearts..spades]
         [suit(card0) =suit(card1)+ 3..1]
[mod3(card0) =0..1]
[mod3(card0) =-mod3(card1)+ 1..2],
[value(card0) <=10]
[value(card0) <>value(card1)]
[value(card0) =-value(card1)+ 5..15]
[suit(card0) =diamonds..hearts]
[suit(card0) =suit(card1)+ 3..1]
[color(card0) =red]
[color(card0) =color(card1)]
[face(card0) =false]
[face(card0) =face(card1)]
[mod3(card0) =-mod3(card1)+ 1..2])
```

The first rule is absolutely miserable. Because the plausibility evaluation part of the program is only partially implemented, this rule manages to make its way up to the top level. The rule says that the main line is made up of strings of cards which have the same parity. These strings are either 1, 2, or 5, cards in length. Under the SPARC/E(V.2) knowledge of plausibility, this rule would be eliminated because there are many times when any card is legal.

The second rule is not much better. One can see that the dealer's rule was discovered (e.g. [suit(card0)= diamonds..hearts]), but when the periodic algorithm attempted to remove overlapping selectors, it removed the significant selectors along with the insignificant ones. Recall that the algorithm backs up in such cases and returns the ungeneralized rule.

Since these descriptions were so bad, the program was instructed to examine a DNF model for this game. The following rule was discovered:

```
rule  2: lookback: 1 nphases: 0 dnf
  [value(card0) <=-value(card1)+ 16][suit(card0) =diamonds..spades] v
  [suit(card0) =hearts]
```

This rule states that hearts are always legal, and that if the sum of the values of the current card and the previous card is less than or equal to 16, then the current card may be a diamond or spade. Although this rule is incorrect, it does serve the useful purpose of isolating the relevant variables. A user of the program might then be able to identify the rule.

It is clear that the program does not handle asymmetrical rules well. The DNF model is able to isolate relevant variables even though the rule it discovered will lead to incorrect play.

4.2.3. Example 3

In this example, we show the program discovering a segmented rule. Notice that in the previous rules, although several segmentation conditions were suggested, only one (very poor) segmented rule was discovered. Included in the parameters for these example sessions are the plausibility limits for segmentation. These were set so that a segmentation of the layout must produce at least 5 segments, and the number of events in the segmented layout must be no more than half the number in the original layout. These plausibility limits have been very successful in weeding out unpromising segmentations. Furthermore, in all of our testing of the program, never has a segmentation condition been erroneously eliminated from further consideration.

The layout for this example is:

```
main line  ah 7c 6c 9s 10h 7h 10d jc ad 4h 8d 7c 9s 10c ks 2c 10s js as 5c kc
                 kd            5s     qd    3s          9h      qh
                 jh                                     6h      ad
```

The program only discovered one rule for this layout, precisely the rule which the dealer had in mind:

```
rule  1: lookback: 0 nphases: 1 periodic
    string = [color(card0) =color(card1)] :
    period([lenmod2(string0) =1])
```

The rule states that one must play strings of cards with the same color. The strings must always have odd length. Actually, the rule which the dealer had in mind had one additional constraint: a queen must not be played adjacent to a jack or king. This is a type of exception-based description. The program cannot handle such exceptions. This is a problem for further research (see below).

4.2.4. Example 4

The layout is taken from Abbott's rules for Eleusis.

```
main line  3h   9s   4c   jd   2c   10d  8h   7h   2c   5h
                jd        ah   as             10h
                5d        8h   10s
                          qd
```

The program discovered two rules to explain the layout. The first rule is very close to the rule described by Abbott:

```
rule  1: lookback: 1 nphasesl: 0 decomp
[parity(card1) =odd] => [color(card0) =black] v
```

```
[parity(card1) =even] => [color(card0) =red]

rule 2: lookback: 1 nphases: 2 periodic
period([value(card0) =2..8][value(card0) <>value(card1)]
[value(card0) =-value(card1)+ 4..8]
[suit(card0) =clubs..hearts][suit(card0) =suit(card1)+ 0..2]
[face(card0) =false]
[face(card0) =face(card1)][prime(card0) <>prime(card1)]
[parity(card0) =even]
[mod3(card0) =1..2][mod3(card0) =mod3(card1)+ 0..1]
[mod3(card0) =-mod3(card1)+ 0..1],
[value(card0) =5..jack][value(card0) <>value(card1)]
[value(card0) =-value(card1)+ 10..19]
[suit(card0) =diamonds..hearts][suit(card0) =suit(card1)+ 0..2]
[color(card0) =red]
[mod3(card0) =1..2][mod3(card0) =-mod3(card1)+ 2..0])
```

The second rule is worthless!

4.2.5. Example 5

```
current layout:
main line  4c   8c   6c   10c  2h   qh   qh   qs   4c   10s  2s   as   6h
                     jc       (9h   ad   5d                       qs   8s
                               4d )      10h


main line  qd   kc   9c   2c   10c  2c   6c   10d  jh
               (jd             jc
                ks )
```

The program discovered the following rule:

```
rule 1: lookback: 1 nphases: 0 decomp
[color(card1) =red] => [face(card0) =true] v
[color(card1) =black] => [face(card0) =false]
```

4.2.6. Example 6

```
main line  4h   2h   ah   qs   ks   8h   9h   5c   4d   5d
           10s  as   3h        kc   4s   7d   7c        8d
           7c   jc                       jd
                                         3h
                                         kh
```

The program discovered the following rule:

```
rule 1: lookback: 1 nphases: 0 decomp
[parity(card1) =odd] => [suit(card0) =suit(card1)+ 1..3][color(card0) <>color(card1)] v
[parity(card1) =even] => [suit(card0) =suit(card1)+ 0][color(card0) =color(card1)]
```

4.2.7. Example 7

```
current layout:
main line  8s    3d    6s    10c   jh    9d    ad    2c
           ac          js    as          ks
                             2h          7s
                                         6d
```

The program discovered the following rule:

```
rule 1: lookback: 0 nphases: 0 dnf
[color(card0) =red][parity(card0) =odd] v
[color(card0) =black][parity(card0) =even]
```

4.2.8. Example 8

This example shows the upper limits of the program's abilities. During this game, only one of the human players even got close to deducing the rule, yet the program discovers a good approximation of the rule using only a portion of the layout that was available to the human players. Here is the layout:

```
main line    4h   5d   8c   js   2c   5s   ac   5s   10h
             7c   6s   kc   ah        6c        as
             jh   7h   3h   kd
             4c   2c        qs
            10s   7s
             8h   6d
             ad   6h
             2d   4c
```

The program was told, in this game, to check all three models. It produced the following rules:

```
rule 1: lookback: 1 nphases: 0 dnf
[value(card0) <=5][suit(card0) =suit(card1)+ 1] v
[value(card0) >=5][suit(card0) =suit(card1)+ 3]

rule 2: lookback: 1 nphases: 1 periodic
period([value(card0) =value(card1)-9]
    [value(card0) =-value(card1)+ 4,5,7,11,13,17]
    [suit(card0) =suit(card1)+ 1,2,3])

rule 3: lookback: 1 nphases: 2 periodic
period([value(card0) =ace,2,8,10]
    [value(card0) =-value(card1)+ 1,8,9,10],

    [value(card0) =5..jack][value(card0) =value(card1)+ -0..6]
    [value(card0) =-value(card1)+ 8..14]
    [suit(card0) =spades][suit(card0) =suit(card1)+ 0..2]
    [color(card0) =black]
    [prime(card0) =ptrue][prime(card0) =prime(card1)]
    [parity(card0) =even][parity(card0) =parity(card1)]
    [parity(card0) =parity(card1)][mod3(card0) =2]
    [mod3(card0) =mod3(card1)+ 0][mod3(card0) =-mod3(card1)+ 1])
```

The rule which the dealer had in mind was:

```
[suit(card0)=suit(card1)+ 1][value(card0)>=value(card1)] v
[suit(card0)=suit(card1)+ 3][value(card0)<=value(card1)]
```

It is very likely that a player could have deduced the correct rule once he/she had seen the rule produced by the program. The program has isolated the relevant variables, and has produced a very plausible description. Note that adding three to a suit gives the next lower suit in the cyclic interval domain of suits.

4.2.9. Example 9

This example is also not based on actual game. The layer 2 learning element invokes periodic model recursively and stops when the length of the layout is too small.

```
main line  2s   4c   kd   ks   7h   qd   3s   8c   2d   9s   5h   4d   6s
                js             as   kc   2h        4h   8d   6h
                10c                                ad        9c
                                                             ah
                                                             qs


main line  jc   9d   9s   8h   6d   3s   10c  6d   as
                                    10s  2c
```

The program was told to examine periodic model with 1 or 2 or 3 phases. It produced the following rule:

```
rule  1: lookback: 0 nphases: 3 periodic
period([suit(card0) =spades],
      Embedded rule: lookback: 0 nphases: 2 periodic
      period([suit(card0) =clubs],
            [suit(card0) =hearts]),
      [value(card0) >=2][suit(card0) =diamonds])
```

Without the recursion, the program produced [suit(card0)=clubs,hearts] in the second phase.

4.3. SPARC/E(V.2) Game Player

During a game session SPARC/E(V.2) is capable of playing as an independent player and competes with other human players. The user of the program must help reporting the progress of the game to the program through the use of menus, and executing the program's decision by physically play a card from the program's hand of cards. When it is the turn for the program to play, the program will play a card upon request from the user. It then receive from the user the judgement concerning the correctness of the card played, together with penalty cards if any, and modify the internal representations to reflect the changes that have been made.

The Menus

The menus are organized into a tree structure, where each node in the tree as a lower level menu, and the leaves of the tree as actions to be executed. Selecting a lower level menu bring you down the tree one level deeper. Only one menu at a time is displayed.

The **top level menu** consists of 11 entries, as shown in figure 1, which can be divided roughly into three categories according to their functions: game playing commands, help commands and parameter adjusting commands. Game playing commands include entries 1, 2, 3, 4, 5, 6 and 10 which are the commands that will normally be used in playing the game. Help commands include entries 7 and 11, which provide online information for the user. Entries 8 and 9 are parameter adjusting commands that allow the user to fine tune the system parameters. Parameter adjusting commands allow the user to direct the Rule Generator of SPARC/E(V.2) to instantiate rule models in certain particular way, and search the rule space according user specified criteria. It is thus advisable that a user have some knowledge about the theory behind the program in order to understand and make use of these commands. Command number 9 (Interactive Eleusis) allows the program user to bypass the menus and communicate interactively with the Rule Generator. This command is reserved for users that know his way in the program. Casual user are advised not to invoke this command. See appendix 2 for detail descriptions of parameter adjusting commands.

1. Add new card(s) to the layout
2. Play a card
3. Show me your hand
4. Show me the layout
5. Show me your rules
6. Add cards to your hand
7. Help
8. Accept my advice
9. Interactive Eleusis (for advanced users)
10. Quit
11. Rules of the game

Figure 1. The top level menu

In general, it is assumed that the user of the program is "talking" to the program, thus the "you" in the menu commands indicates the program, and "me" indicates the program user. For example, the command "Show me your rules" is used to ask SPARC/E(V.2) to display the rules it has found.

The other menus at different levels have a structure similar to that of top level menu's, the meaning of most entries are rather simple and self-explanatory. Here we shall elaborate only on the second command "Play a card" where most game playing strategies reside.

When a card is selected and played by the program, the program use is asked to specify if the card has been declared correct or incorrect by the dealer. If the card played is declared a negative card, the program will ask for the penalty cards to be added to its hand. If the program has no playable card in its hand, the No Play option will be considered.

Play a card

When the program user request SPARC/E(V.2) to play a card, the rule base is investigated to see if the rules (if any) predicted in the previous play is still valid. If not, the rulebase is cleared and a new set of rules are generated by invoking the induction algorithm from scratch. The hand is then evaluated against the rule base, and a card is played according to the strategies explained in the following section.

Strategies

SPARC/E(V.2) will play cards in its hand that satisfy the rules produced by the Rule Generator. There are two options if there is no playable card in hand: either declare No Play, or assume that the rules produced by the Rule Generator are wrong and play a card randomly. Since the stake of declaring **No Play** is rather high, we use the following rules to make the decision when none of the cards in the hand satisfies the rules:

(1) If there exists at least a rule that looks good, and the number of cards in the hand is few, then the program will declare **No Play**.

(2) Otherwise a card is played randomly from the hand. This is because that even if some rules look plausible, if the hand contains too many cards, then it is unlikely that none of them are legal. So there is a higher chance of being penalized if No Play is declared.

REFERENCES

[3] Abbott, Robert, "The New Eleusis," Available from Abbott at Box 1175, General Post Office, New York, NY 10001 ($1.00).

[4] Barto, A. G., J. M. Prager, "Forming Logically Simple Hypotheses in Parallel," Paper submitted to IJCAI-6, University of Massachusetts, Amherst, 1979.

[5] Box, G. E. P., G. M. Jenkins, **Time-Series Analysis: Forcasting and Control,** Revised Edition, Holden-Day, San Francisco, 1976.

[6] Buchanan, B. G., E. A. Feigenbaum, J. Lederberg, "A Heuristic Programming Study of Theory Formation in Science," in **Proceedings of the Second International Joint Conference on Artificial Intelligence,** 1971, pp. 40-48.

[7] Buchanan, B.G., D. H. Smith, W. C. White, R. J. Gritter, E. A. Feigenbaum, J. Lederberg, C. Djerassi, **Journal of the American Chemical Society** 98 (1976) p. 6168.

[8] Buchanan, B. G., E. A. Feigenbaum, "Dendral and Meta-Dendral, Their Applications Dimension," **Artificial Intelligence** 11 (1978) pp. 5-24.

[9] Buchanan, B. G., T. M. Mitchell, R. G. Smith, C. R. Johnson, Jr., "Models of Learning Systems," in **Encyclopedia of Computer Science and Technology,** J. Belzer, A. G. Holzman, and A. Kent, eds., Marcel Dekker, Inc., New York, 1977. (also available as HPP memo 77-39, Heuristic Programming Project, Stanford Universit, Stanford, CA).

[10] Chilausky, R., B. Jacobsen, and R.S. Michalski, "An applicaton of Variable Valued Logic to Inductive Learning of Plant Disease Diagnostic Rules," in **Proceedings of the Sixth Annual Symposium on Multiple Valued Logic** Logan, Utah, 1976.

[11] Dietterich, Thomas G., R. S. Michalski, "Learning and Generalization of Characteristic Descriptions: Evaluation Criteria and Comparative Review of Selected Methods," **Proceedings of the Sixth International Joint Conference on Artificial Intelligence,** pp. 223-231, Tokyo, August 1979.

[12] Erman, L. D., V. R. Lesser, "A Multi-level Organization for Problem Solving Using Many, Diverse, Cooperating Sources of Knowledge," **Advance Papers of the Fourth International Joint Conference on Artificial Intelligence,** MIT, Cambridge, MA, 1975.

[13] Freuder, E., "A Computer System for Visual Recognition Using Active Knowledge," PhD thesis, AI-TR-345, The Artificial Intelligence Laboratory, MIT, Cambridge, Massachusetts, 1976.

[14] Gardner, Martin, "On Playing the New Eleusis, the game that simulates the search for truth," **Scientific American,** 237, October, 1977, pp 18-25.

[15] Hayes-Roth, F., "Collected Papers on the Learning and Recognition of Structured Patterns", Department of Computer Science, Carnegie-Mellon University, Jan. 1975.

[16] Hayes-Roth, F., "Patterns of Induction and Associated Knowledge Acquisition Algorithms," Department of Computer Science, Carnegie-Mellon University, May 1976.

[17] Hayes-Roth, F., J. McDermott, "Knowledge Acquisition from Structure Descriptions", In **Proceedings of the Fifth International Joint Conference on Artificial Intelligence,** 1977, pp. 356-362.

[18] Hayes-Roth, F., J. McDermott, "An Interference Matching Technique for Inducing Abstractions", **Communications of the ACM,** 21:5, 1978, pp. 401-410.

[19] Hedrick, C. L., "A Computer Program to Learn Production Systems Using a Semantic Net," PhD. Thesis, Department of Computer Science, Carnegie Mellon University, Pittsburgh, Pa., 1974.

[20] Hunt, E.B., **Experiments in Induction,** Academic Press, 1966.

[21] Larson, J., "A Multi-Step Formation of Variable Valued Logic Hypotheses," in **Proceedings of the Sixth International Symposium on Multiple-Valued Logic,** Logan, Utah, 1976.

[22] Larson, J., and R. S. Michalski, "Inductive Inference of VL Decision Rules," **SIGART Newsletter,** June 1977, pp. 38-44.

[23] Larson, J.; 'Inductive Inference in the Variable Valued Predicate Logic System VL21 : Methodology and Computer Implementation', Rept. No. 869, Dept. of Comp. Sci., Univ. of Ill., Urbana, May 1977.

[24] Lenat, D., "AM: An artificial intelligence approach to discovery in mathematics as heuristic search," Comp. Sci. Dept., Rept. STAN-CS-76-570, Stanford University, July 1976.

[25] Michalski, R. S., "Algorithm Aq for the Quasi-Minimal Solution of the Covering Problem," **Archiwum Automatyki i Telemechaniki,** No. 4, Polish Academy of Sciences, 1969 (in Polish).

[26] Michalski, R.S., "A Variable-Valued Logic System as Applied to Picture Description and Recognition," in **Proceedings of the IFIP Working Conference on Graphic Languages,** Vancouver, Canada, 1972.

[27] Michalski, R. S., "Conversion of Normal Forms of Switching Functions into Exclusive-Or-Polynomial Forms," Archiwum Automatyki i Telemachaniki, No. 3, Polish Academy of Sciences, 1971a (in Polish).

[28] Michalski, R. S., "DISCOVERING CLASSIFICATION RULES USING VARIABLE-VALUED LOGIC SYSTEM VL1," Advance Papers of the Third International Joint Conference on Artificial Intelligence, Stanford University, Stanford, CA, pp162-172.

[29] Michalski, R.S. "Pattern Recognition as Knowledge-Guided Induction," Rept. 927, Dept. of Comp. Sci., Univ. of Ill. Urbana, 1978.

[30] Michalski, R. S., J. Larson, "SELECTION OF MOST REPRESENTATIVE TRAINING EXAMPLES AND AN INCREMENTAL GENERATION OF VL1 HYPOTHESES: the underlying methodology and description of programs ESEL and AQ11," Report No. 867, Department of Computer Science, University of Illinois, Urbana, May 1978.

[31] Michalski, R. S., "Variable-valued logic and its application to pattern recognition and machine learning," In **Computer Science and Multiple-Valued Logic,** ed. D. C. Rine, North-Holland, 1977, pp. 506-534.

[32] Michalski, R. S., "VARIABLE-VALUED LOGIC: System VL1," **1974 International Symposium on Multiple-Valued Logic,** West Virginia University, Morgantown, West Virginia, May 29-31, 1974.

[33] Michie, D., "Measuring the Knowledge-Content of Programs," University of Illinois, Department of Computer Science Report UIUCDCS-R-76-786, May 1976.

[34] Michie, D., "New Face of AI," Experimental Programming Repts.: No. 33, MIRU, Univ. of Edinburgh, 1977.

[35] Schwenzer, G. M., T. M. Mitchell, "Computer-assisted Structure Elucidation Using Automatiically Acquired Carbon-13 NMR Rules," in ACS Symposium Series, No. 54, 'Computer-assisted Structure Elucidation,' D.H. Smith (ed), 1977.

[36] Soloway, E., E. M. Riseman, "Knowledge-Directed Learning," in "Proceedings of the Workshop on Pattern Directed Inference Systems," **SIGART Newsletter,** June 1977, pp 49-55.

[37] Soloway, E., "Learning = Interpretation + Generalization: a case study in knowledge-directed learning," PhD Thesis, COINS TR 78-13, University of Massachusetts, Amherst, MA., 1978.

[38] Vere, S.A., "Induction of Concepts in the Predicate Calculus," In **Advance Papers for the Fourth International Joint Conference on Artificial Intelligence,** 1975.

[39] Vere, S. A., "Induction of Relational Productions in the Presence of Background Information," In **Proceedings of the Fifth International Joint Conference on Artificial Intelligence,** MIT, Cambridge, MA., 1977.

[40] Vere, S. A., "Inductive Learning of Relational Productions", in **Pattern-Directed Inference Systems,** D.A. Waterman and F. Hayes-Roth (eds), Academic Press, 1978.

[41] Vere, S. A., "Multilevel Counterfactuals for Generalizations of Relational Concepts and Productions," Department of Information Engineering, University of Illinois, Chicago Circle, 1978.

[42] Waterman, D. A., "Serial Pattern Acquisition: A Production System Approach," working paper No. 286, Department of Psychology, Carnegie Mellon University, Pittsburgh, Penn., 1975.

[43] Winograd, T., **Understanding Natural Language** PhD thesis, Academic Press, 1972.

APPENDIX I

Input Grammar for Interactive SPARC/E(V.2) Commands

This grammar describes the valid syntax of all commands and rules typed in the Interactive Mode. The following differences should be noted between this grammar and that of VL_{21}. Firstly, this grammar permits functions and operators in the reference of a selector. The function may have an optional unary minus sign in front of it. Secondly, this grammar permits all selectors to use relations such as >=, <>, <, etc.

```
1 session ::= commandlist

2 commandlist ::= command
3         | commandlist ; command

4 command ::= HELP
5         | INDUCE
6         | EVAL
7         | PLAY
8         | Q
9         | CARD cardlist : ID
10 | UNCARD
11 | LIST listitem
12        | RULE vl2rule
13        | DELETE cardlist
14        | MINE cardlist
15        | STRATEGY ID
16        | KILL NUMBER
17        | DEFINE ID defdomain = deflist
18        | ADVICE advice
19        |            /* empty */

20 cardlist ::= SYCARD
21        | cardlist SYCARD

22 listitem ::= MINE
23 | STRATEGY
24 | ADVICE
25 | RULE
26 | ID                /* for other options */

27 vl2rule ::= segdefn ruledefn

28 segdefn ::= ID = simpleconjunct :  /* ID must be 'string' */
29        |     /* empty */
```

```
30 ruledefn ::= dcrule
31         | periodicrule

32 dcrule ::= conjunct
33         | dcrule V conjunct

34 periodicrule ::= PERIOD ( sconjunctlist )

35 sconjunctlist ::= simpleconjunct
36         | sconjunctlist , simpleconjunct

37 conjunct ::= simpleconjunct
38         | simpleconjunct => simpleconjunct

39 simpleconjunct ::= selector
40         | simpleconjunct selector

41 selector ::= [ referee rop reference ]

42 referee ::= ID ( varlist )

43 varlist ::= ID
44         | varlist , ID

45 rop ::= =
46         | <>
47         | >=
48         | <=
49         | >
50         | <

51 reference ::= value
52         | sign ID ( varlist ) moreref

53 value ::= valuelist
54         | NUMBER .. NUMBER

55 valuelist ::= valueentry
56         | valuelist , valueentry

57 sign ::= -
58         |     /* empty */

59 valueentry ::= NUMBER
60         | VALUE         /* a defined reference value */

61 op ::= +
62         | -
63         | +-

64 moreref ::= op value
65  |

66 defdomain ::= ( ID , NUMBER )
67         | ( ID )
```

```
68        |                  /* empty */

69 deflist ::= def
70        | deflist , def

71 def ::= defvalue simpleconjunct

72 defvalue ::= ID
73        | NUMBER

74 advice ::= PARAMETERS params
75        | SEGMENTS sconjunctlist
76        | PLAUS plauslist
77        | DOMAIN domainlist
78        | ID tokenlist

79 tokenlist ::= token
80        | tokenlist token

81 token ::= ID
82        | NUMBER

83 params ::= param
84        | params , param

85 param ::= - parm1
86        | parm1

87 parm1 ::= NUMBER
88        | NUMBER ( NUMBER )         /* cost with tolerance */

89 plauslist ::= plaus
90        | plauslist , plaus

91 plaus ::= ID = NUMBER

92 domainlist ::= domain
93        | domainlist , domain

94 domain ::= ID = SYID
```

APPENDIX II

Interactive Eleusis Commands

Here is a synopsis of the user commands for the Interactive Eleusis, which can be entered through the entry number 9 of the Top Level Menu. The commands can be broken down into five categories: layout management, managing the hand, managing the rule base, the learning element, and the performance element. The command input to the Eleusis program is free-format. Each command must be terminated by a semi-colon. When the program is ready for a command, it types:

eleusis tool ready (nnn ms)

where nnn is the number of milliseconds required for the previous step. If a command is not yet terminated (e.g. missing its ';'), the program just prompts with a question mark. In the descriptions below, optional entries are placed in brackets.

Layout Management Commands

c[ard]

The **card** command adds a string of cards to the layout. One **card** command should be used for each turn of a player in the game. The syntax of the command is:

```
c[ard] cardlist : judgment;
```

where cardlist is a list of cards of the form '2c' or 'qs' separated by spaces. Judgment indicates the dealer's judgment concerning the correctness of the play: 'y' indicates the cards are correct, 'n' indicates the cards are incorrect.

u[ncard]

The **uncard** command is the reverse of the **card** command. It removes from the layout the string of cards added by the most recent **card** command. It may be used repeatedly to undo several **card** commands.

list layout

This command lists the layout horizontally along the page. The first card played is in the upper left-hand column. The correct cards (mainline) are laid from left to right. The incorrect cards are listed across the page below the correct card which they followed. Negative string plays (a string of cards declared to contain an error) are listed in parentheses.

Hand Management Commands

m[ine]

The **mine** command (or **hand**, a synonym) adds cards to the player's hand. The cards are simply listed after the command separated by spaces:

```
    mine ac 6s 9d jc 10c qs;
```

d[elete]

The **delete** command removes cards from the player's hand. The cards are simply listed after the command.

```
    delete kc 2c;
```

list m[ine]

Use **list mine** to list the contents of your hand. See the section on the performance element below for an interpretation of the cards vs. rules matrix that is printed in the listing. The cards in the hand are listed along the left-hand column.

Rule Management Commands

r[ule]

The **rule** command permits the user to enter a rule in VL_{22}. Refer to the grammar in appendix I for details concerning rule syntax. An example of a **rule** command is:

```
rule period( [color(card0)=red], [color(card1)=black]);
```

Of course the rule can go on for more than one line. It is terminated by the ';'. When a rule is entered, it is immediately checked to see if it is consistent with the layout. This invokes the Critic function of the Eleusis program. A line will be printed indicating whether or not the rule is consistent with the layout. If the rule is inconsistent, some information concerning the source of the inconsistency is also printed. Then the rule is added to the VL_{22} rule base. The rule base is used by the performance element (see below).

i[nduce]

The **induce** command discovers rules that describe the layout and adds those rules to the rule base. Most of the relevant information is listed under the learning element below.

list r[ules]

Use **list rules** to see the rules (and their assigned numbers) in the rule base. Rules remain in the rule base until deleted by the **kill** command. Each rule is assigned a number. The number is used in the **list hand** printout, and it is used to report information concerning the rule during the rule evaluation process.

k[ill]

The **kill** command serves to delete rules from the VL_{22} rule base. Often a bad rule gets into the rule base, usually the result of an **induce** command. To delete poor and implausible rules, a **kill** command may be used. The **kill** command accepts one number, the number of a rule to be deleted:

```
kill 5;
```

This deletes rule number 5. No acknowledgment is printed. In order to determine the numbers corresponding to the rules, use the **list rules** command.

The Performance Element

s[trategy]

The strategy used by the program is entered using the **strategy** command. This strategy is used by the **play** command to choose a card from the hand to play. There are two strategies. The **conservative** strategy directs **play** to select the card which is legal under the largest number of rules in the rule base. The **discriminant** strategy directs **play** to select a card which will discriminate between the rules listed in the VL_{22} rule base. **play** attempts to select a card which is covered by approximately half of the rules. It is wise to play **discriminant** early in the game, and **conservative** after the first 30 cards have been played.

e[valuate]

The **evaluate** command instructs the program to evaluate each rule in the rule base to determine what cards are currently playable under that rule. This information is then used to determine which cards currently in the player's hand are playable under each rule. This information can be printed out using the **list hand** command. **list hand** prints a matrix of cards in the hand versus rules in the rule base. A 'y' indicates that the card on that row is legal according to the rule in that column. The columns are numbered according to the rule numbers of the rules in the rulebase (See Chapter 4).

p[lay]

The **play** command instructs the program to choose a card to play according to the current strategy. See the **strategy** description above for details of how the selection is made. The program makes the selection based on the most recent **evaluate** command. Thus, one should always precede a **play** command by an **evaluate**. If no card in hand is playable, **No Play** will be considered. After a card is played, the program will asked about the decision being made, and if the card played is a negative card, the penalty cards to be added to its hand.

The Learning Element

The learning element is the most complicated part of the program to use. The user must set up a collection of parameters which delimit the space of possible rules. When an **induce** command is given, the program searches this space for plausible rules describing the main line. We shall examine the parameters from the perspective of the layers.

Layer 4 parameters

def[ine]

define may be used to add new descriptors to the program. The program initially only has knowledge of **suit**, **value**, and **length** . To add color to the program, we would type:

```
define color (nominal, 50) =
   red [suit(card0)=diamonds, hearts],
   black [suit(card0)=spades, clubs];
```

To add value modulo 2 to the program, we would type:

```
define valmod2 (clinear, 50) =
  0 [value(card0)=2,4,6,8,10,12],
  1 [value(card0)=1,3,5,7,9,11,13];
```

The variables must be limited to 10 characters (only the first 10 characters are used). The notations **nominal** and **clinear** define the domain type of the variable. The **50** gives the plausibility for this variable (see below). After the = sign, we give a list of value-complex pairs separated by commas . Each value (either a symbol or a number) is defined by the VL_{22} complex which follows it. The complex may use any variables previously defined. The dummy variables used in the complex (in this case **card0**) determine what this new variable will be applied to (i.e. cards or strings).

Level 3 Parameters

a[dvice] seg[mentation]

This command gives a list of all segmentation conditions the program should examine. As indicated in the diagram above, the null segmentation (left-most branch) is always investigated. Additional segmentations may be added by typing:

```
a seg [color(card0)=color(card1)],
[value(card0)=value(card1)+ 1];
```

All segmentation conditions are selectors which have a variable in the reference. The segmentation condition must express the difference between two variables. A segmentation condition may have more than one selector in it. Segmentation conditions must apply to cards not strings. The segmentation conditions given by each A **seg** command completely replace the previous segmentation list.

a segplaus

This controls pruning of unpromising segmentations. After each segmentation has been performed on the layout, it must satisfy two tests. First, the segmented layout must have at least <minsegplaus> number of events in it. Second, the segmented layout must have no more than <maxsegplaus>*<size of unsegmented layout>/100 events in it. The <minsegplaus> and <maxsegplaus> parameters are given as:

```
a segplaus <minsegplaus> <maxsegplaus>
```

Level 2 Parameters

a models

This command tells level 2 and level 1 which models to investigate. The possible models are **dnf**, **decomp**, and **periodic**. They are always investigated in that order. Example:

```
a models periodic decomp;
```

This tells the program to investigate only the decomposition and periodic models. Each **a models** command completely replaces the previous setting of the **models** list.

a lookback

For the **decomp** and **dnf** models, the possible settings from the lookback parameter are determined by the **a lookback** command. The command provides two numbers, a minimum and a maximum lookback:

```
a lookback 0 2;
```

To set lookback for periodic rules use:

a plookback

This gives the minimum and maximum lookbacks for periodic rules. Recall that a lookback in a periodic rule looks back to the prior occurrences of each phase, not on a card by card basis.

a phase

This determines the possibilities for the number of phases to be examined for periodic rules. It is given as:

```
a phase <min> <max>
```

<min> must be at least 1.

Layer 1 parameters

The layer 1 parameters differ for each model. First we list the parameters applicable to the decomposition model, then to the **dnf** model. The periodic model has no additional parameters at this layer.

a complex

This is a general parameter which applies to both the **dnf** and decomposition models. It indicates, for **dnf**, the maximum number of complexes that can appear in the solution. If the Aq algorithm has not found a solution before it reaches this quota, it gives up. For the decomposition algorithm, it indicates the maximum number of variables to be decomposed on (i.e. the maximum number of selectors to appear on the left-hand side of each if-then rule). It is specified as:

```
a complex 4;
```

a dec

This command specifies the parameters for the decomposition functional sort. See the body of this report for the meanings of the cost functions. They are entered in order of evaluation, with tolerances in parentheses:

```
a dec 1(20),2,-3,4;
```

This specifies that cost function 1 is to be applied, with a tolerance of 20%. Then cost function 2 will be used. Then cost function 3 will be used, but first its value will be negated. Finally cost function 4 will be used to resolve ties still existing after the first three cost functions have been applied.

a decgen

This determines when the best trial decomposition is selected. If **decgen** is 0, the selection takes place immediately after the references are unioned. If **decgen** is 1, the references are generalized according to domain specific rules of generalization, and then the best decomposition is selected. If **decgen** is 2, overlapping selectors are removed, and then the best decomposition is selected. Example:

```
a decgen 1;
```

This is the recommended value for this parameter. If **decgen** is 2 and the rule does not fit the decomposition model, the program tends to run out of memory space. For the **dnf** model, the following parameters may be used:

a aq

This sets the Aq cost functional. The cost functions and their meanings are:

(1) Number of new events (events not covered by any previous star) covered by this complex in the set of positive examples.

(2) Total number of positive examples covered by this complex.

(3) Total number of negative examples covered by this complex.

(4) Number of non-irrelevant selectors in this complex.

(5) Sum of the costs of the non-irrelevant selectors in this complex. The cost of a selector is the plausibility of its variable subtracted from 100.

(6) Number of non-irrelevant selectors that this complex has in common with the last complex on the **mq**. This function is used to encourage the discovery of symmetric descriptions.

The cost functional is specified in the same way as the decomposition cost function above:

```
aq 4(30),-1,3,-6;
```

a aqmax

This sets the **maxstar** parameter for the Aq algorithm:

```
a aqmax 6;
```

There is one other general set of parameters which control the adjustment phase of layer 1. These control the performance of the **aqstar** procedure when it is called during the adjustment phase:

a adj

This enters the adjustment cost functional. The cost functions are the same as for the Aq cost functions above.

a adjmax

This sets the **maxstar** parameter for the adjustment process. It is entered in the same manner as **a aqmax**.

The learning element is invoked by using the **induce** command. New rules are discovered and added to the rule base. When the **induce** command is completed, it executes a **list rules** command automatically.

To list the various settings of these parameters, use the **list advice** command. Also note that there are parallel settings for the **lookback**, **plookback**, **phase**, and **models** advice parameters for use with segmented rules.

For a list of legal commands, type **h[elp]**. To exit the program, type 'q'.

APPENDIX III

Official Rules of the Game Eleusis

A. Eleusis can have 4 to 8 players.

B. procedure of playing eleusis

1. The dealer records the secret rule in unambiguous language on a sheet of paper that is put aside for future confirmation.
2. The dealer shuffles the double deck and deals 14 cards to each player.
3. The dealer places a single card called **starter** at the extreme left of the table.
4. A play consists of placing one or more cards on the table. To play a single card, the player takes a card form his hand and show it to everyone. If according to the rule the card is playable, the dealer says **Right** The card is then placed to the right of the starter card on the main line.
5. If the card fails to meet the rule, the dealer says **Wrong** In this case, the card is placed directly below the last card played, the side lines.
6. If a player displays a wrong card, the dealer gives him two more cards as a penalty.
7. A player may play a **string** of cards at once by overlapping them to preserve order and shows them to everyone, the dealer says **Right** if

all the cards conform to the rule or say **Wrong** if one or more cards are wrong. If the dealer says **Right**, the entire string is put in the main line as if the cards in the string were individually played. If the dealer says **Wrong**, the entire string is goes below the last card played and the player is dealt twice as many cards as there are in the string.

8. NO PLAY

If a player think that he knows the secret rule but finds no card that can legally played, he may declare **No play** In this case, he shows his hand to everyone.

a. If the dealer says **Right**, there are two cases.

If his hand contains fewer than 5 cards, the cards are returned to the deck and the round ends. If his hand contains more than 4 cards, his cards are put back to the deck and he is dealt a fresh hand with four fewer cards.

b. If the dealer says **Wrong**, the dealer takes one of his correct cards and puts it on the main line. The player keeps the rest of his hand and, as a penalty, is dealt five more cards.

9. After 30 cards have been played, players are expelled from the round if they make a wrong play. An expelled player is given the usual penalty cards for his final play and keeps his hand for scoring later.

10. The dealer changes after each round.

C. Scoring

1. The greatest number of card held by anyone is called **high count** Each player subtracts the number of cards in his hand form the high count. The difference is his score. If he has no cards, he gets a bonus of 4 points.
2. The dealer's score equals the highest score of any player.
3. Each player adds his scores for all the rounds played plus 10 more points if he has not been a dealer. This compensates for the fact that the dealers tend to have higher than average scores.

BIBLIOGRAPHIC DATA SHEET	1. Report No. UIUCDCS-F-85-941	2.	3. Recipient's Accession No.

4. Title and Subtitle

SPARC/E(V.2)

An Eleusis Rule Generator and Game Player

5. Report Date March, 1985

6.

7. Author(s) R. S. Michalski, Heedong Ko, Kaihu Chen

8. Performing Organization Rept. No.

9. Performing Organization Name and Address

Department of Computer Science
University of Illinois
Urbana-Champaign, Illinois

10. Project/Task/Work Unit No.

11. Contract/Grant No. NSF DCR 84-06801

12. Sponsoring Organization Name and Address

National Science Foundation
Washington, D. C.

13. Type of Report & Period Covered

14.

15. Supplementary Notes

16. Abstracts

Process prediction is a form of inductive inference where hypothesis is formed not only from the description of observations but also from the order of the observations. The program reported here, SPARC/E(V.2), is capable of forming hypothesis in the domain of a card game Eleusis. The program also provides user-friendly interface that allows the program to participate in an actual game and competes with human players.

17. Key Words and Document Analysis. 17a. Descriptors

Sequential process prediction, model driven learning, inductive learning, Eleusis.

17b. Identifiers/Open-Ended Terms

17c. COSATI Field/Group

18. Availability Statement	19. Security Class (This Report) UNCLASSIFIED	21. No. of Pages 75
	20. Security Class (This Page UNCLASSIFIED	22. Price

FORM NTIS-35 (10-70) USCOMM-DC 40329-P71