SOFTWARE TESTING, VERIFICATION AND RELIABILITY, VOL. 5, 107-132 (1995)

Decision Tables: Formalisation, Validation, and Verification

STEPHEN MURRELL
Department of Computer Science and Mathematics, University of Miami, Coral Gables, FL 33124, U.S.A.

ROBERT PLANT
Department of Computer Information Systems, University of Miami, Coral Gables, FL 33124, U.S.A.

SUMMARY

The decision table is one of the simplest representations of the rules underlying a systematic decision-making process, and is especially valuable in the development of knowledge-based systems. A two dimensional table links all relevant combinations of input conditions to the desired combinations of output actions in a very intuitive way. This simplicity belies the complex considerations involved in verifying, validating, formulating or interpreting this (or any other) representation of machine-based knowledge. In this paper, the common styles of decision table representation are reviewed, a formulation of their meaning is presented, construction methods are reviewed, and an algorithm for ensuring consistency is suggested. The problems that may occur in imperfectly constructed tables are discussed, detection methods reviewed, and some implementation methods are presented.

KEY WORDS Decision tables Validation Verification Rule-based decision making

1. INTRODUCTION

The physical representation of a decision-making process may take any one of a number of equivalent forms, including decision tables, decision trees or dependency diagrams (Metzner and Barnes, 1977; CODASYL, 1982; Subramanian *et al.*, 1992; Vanthienen and Dries, 1995).

The aim of this paper is to present a review of one of them: the 'decision table'. The use of decision tables to structure logic is not new and was a popular technique employed by systems analysts to convey programming logic to programmers in the 1970s. However, with the increasingly mainstream use of knowledge-based systems, there has been a resurgence of interest in this structuring technique by developers and researchers. This paper first presents the four established techniques that may be employed in the development of decision tables; this is then followed by a classification of error types that are related to the creation process, and a presentation of some alternative implementation methods.

CCC 0960–0833/95/020105–26

Received 6 June 1994
Revised 24 March 1995

1.1. Decision table formats

There are two generally used and functionally equivalent formats for the presentation of a decision table: the 'expositive' format, which uses a multi-dimensional layout, and the two dimensional 'classical' format, which is generally preferred for its simplicity.

1.1.1. The expositive table

The expositive table format illustrates the combinations upon which an action depends in a multi-dimensional table. The number of dimensions is equal to the number of input variables; the size of each dimension is the number of distinct values or value ranges that the corresponding variable may take on. Thus there is an entry in the table for every possible combination of inputs. The entries are simply labels indicating which action should take place.

Figure 1 illustrates the expositive style; there are three condition variables C_1, C_2, C_3; C_1 and C_3 have three possible values each while C_2 has two. As an example use of this table, if variable C_1 has value V_{12}, C_2 has value V_{22}, and C_3 has value V_{33}, action A_{223} is selected.

This form of table can become prohibitive in terms of repetitive representation of conditions as the number of conditions and value states increases.

An alternative, more compact and more easily manipulated form of representation than the expositive form is the dependency diagram or decision table. The traditional decision table form has an extensive literature associated with it; Metzner and Barnes (1977) provide an excellent bibliography, so only the more recent results in the theory of decision tables are presented here.

1.1.2. Classical decision table format

The basic form of decision tables is shown in Figure 2, in which the names of the condition variables are placed in the condition stub and potential actions that can occur from combinations of these conditions are placed in the action stub. The possible combinations of conditions can then be placed in the condition entry section of the table and

	$C_1=V_{11}$		$C_1=V_{12}$		$C_1=V_{11}$	
	$C_2=V_{21}$	$C_2=V_{22}$	$C_2=V_{21}$	$C_2=V_{22}$	$C_2=V_{21}$	$C_2=V_{22}$
$C_3=V_{31}$	A_{111}	A_{121}	A_{211}	A_{221}	A_{311}	A_{321}
$C_3=V_{32}$	A_{112}	A_{122}	A_{212}	A_{222}	A_{312}	A_{322}
$C_3=V_{33}$	A_{113}	A_{123}	A_{213}	A_{223}	A_{313}	A_{323}

Figure 1. Expositive form of decision table

Condition Stub	Condition Entries
Action Stub	Action Entries

Figure 2. Generic decision table framework

the associated actions indicated in the action entry section. This can be expanded into the form of Figure 3.

Whenever the values of the condition variables, $\langle C_1, C_2, \ldots, C_n \rangle$ match the values $\langle c_{1k}, c_{2k}, \ldots, c_{nk} \rangle$ in the condition entry for rule R_k, rule R_k may be selected, or 'fired'. When a rule is fired, the actions $\langle A_1, A_2, \ldots, A_m \rangle$ corresponding to positive values $\langle a_{1k}, a_{2k}, \ldots, a_{mk} \rangle$ in the action entry are executed. Frequently the action entry values are either 'yes' or 'no'; when a greater range of values is allowed, the actual value a_{jk} may be used as a parameter to the action A_{jk}.

1.2. A formal description

For the purposes of formal specifications, and rigorous proofs of decision-table based systems, the semantics may be formalized.

There is a collection of 'condition variables' V_1, V_2, V_3, . . ., V_k; at any given time, each variable V_i will have a value v_i drawn from an associated set (its type), T_i:

$$v_i \in T_i$$
$$T_i = \{x_{i,1}, x_{i,2}, x_{i,3}, \ldots, x_{i,k_i}\}$$

A state may be considered to be a vector in k-space consisting of 'current' values for each of the variables.

$$S = T_1 \times T_2 \times T_3 \times \ldots \times T_k$$
$$s \in S \quad \text{or} \quad s = \langle v_1, v_2, v_3, \ldots, v_k \rangle$$

In the following examples, all types are assumed to be boolean (i.e. each T_i = {Yes, No}) but could be extended to any type.

A 'condition expression' is an expression which is true for some states, and false for others. It may be viewed as a function CF from states to boolean truth values.

			1	2		k		t		K
	1	<C_1>	c_{11}	c_{12}		c_{1k}		c_{1t}		c_{1K}
I									III	
	i	<C_i>	c_{i1}	c_{i2}		c_{ik}		c_{it}		
Condition									If	
	n	<C_n>	c_{n1}	c_{n2}		c_{nk}		c_{nt}		c_{nK}
	1	<A_1>	a_{11}	a_{12}		a_{1k}		a_{1t}		a_{1K}
II									IV	
	j	<A_j>	a_{j1}	a_{j2}		a_{jk}		a_{jt}		
Action									Then	
	m	<A_m>	a_{m1}	a_{m2}		a_{mk}		a_{mt}		a_{mK}

Figure 3. The semantic fields of a standard decision table

$$\text{CF} : S \rightarrow \{\text{true}, \text{false}\}$$

In a decision table, a condition expression is represented as a vector of values; evaluation of a condition expression returns true for a state, if and only if each value in the condition vector matches the corresponding value in the state vector. The domain of condition vectors, CV, may be defined as follows:

$$\text{CV} = T^*_1 \times T^*_2 \times T^*_3 \times \ldots \times T^*_k$$

where T^*_i is a simple extension of T_i, made by adding a 'don't care' value generally denoted '-'. A condition variable may not have 'don't care' as its value, but 'don't care' may appear in condition expressions.

$$T^*_i = T_i \cup \{\text{'-'}\}$$

This allows a simple definition of an interpreting function:

$$\text{Interpret: CV} \times S \rightarrow \{\text{true}, \text{false}\}$$

$$\text{Interpret}\,(\text{cv}, s) = \text{true} \Leftrightarrow \forall i\text{: } 1 \ldots k, (\text{cv}_i = v_i \vee \text{cv}_i = \text{'-'})$$

A rule consists of a condition vector paired with a set of *actions*. For specification purposes, actions may generally be taken to be a collection of arbitrary but distinct objects:

$$A = \{a_1, a_2, a_3, \ldots, a_w\}$$

Then rules are of type R:

$$R = \text{CV} \times \mathbb{P}\,(A)$$

where $\mathbb{P}$ is the usual powerset constructor.

A decision table consists of a set of rules:

$$\text{DT} = \mathbb{P}\,(R)$$

Each rule $r = \langle \text{cv}, \text{sa} \rangle$, consisting of a condition vector paired with a set of actions, is interpreted as meaning that each action $a \in \text{sa}$ is executable in any state $s \in S$, under which the interpretation of cv is true.

$$\text{Executable: } A \times S \rightarrow \{\text{true}, \text{false}\}$$

$$(\langle \text{cv}, \text{sa} \rangle \in \text{DT}) \wedge (a \in \text{sa}) \Rightarrow (\text{Interpret}\,(\text{cv}, s) = \text{true} \Rightarrow \text{Executable}(a, s) = \text{true})$$

This may be reformulated in a function that provides the subset of actions that may be *fired* in any given state:

Actions: $S \rightarrow \mathbb{P}(A)$
Actions(s) = $\{a : A \mid \exists \langle cv, sa \rangle : DT, (a \in sa) \wedge (\text{Interpret}(cv, s) = \text{true})\}$

2. DECISION TABLE CONSTRUCTION METHODS

Four general methods have been identified for the construction of decision tables:

(i) the classical method (Metzner and Barnes, 1977);
(ii) the progressive rule development method (Dumitrascu, 1990);
(iii) Verhelst's method (Verhelst, 1975; Maes *et al.*, 1981);
(iv) the 'constructed negations' method (Wallace, 1987; Maluszynski and Naslund, 1989).

These methods are described individually below.

2.1. The classical method

The classical approach (Metzner and Barnes, 1977) is the 'common sense' approach. Essentially, a skeleton table is created and information from the knowledge source is inserted into the appropriate places. The classical method is best suited to simple applications, and may be summarized in the following four steps:

(1) list all the conditions (queries, such as 'is it a bird?', or 'what is x?') in the condition stub;
(2) list all the actions in the action stub;
(3) fill in the matrix of condition values. Often, this means that every possible combination of condition values will appear as a *situation*, but when impossible or irrelevant combinations exist, this may not be so;
(4) fill in the action values for each situation by directly interpreting the rules.

2.2. The progressive rule development method

The aim of this approach (Dumitrascu, 1990) is to develop the decision table a single rule at a time, using as few conditions as possible, as opposed to the construction of a full decision table outlined in the classical method. A decision table may be created using the progressive rule development method through the following five steps:

(1) list all the conditions in the condition stub;
(2) list all the actions in the action stub;
(3) considering the conditions *in order*, answer 'yes' to only as many conditions as are required to make a decision;
(4) record the constructed situation and its corresponding action in the table;
(5) backtrack to the last condition with a positive answer, and change that answer to 'no'; continue to answer 'yes' to as many subsequent conditions as are required to make a further decision.

Repeat steps 4 and 5 until there are no remaining positive answers to backtrack over. The following points should be noted.

(1) In a limited entry table (i.e. when conditions do not all have boolean values), an arbitrary ordering must be given to the possible values. 'Yes' corresponds to the first of those values; changing an answer to 'no' corresponds to changing that answer to the next value in sequence.
(2) As the order in which conditions appear in the condition stub has a significant effect on the construction of the table, that order should be chosen with as much care as is possible — the most significant conditions should appear first.

This method attempts to overcome the problems associated with the scale of creating a full decision table by the classical method. However, this approach relies upon the developer's ability to understand the dependencies as the table develops, and hence is open to omissions or inconsistencies created by humans.

2.3. The Verhelst method

In the classical method and the progressive method, the developer works through the conditions to the actions. Verhelst (1975) and Maes *et al.* (1981) have proposed the following form for preprocessing of the knowledge-base that will simplify the tables.

(1) Identify the conditions and actions, but do not enter them into their stubs.
(2) Whenever more than one condition depends upon the same 'input variable', combine those conditions into a single condition with a more complex domain of answers (e.g. the three boolean conditions '$X<7$?', '$7\leq X<12$?', '$12\leq X$?' are converted into a single condition 'X?' with possible values '<7', '≥ 7 & <12?', '≥ 12').
(3) Reformulate the original knowledge-base in terms of those new conditions.
(4) Construct the table by any appropriate method.

2.4. Constructed negations method

In many basic systems, only positive deductions are possible, for example, in the table of Figure 4, each 'x' denotes a conclusion being deduced. Conclusions are generally considered false simply when they are not deduced to be true. This paradigm, known as 'negation as failure', can be too inflexible, making certain conditions impractical to specify.

	1	2	3	4	5
c1:	N	N	N	N	Y
c2:	N	N	Y	Y	N
c3:	N	Y	N	Y	Y
a1:	X	X		X	X
a2:			X		X
a3:			X		

Figure 4. Positive deductions only

	1	2	3	4	5	6
c1: A car	-	-	-	-	-	-
c2: A table	-	-	-	-	-	-
c3: A cat	-	-	Y	-	-	-
c4: A dog	-	-	-	Y	-	-
c5: A horse	-	-	-	-	Y	-
c6: A snake	-	Y	-	-	-	-
c7: A spider	-	-	-	-	-	Y
...	-	-	.	.	.	.
a1: Arthur likes	X	-	-	-	-	-
a1: Boris likes	-	X	-	-	-	-
a1: Cabunzel likes	-	-	X	X	X	X

Figure 5. The inflexibility of positive deductions

Consider for example:

(i) Arthur likes everything;
(ii) Boris likes snakes;
(iii) Cabunzel likes all animals except snakes.

(i) and (ii) are easy to express in a decision table — the deduction 'Arthur likes it' is always made — it has no conditions, as shown in Figure 5.
(iii) may only be encoded if there is a condition for every possible kind of animal, i.e. the computation is only valid if:

$$(\text{it is an animal}) \Rightarrow c3 \vee c4 \vee c5 \vee c6 \vee \ldots$$

which may be very difficult to verify. Naturally, if it is known that *only* animals are under consideration, the situation is simplified, as in Figure 6, but this does require that 'it is a snake' is known to be false, not just undeduced for a1 to be activated; therefore c6 would have to be an input, not a deduced fact under the negation as failure model.

If negated deductions are allowed (Maes and Van Dijk, 1988), the system becomes much more flexible (see Figure 7); here 'Cabunzel likes it' is actively deduced to be false, whenever 'it' turns out to be a snake. This is not the same as the previous table, which requires positive knowledge that 'it' is *not* a snake.

However, negative conclusions can lead to errors that would not be possible otherwise, and are difficult to detect.

It is possible that a situation such as that illustrated in Figure 8 could (accidentally)

	1
.	-
.	-
c6: A snake	N
.	-
.	.
a1: Cabunzel likes	X

Figure 6. Using animals only

	1	2	3
c1: A car	-	-	-
c2: A table	-	-	-
c3: A cat	-	-	-
c4: A dog	-	-	-
c5: A horse	-	-	-
c6: A snake	-	Y	Y
c7: A spider	-	-	-
...	.	.	.
a1: Arthur likes	CY	-	-
a2: Boris likes	-	CY	-
a3: Cabunzel likes	-	-	CN

Figure 7. Negative deductions

	1	2	3	4
c1:	-	Y	-	Y
c2:	-	Y	Y	-
c3:	Y	-	-	Y
c4:	Y	-	-	Y
a1:	CY	CN	CN	-
a2:	-	CY	-	-
a3:	-	-	CY	CN

Figure 8. Inconsistent table

arise. If c2, c3 and c4 are all true, rules 1 and 3 may fire, and a1 will be deduced to be both true and false.

An alternative method, which ensures that conclusions that could never be deduced true must be deduced false, without allowing for human-error-induced inconsistencies, is known as 'constructive negation' (Wallace, 1987; Maluszynski and Naslund, 1989; Plaza, 1992). Here, the user specifies a table with only positive deductions, such as the example of Figure 9. From this, negative deductions are constructed as follows.

(1) One implication to represent each conclusion is formed directly from the decision table:

$$(c1 \wedge \neg c2) \vee (\neg c1) \Rightarrow a1 \quad (c1 \wedge c2 \wedge c3) \Rightarrow a2$$

	1	2	3
c1:	Y	N	Y
c2:	N	-	Y
c3:	-	-	Y
a1:	X	X	
a2:			X

Figure 9. User's positive specification

and the implications for each conclusion are independently processed by the remaining steps.

(2) The 'completion assumption' is made, i.e. assume that the table contains *all* the information, so each implication becomes an equivalence:

$$(c1 \wedge \neg c2) \vee (\neg c1) \Leftrightarrow a1$$

(3) Invert both sides of each equivalence:

$$\neg [(c1 \wedge \neg c2) \vee (\neg c1)] \Leftrightarrow \neg a1$$

(4) Use DeMorgan's laws to move all negations to positions immediately before variable names:

$$[\neg (c1 \wedge \neg c2)] \wedge [\neg (\neg c1)] \Leftrightarrow \neg a1$$
$$((\neg c1) \vee (\neg \neg c2)) \wedge (\neg \neg c1) \Leftrightarrow \neg a1$$

(5) Simplify if necessary, removing double negatives, tautologies and contradictions:

$$((\neg c1) \wedge (\neg \neg c1)) \vee ((\neg \neg c2) \wedge (\neg \neg c1)) \Leftrightarrow \neg a1$$
$$(\neg c1 \wedge c1) \vee (c2 \wedge c1) \Leftrightarrow \neg a1$$
$$(c1 \wedge c2) \Leftrightarrow \neg a1$$

(6) Reduce to an implication rule for the negative conclusion:

$$(c1 \wedge c2) \Rightarrow \neg a1$$

The same steps applied to conclusion a2 give:

$$(c1 \wedge c2 \wedge c3) \Leftrightarrow a2$$
$$\neg (c1 \wedge c2 \wedge c3) \Leftrightarrow \neg a2$$
$$(\neg c1) \vee (\neg c2) \vee (\neg c3) \Leftrightarrow \neg a2$$
$$(\neg c1) \vee (\neg c2) \vee (\neg c3) \Rightarrow \neg a2$$

(7) The new rules are finally added to the table. The results are shown in Figure 10.

	a	b	c	d	e	f	g
c1:	Y	N	Y	Y	N	-	-
c2:	N	-	Y	Y	-	N	-
c3:	-	-	Y	-	-	-	N
a1:	CY	CY		CN			
a2:			CY		CN	CN	CN

ORIGINAL NEW

Figure 10. The completed table

The result is a complete table automatically produced, which is guaranteed to deduce every conclusion to be *either* true *or* false, without possibility of inconsistencies. This does not produce the most compact table possible; a more compact version could be generated using methods discussed in Section 3.1.5.

Although *inconsistencies* are not possible, it may be observed that *conflicts* often do occur (for example, see columns 3 and 4 in Figure 10, above). Such conflicts may only exist between rules leading to compatible conclusions, so under some implementations there is no problem, but under others compaction is required.

If the user wishes to specify some negative deductions, these may be induced in the completion of step 2; if inconsistencies are to be avoided, some prioritization is necessary. It is most usually appropriate to consider negative deductions to be special cases that over-ride the general cases of the positive deductions.

For example (see Figure 11), Denzel likes all animals and plants except snakes and daffodils.

First, the rules leading to positive and negative deductions are combined into separate disjunctions:

$$c1 \vee c2 \Rightarrow a1$$
$$c3 \vee c4 \Rightarrow \neg a1$$

then the negative disjunction is used to over-ride the positive, giving an equivalence for the positive conclusion, and another for the negative:

$$(c1 \vee c2) \wedge \neg (c3 \vee c4) \Leftrightarrow a1$$
$$\neg [(c1 \vee c2) \wedge \neg (c3 \vee c4)] \Leftrightarrow \neg a1$$

These are both reduced to the appropriate form by the usual methods:

$$(c1 \vee c2) \wedge \neg (c3 \vee c4) \Leftrightarrow a1 \qquad \neg [(c1 \vee c2) \wedge \neg (c3 \vee c4)] \Leftrightarrow \neg a1$$

$$(c1 \vee c2) \wedge (\neg c3 \wedge \neg c4) \Leftrightarrow a1 \qquad \neg (c1 \vee c2) \vee \neg \neg (c3 \vee c4) \Leftrightarrow \neg a1$$

$$(c1 \wedge \neg c3 \wedge \neg c4) \vee (c2 \wedge \neg c3 \wedge \neg c4) \Leftrightarrow a1 \qquad (\neg c1 \wedge \neg c2) \vee c3 \vee c4 \Leftrightarrow \neg a1$$

	1	2	3	4
c1: animal	Y	-	-	-
c2: plant	-	Y	-	-
c3: snake	-	-	Y	-
c4: daffodil	-	-	-	Y
a1: Denzel likes	CY	CY	CN	CN

Figure 11. Positive and negative deductions

	1	2	3	4	5
c1:	Y	-	N	-	-
c2:	-	Y	N	-	-
c3:	N	N	-	Y	-
c4:	N	N	-	-	Y
a1:	CY	CY	CN	CN	CN

Figure 12. Consistent table

giving a totally new decision table (Figure 12), which is, again, *guaranteed* to be complete and free of inconsistencies *and* to encode the closest possible consistent meaning to the user's original specification. However, these processes can, in exceptional cases, take time exponential in the number of condition variables.

2.5. Single hit and complete tables

Most decision tables in practical use are 'multiple hit', meaning that for some values of the input variables more than one rule will be fireable. This possibility is the root cause of many of the errors discussed later (Section 3). Techniques for guaranteeing the mutual independence of the columns of the table (Vanthienen and Dries, 1995) may be applied; these may either involve a modified knowledge acquisition technique, which prevents the inclusion of non-independent rules, or an *a posteriori* restructuring of the table.

A single hit table is guaranteed not to require conflict resolution in the inference engine, but will represent knowledge in a way that the original expert may consider unnatural or even incorrect (the representation cannot actually become incorrect, but human experts often do not appreciate formal logic, or deviations from their own original structuring), and may therefore lead to later verification problems.

Furthermore, most construction methods do not guarantee that the table will be complete, and completeness prevents many of the types of error discussed in later sections from occurring. One way to guarantee that a table is both complete and 'single hit' is to ensure that every possible combination of condition values appears. An example of such a complete decision table is given in Figure 13.

Clearly, such a table may grow to excessive size, and further processing may be necessary to recognize compatible rules and compress them (e.g. 'C1 $\wedge$ C2 $\wedge$ C3 $\Rightarrow$ A1', and 'C1 $\wedge$ C2 $\wedge$ $\neg$ C3 $\Rightarrow$ A1' may be combined into 'C1 $\wedge$ C2 $\Rightarrow$ A1', without compromising completeness).

C1	Y	Y	Y	Y	N	N	N	N
C2	Y	Y	N	N	Y	Y	N	N
C3	Y	N	Y	N	Y	N	Y	N
A1	X	X						
A2			X	X		X		X
A3					X		X	

Figure 13. Example decision table

3. ERRORS IN DECISION TABLES

The creation and use of a decision table to represent a decision making process, is often a major aspect of formalizing a domain for a knowledge-based system (Santos-Gomez and Darnell, 1992; Vanthienen and Dries, 1993; Vanthienen and Robben 1993; Vanthienen and Wets, 1993). This process is very sensitive to errors in the table. These may be logical, epistemological, or semantic, and generally fall into the following four major categories, for which informal definitions are given below.

- *Redundancy*. A rule that adds no contribution, which may be further subdivided into:
  - *Identity*. Two rules completely identical (Section 3.1.1)
  - *Subsumption*. One rule being a generalization of another (Section 3.1.2)
  - *Indirect*. Two deductive paths lead to same result (Section 3.1.3)
  - *Unfireable rules*. A rule's condition can never be satisfied (Section 3.1.4)
  - *Reducible*. Two rules may be combined into one (Section 3.1.5)
- *Conflicting rules*. Simultaneously fireable rules with inconsistent results (Section 3.2)
- *Circularity*. A circular chain of rules (Section 3.3)
- *Errors of omission*. Commonly errors that originate from the knowledge base:
  - *Unused inputs and outputs* (Section 3.4.1)
  - *Missing rules*. The set of rules does not cover all possible inputs (Section 3.4.2)
  - *Impossible combinations*. Input conditions that cannot coexist (Section 3.4.3)
  - *Dead end rules*. Rules that do not lead to any conclusions (Section 3.4.4)

A significant research effort has been directed towards the detection of errors in decision tables (Cragun and Steudel 1987; Puuronen, 1987; Merlevede and Vanthienen, 1991; Tanaka *et al.*, 1993). These various types of error that may occur in a decision table, and the techniques for identifying and possibly correcting them, are discussed below.

3.1. Redundant rules

A common type of error for which a verification check may be applied is redundancy. A redundant rule is simply one which adds no contribution to the system. Redundancy may be decomposed into five subcategories: 'identity', 'subsumption', 'indirect redundancy', 'unfireability' and 'reducibility'.

3.1.1. Identity

The first type of redundancy to be considered is that of identical rules, which can be broken down into two subcategories: 'syntactic' and 'semantic' redundancy.

The case of 'syntactic redundancy' is illustrated in Figure 14, which shows that syntacti-

	... 26 ... 93 ...
c1: A	... Y ... Y ...
c2: B	... Y ... Y ...
a1: C	... X ... X ...

Figure 14. Syntactic redundancy

```
! Identity Error: Syntactic Redundancy
! Actions Block

ACTIONS                                  ! Required block heading
    DISPLAY "Identity!!"                 ! Opening message
    FIND Advice                          ! Goal variable designation
    DISPLAY "The best advice we have for you is: {#Advice}.";
                                         ! Display of goal variable
! Rules Block
RULE 26                                  ! Mandatory rule label
IF      A = Yes                          ! Condition
        AND B = Yes                      ! Condition
THEN    Advice = C;                      ! Rule conclusion

RULE 93                                  ! Mandatory rule label
IF      B = Yes                          ! Condition
        AND A = Yes                      ! Condition
THEN    Advice = C;                      ! Rule conclusion

! Statements Block
ASK A: "What is the value of A?";        ! Generates question
CHOICES A: Yes, No;                      ! Possible answers for A

ASK B: "What is the value of B?";        ! Generates question
CHOICES B: Yes, No;                      ! Possible answers for B
```

Figure 15. VP-Expert program with syntactic redundancy

cally redundant rules can be identified as identical columns in a decision table. This error type is illustrated in Figure 15, as a program in VP-Expert (Hicks and Lee, 1988), a production system shell. This shell does not have a verification or error system associated with it other than the compiler's syntax checker. The program in Figure 15 will compile without error.

In order to prevent the occurrence of syntactically identical rules it is necessary to examine the decision tables for identical columns. Following the convention of Figure 16, the following symbols may be defined: C is the total number of columns; Q is the total number of conditions; A is the total number of actions; and R is the total number of rows.

As the number of rules is generally much greater than the number of actions or condition variables, a $O(\ldots)$ formulation involving C and any of R, Q or A, will be dominated by the C component. Thus R, Q and A may usually be considered constants, and ignored, e.g. $O(RC^2)$ is effectively $O(C^2)$.

Identical columns may be detected by a simple [time = $O(RC^2)$] search of all pairs. However, if the columns are first sorted [time = $O(RC \log_2 C)$] so that identical columns become neighbours, a simpler [time = $O(RC)$] search will find all identical pairs.

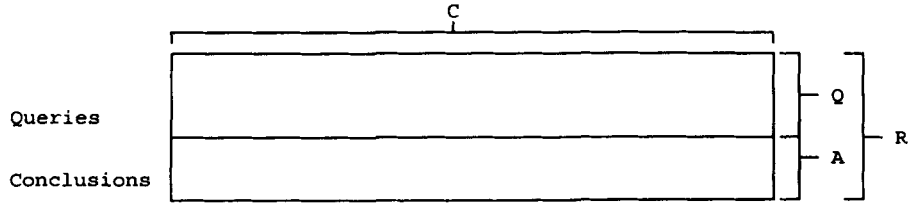

Figure 16. Decision table format

	45	83
c1: A	Y	Y
c2: B	Y	Y
a1: C	X	
a2: C		X

Figure 17. Semantic redundancy

The case of 'semantic redundancy' occurs when two or more rules have conditions and conclusions which are semantically equivalent but represented through different symbolic notation. An illustration of this situation is provided by the decision table of Figure 17, and a corresponding program in Figure 18.

A similar but even worse problem occurs when conditions are semantically but not syntactically identical, e.g.:

RULE 63: IF hot AND humid THEN thunderstorms
RULE 99: IF sultry THEN electricalstorms

Here, although conditions and conclusions are both equivalent, there is no clue in their form, and no mechanical system could possibly detect any problem. The problem of semantic redundancy is illustrated in Figure 18, which shows a VP-Expert system that utilizes different representations of a value (1 Ton) in the same program. The reader of such a system may not know whether the writer wished for this difference or not, making validation extremely difficult.

The case of identifying semantically redundant rules is extremely difficult, as they are impossible to detect without an expert whose expertise exceeds that of the expert from whom the information was elicited. The only techniques available are to identify rules with identical conditionals and notify the developer of their existence as a warning that

```
! Identity Error: Semantic Redundancy
! Actions Block

ACTIONS                                  ! Required block heading
      DISPLAY "Identity!!"               ! Opening message
      FIND Advice                        !  Goal variable designation
      DISPLAY "The best advice we have for you is: {#Advice}.";
                                         ! Display of goal variable
 ! Rules Block
 RULE 1                                  ! Mandatory rule label
 IF       A = Yes                        ! Condition
          AND B = Yes                    ! Condition
 THEN     Advice = Weight_GT_2240;       ! Rule conclusion

 RULE 2                                  ! Mandatory rule label
 IF       B = Yes                        ! Condition
          AND A = Yes                    ! Condition
 THEN     Advice = Weight_GT_1_Ton;      ! Rule conclusion

 ! Statements Block
 ASK A: "What is the value of A?";       ! Generates question
 CHOICES A: Yes, No;                     ! Possible answers for A

 ASK B: "What is the value of B?";       !Generates question
 CHOICES B: Yes, No;                     ! Possible answers for B
```

Figure 18. A VP-Expert program with semantic redundancy

there may be a redundancy and provide the facility for manual authorization and checking of the validity of these rules.

3.1.2. *Subsumed rules*

The case of rule subsumption occurs when two rules have identical conclusions while the conditions for one are either a generalization or special-case of the conditions for the other. The decision table shown in Figure 19 represents an example of this situation.

If the developer should choose to detect rule subsumption, a mechanical test may be performed before the creation of the graphs at the decision table stage. This is done by identifying all pairs of rules that reach the same conclusion, then checking that none has a condition that is a generalization of any other condition.

In Figure 19, rules R1, R2 and R3 all lead to conclusion c2, and thus the following six tests are performed:

$$R1 \subset R2$$
$$R2 \subset R1$$
$$R3 \subset R1$$
$$R1 \subset R3$$
$$R2 \subset R3$$
$$R3 \subset R2$$

If any of the tests are true then there is a subsumed rule. The $\subset$ test takes time $O(Q)$ and there are $O(CS)$ tests. (If S is the number of rules leading to the same conclusion, there are S^2 tests for each conclusion, multiplied by $C \div S$ conclusions.) The total test time is therefore $O(QCS)$, or $O(QC^2)$ as an absolute worst case, in which there is only one conclusion in the whole table. No pre-sorting can reduce this time.

The program illustrated in Figure 20 shows an example of the problem of rule subsumption, in a VP-Expert program. Again this error type is not detected at compilation.

3.1.3. *Indirect redundancy*

Indirect redundancy occurs when there are two deductive paths from a set of input conditions to the same conclusions. It can only be reliably detected by a brute force search over all possible sets of inputs (alternative routes to the same conclusion may be of any length). Attempting to work backwards from the conclusions to their causes is no simpler. Clearly such a search would require exponential time and therefore is not a practical possibility for any real system. An example of indirect redundancy in a VP-Expert program is illustrated in Figure 21, which is based upon three rules:

	R0	R1	R2	R3	R4
q1: A	Y	Y	Y	Y	N
q2: B	N	Y	Y	-	Y
q3: C	Y	N	Y	N	-
	Y	Y	N	Y	-
c1	X	-	-	-	-
c2	-	X	X	X	-
c3	-	-	-	-	X

Figure 19. Decision table with subsumed rules

```
! Identity Error: Subsumed Rules
! Actions Block

ACTIONS                                  ! Required block heading
      DISPLAY "Subsumption!!"            ! Opening message
      FIND Advice                        ! Goal variable designation
      DISPLAY "The best advice we have for you is: {#Advice}.";
                                         ! Display of goal variable
 ! Rules Block
 RULE 1                                  ! Mandatory rule label
 IF       A = Yes                        ! Condition
          AND B = Yes                    ! Condition
          AND C = Yes                    ! Condition
 THEN     Advice = Z;                    ! Rule conclusion

 RULE 2                                  ! Mandatory rule label
 IF       A = Yes                        ! Condition
          AND B = Yes                    ! Condition
 THEN     Advice = Z;                    ! Rule conclusion

 ! Statements Block
 ASK A: "What is the value of A?";       ! Generates question
 CHOICES A: Yes, No;                     ! Possible answers for A

 ASK B: "What is the value of B?";       ! Generates question
 CHOICES B: Yes, No;                     ! Possible answers for B

 ASK C: "What is the value of C?";       ! Generates question
 CHOICES C: Yes, No;                     ! Possible answers for C
```

Figure 20. VP-Expert program with subsumption

```
! Identity Error: Direct Redundancy
! Actions Block

ACTIONS                                  ! Required block heading
      DISPLAY "Direct      !!"           ! Opening message
      FIND Advice                        ! Goal variable designation
      DISPLAY "The best advice we have for you is: {#Advice}.";
                                         ! Display of goal variable
 ! Rules Block
 RULE 1                                  ! Mandatory rule label
 IF       p = Yes                        ! Condition
 THEN     Advice = q;                    ! Rule conclusion

 RULE 2                                  ! Mandatory rule label
 IF       q = Yes                        ! Condition
 THEN     Advice = r;                    ! Rule conclusion

 RULE 3                                  ! Mandatory rule label
 IF       p = Yes                        ! Condition
 THEN     Advice = r;                    ! Rule conclusion

 ! Statements Block
 ASK p: "What is the value of p?";       ! Generates question
 CHOICES p: Yes, No;                     ! Possible answers for p

 ASK q: "What is the value of q?";       ! Generates question
 CHOICES q: Yes, No;                     ! Possible answers for q
```

Figure 21. VP-Expert program with indirect redundancy

Rule 1: IF p THEN q
Rule 2: IF q THEN r
Rule 3: IF p THEN r

As indirect redundancy depends upon the existence of deductive paths, it must be possible to link rules by having a variable whose value is deduced as an action of one rule appearing as a condition of another. This is illustrated by variable *q* in the above rules and again in Figure 22.

It should be pointed out that indirect redundancy is not necessarily a fault in the system. It is often possible to deduce something more than one way. However, it is a situation of which the developer may wish to be aware.

3.1.4. Unfireable rules

This subcategory related to redundancy covers those rules in knowledge-bases that can never be fired. It only really stems from semantic properties:

RULE 16: IF vital AND unimportant THEN action

which like all the other semantic redundancies is not mechanically detectable. A form of syntactic unfireability can occur, as illustrated by the following three rules, and the decision table derived from them (Figure 22).

Rule 1: IF p THEN q
Rule 2: IF ¬ p THEN r
Rule 3: IF q AND r THEN s

This can be expressed in an extended decision table and would again require an exponential time search to detect. This situation is more likely to result from incomplete domain knowledge than an error in the decision table. An example of unfireable rules, which like all the other semantic redundancies is not mechanically detectable, in a VP-Expert program may be found in Figure 23.

3.1.5. Reducible rules

Reducible rules occur when two or more rules contain conditions that conflict and can be combined into one by removing the conflicting variable. It is possible to identify these unnecessary rules and improve the efficiency through identification of conflicts in the decision table. In the example of Figure 24 it can be seen that rules 'a' and 'b' can be merged into one, as shown below in Figure 25 (it would also have been possible to combine rules 'b' and 'c', but not all three). The reduction process requires a search of all possible pairs of rules which lead to the same conclusions and a comparison of their

	R1	R2	R3
p	Y	N	-
q	CY	-	Y
r	-	CY	Y
s	-	-	CY

Figure 22. Decision table with subsumed rules

```
! Identity Error: Unfireable rules
! Actions Block

ACTIONS                                  ! Required block heading
      DISPLAY "Unfirable Rule!!"         ! Opening message
      FIND Advice                        ! Goal variable designation
      DISPLAY "The best advice we have for you is: {#Advice}.";
                                         ! Display of goal variable
! Rules Block
RULE 1                                   ! Mandatory rule label
IF      p = Yes                          ! Condition
AND     p = No                           ! Condition
THEN    Advice = r;                      ! Rule conclusion

! Statements Block
ASK p: "What is the value of p?";        ! Generates question
CHOICES p: Yes, No;                      ! Possible answers for p
```

Figure 23. VP-Expert program with an unfireable rule

	a	b	c
c1:	Y	Y	Y
c2:	Y	Y	N
c3:	N	Y	Y
c4:	Y	Y	Y
c5:	N	N	N
a1:	-	-	-
a2:	X	X	X
a3:	-	-	-

Figure 24. Reduction of rules

	ab	c
c1:	Y	Y
c2:	Y	N
c3:	-	Y
c4:	Y	Y
c5:	N	N
a1:	-	-
a2:	X	X
a3:	-	-

Figure 25. Reduced rules

conditions to see if they are identical except for one variable. This would take time O(QCS).

3.2. Conflicting rules

Rules are in conflict when one allows a particular conclusion to be deduced, another allows the inverse of that conclusion to be deduced, and both are able to fire. For example, Figure 26 illustrates a syntactically detectable form.

This has a similar form to the first two cases of redundancy (identity and subsumption),

	9	12
c1: p	Y	Y
a1: z	CY	CN

Figure 26. Conflicting rules

	9	10
c1: A	Y	
c2: B		Y
a1: B	X	
a2: A		X

Figure 27. Circular rules

and may be detected by identical means. There is also an indirect form of conflict in which conflicting conclusions are reached only after a chain of deductions. This must be detected in the same way as indirect redundancy. There is also a semantic counterpart as illustrated by the following two rules, which is practically undetectable:

RULE 3: IF very_cold THEN nice_day
RULE 40: IF frigid THEN unpleasant_day

3.3. Circular rules

Circularity in rules is self-explanatory and is illustrated by the following example:

RULE 9: IF A THEN B
RULE 10: IF B THEN A

which could be represented by the tables of Figure 27 (with standard notation) or Figure 28 (using the extended notation).

It should be noted that circularity does not necessarily signify an error, as illustrated in Figure 29, but it is normally considered undesirable as many implementations are unable to deal with it.

3.4. Errors of omission

This category covers errors that usually indicate some deficiency in the original knowledge base rather than the more operational errors covered by the other categories. Such

	9	10
A	Y	CY
B	CY	Y

Figure 28. Circular rules

```
AUTOQUERY;
! Identity Error: Circular Rules
! Actions Block

ACTIONS                                  ! Required block heading
     DISPLAY "Circular Rules!!"          ! Opening message
     FIND Oldadvice                      !  Goal variable designation
     DISPLAY "The best advice we have for you is: {#Advice}.";
                                         ! Display of goal variable
 ! Rules Block
 RULE 1                                  ! Mandatory rule label
 IF      quadruped = Y                   ! Condition
 THEN    legs = 4;                       ! Rule conclusion

 RULE 2                                  ! Mandatory rule label
 IF      legs = 4                        ! Condition
 THEN    quadruped = Y;                  ! Rule conclusion
```

Figure 29. VP-Expert program with circular rules

c1: A c2: B c3: C	Y Y - N N N N - - - - - - - Y N Y N - Y N
a1: I a2: J a3: K	X X X X X X X X

Figure 30. Missing inputs and outputs

a deficiency will never cause a rule-based system to behave inconsistently, but may prevent a solution from being found. These errors correspond to missing information.

3.4.1. Unused inputs and outputs

An unused input or output is the simplest error to detect, and also the least likely to occur. It is simply a case of one or more inputs never being used in any condition or one or more outputs never being concluded by any rule. These are detectable in O(QC) time by a simple search. For example, in Figure 30, 'B' is a missing input and 'K' is a missing output.

3.4.2. Missing rules or uncovered inputs

Missing rules are those potential rules which correspond to possible combinations of input conditions that do not appear in the decision table (for example, see Figure 31 in

	a	b	c	d	e	f	g
q1:	N	N	N	N	Y	Y	Y
q2:	N	N	Y	Y	N	Y	Y
q3:	N	Y	N	Y	Y	N	Y
c1:	X	X			X		X
c2:			X		X		
c3:			X			X	

Figure 31. Missing rules example

which the condition q1 $\wedge \neg$ q2 $\wedge \neg$ q3 would correspond to a missing rule). The only reliable way to check that all combinations of conditions are covered is to check every possible combination in turn. Even if all conditions are binary (Yes or No) this would take time $O(QC2^Q)$. Again it should be noted that this is not necessarily an error, because some combinations of inputs may be impossible.

3.4.3 Impossible combinations

If certain combinations of conditions (such as 'It is very sunny', and 'It is gloomy and wet') cannot possibly occur together, those combinations may fail to appear in the condition entries of a decision table, without constituting a missing rule. As impossible combinations can often only be detected with deep semantic knowledge (e.g. knowing that 'gloomy' precludes 'sunny'), their existence is not in general automatically detectable, and impossible combinations can result in the 'missing rules' error being erroneously signalled.

3.4.4. Dead end rules

Dead end rules (those that lead to no conclusions) are exceptionally easy to detect; the action entry for a column in the decision table will be completely empty. This is illustrated by rule d in Figure 31. It should be noted that dead end rules may result from an insistence that decision tables be complete when not all combinations of inputs lead to useful conclusions.

3.5. Condition counting systems

A mechanism to assist in the validation of decision tables, known as the incidence matrix method, has been cited in the literature (Agarwal and Tanniru, 1991). Its basis is the use of matrix operations to determine if the rules derived from a decision table contain certain error types (subsumption or redundancy). The literature indicates that a complex sequence of matrix manipulation steps must be performed in order to determine if these error conditions are present. However, these manipulations are unnecessary as the steps can be reduced to a set of fundamental principles based on counting conditions.

Define T_i as the total number of condition variables that are not 'don't care' or '-' in rule i, and C_{ij} as the number of condition variables that have the same value (excluding 'don't care') in both rule i and rule j. Figure 32 shows an example.

The basic idea is that if every condition value in one rule (i) also appears in another

	1	2	3	4
q1:	Y	Y	Y	Y
q2:	N	-	-	-
q3:	Y	Y	Y	Y
q4:	N	N	Y	Y
c1:	X	X		
c2:			X	X

$T_1 = 4$ $T_2 = 3$ $T_3 = 3$ $T_4 = 3$

$C_{12} = 3$ $C_{13} = 2$ $C_{14} = 2$ $C_{23} = 2$ $C_{24} = 2$ $C_{34} = 3$

$C_{21} = 3$ $C_{31} = 2$ $C_{41} = 2$ $C_{32} = 2$ $C_{42} = 2$ $C_{43} = 3$

Figure 32. Missing rules example

(j) and that other (j) involves more condition variables than the first (i), i.e. $T_i = C_{ij} \wedge T_i < T_j$, then rule i is subsumed by rule j.

If the number of conditions that two rules have in common is exactly the same as the number of conditions that each of those rules has, i.e. $C_{ij} = T_i = T_j$, then those two rules are identical.

The technique of condition counting may be slightly faster than the methods listed above, but it does still have the same time complexity, and is capable of detecting fewer error types.

4. IMPLEMENTATION TECHNIQUES

In this section of the paper, several alternative techniques for the creation of knowledge-based systems from decision tables are considered.

The most natural implementation technique is direct interpretation of the decision table. The repeated execution cycle is to search through the columns of the table until one is found for which all necessary conditions are true, when the associated actions are taken, updating the values of the state variables accordingly. Execution terminates when no columns are selectable.

It is hard to argue with the correctness or simplicity of this technique, but it does have certain drawbacks: there is no means for preventing infinite looping if circular rules exist, other than deleting a rule after it fires; encoding any auxiliary actions (beyond simply deducing values for variables) is difficult; in a very large table, when the search time for a fireable rule may become prohibitive, there is no readily available means of parallelization.

For these, and other reasons, many different implementations are available. Translation into a production system shell such as VP-Expert or the more general purpose programming language LISP (McCarthy, 1962) is a popular choice as each column can be directly translated into a rule. This has been illustrated frequently in previous examples.

Another approach is to translate the table into the language of some parallel processing environment, such as the ALICE machine (Darlington and Reeve, 1981), a graph-reduction computer.

4.1. A graph-reduction implementation

In its simplest form, the decision table is converted into a tree-like structure with the condition variables and their negation as leaves, nodes representing AND operations linking them to form the condition parts of rules, and nodes representing OR operations linking the AND-trees for rules that lead to the same conclusion. For example, the decision table shown in Figure 33 is converted into the graph (logically a tree with structure sharing) of Figure 34.

	1	2	3
q1:	Y	Y	N
q2:	N	Y	N
q3:	Y	-	N
c1:	X	X	
c2:		X	X

Figure 33. Missing rules example

An ALICE machine can directly execute this graph, with its multiple processors each selecting an appropriate node to execute. This can lead to an enormous speed-up, with the time required to deduce a particular conclusion totally independent of the number of nodes. The technique has been covered fully by Murrell and Plant (1995).

4.2. Prolog implementation

Prolog (Clocksin and Mellish, 1981) is another popular implementation language. The rules from the decision table are converted directly into Prolog as illustrated by the transformation from the table of Figure 35 to the following program:

```
q1 :- write('is it grey?'), read(Y), Y = 'yes'.
q2 :- write('is it small?'), read(Y), Y = 'yes'.
q3 :- write('does it squeak?'), read(Y), Y = 'yes'.
c1 :- q1, q2.
c1 :- q1, q3.
c2 :- q2, q3.
deduce :- c1, write('it could be a mouse.').
deduce :- c2, write('it could be a hinge.').
```

which is directly executable in Prolog.

Prolog is a very powerful artificial intelligence language; it has much more power and flexibility than is needed for implementing basic decision tables, but suffers from one major drawback — in current forms, it is almost completely incapable of supporting negation, so is suitable *only* for these basic decision tables.

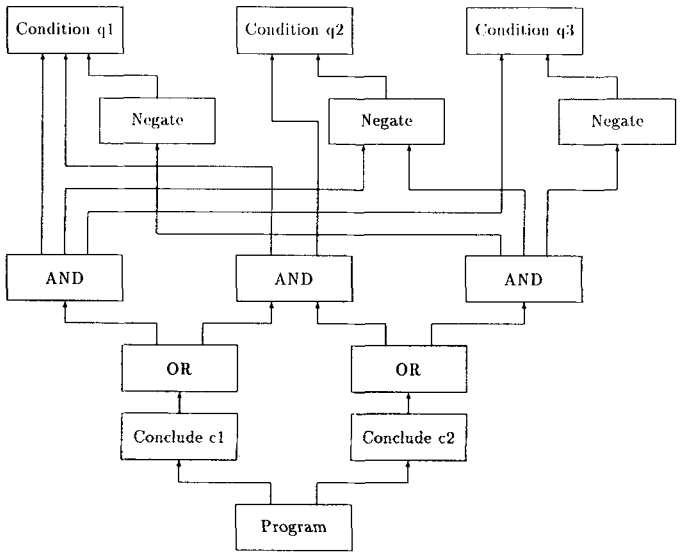

Figure 34. A graph reduction model

	1	2	3
q1: Grey?	Y	Y	-
q2: Small?	Y	-	Y
q3: Squeaks?	-	Y	Y
c1: Mouse.	X	X	-
c2: Hinge.	-	-	X

Figure 35. Rules for conversion into Prolog

4.3. Tools for error detection

The creation of knowledge-based systems through the use of decision tables is, as has been seen, open to several error causing situations. This is especially true if the tables are large and the translation process into code is performed in a manual fashion. However, as demonstrated, several techniques do exist for the structuring of the development process as well as techniques for error detection and prevention. This research has been adapted by the validation and verification research community, whose focus is upon software tools. This research has led to the construction of several interesting and useful tool sets, as follows:

— Rule Checking Program (RCP) (Suwa *et al.*, 1984)
— Expert System Checker (ESC) (Cragun and Steudel, 1987)
— CHECK (Nguyen *et al.*, 1985)
— Art Rule Checker (ARC) (Nguyen, 1987)
— KB-Reducer System (Ginsberg, 1987)
— EVA (Stachowitz *et al.*, 1987)
— COVER (Preece and Shinghal, 1991)
— Rule-based Intelligent Test Case Generator (Gupta, 1990)
— Datamap (Coenen, 1991)
— Rulenut (Yoon, 1989)
— MAUDE (Bench-Capon and Coenen, 1991)

A survey of these tools is beyond the scope of this paper; however, a good overview of them can be found in the work of Coenen and Bench-Capon (1993).

5. COMMENTS AND CONCLUSIONS

Having examined the theory and practice of decision tables as applied to systems development, this well established form has been found to have many advantages. The primary advantage is its intuitive style, which makes construction and use a simple task. Decision tables lend themselves readily to direct implementation in a number of styles, including traditional programming languages, production system shells, and standard logic programming approaches, as well as mapping conveniently onto a very efficient parallel platform.

Although decision tables are subject to a large number of common faults, most of these are efficiently detectable by standard techniques, and may cease to be problems under certain implementations. Traditionally, the support of negative conditions and conclusions has caused many problems, but the new technique of constructed negation provides an

extension that alleviates them. In conjunction with advanced software tools this makes decision tables into a surprisingly powerful knowledge representation method.

References

Agarwal, R. and Tanniru, M. (1991) 'A Petri-net based approach for verifying the integrity of production systems', Workshop Notes, *9th National Conference on AI (AAAI-91)*, Anaheim, California, 17th July 1991. (Distributed by the Authors: Department of MIS, University of Drayton, Drayton, OH 45469, U.S.A.)

Bench-Capon, T. J. M. and Coenen, F. P. (1991) 'The MAKE project: maintenance tools for knowledge-based systems', In Liebowitz, J. (ed.), *Expert Systems World Congress Proceedings*, Vol. 3, Pergamon, pp. 1030–1036.

Clocksin, W. F. and Mellish, C. S. (1981) *Programming in Prolog*, Springer-Verlag, Berlin, Germany.

CODASYL (1982) 'CODASYL, a modern appraisal of decision tables', Report of the Decision Table Task Group, ACM Press, New York, U.S.A.

Coenen, F. (1991) 'A graphical interactive tool for KBS maintenance', In Karagiannis, D. (ed.), *Database and Expert Systems Applications*, Springer-Verlag, pp. 166–196.

Coenen, F. and Bench-Capon, T. (1993) *Maintenance of Knowledge-based Systems*, Academic Press, London, U.K.

Cragun, B. J. and Steudel, H. J. (1987) 'A decision table based processor for checking completeness and consistency in rule-based expert systems', *International Journal of Man-Machine Studies*, **26** (5), 633–648.

Darlington, J. and Reeve, M. (1981) 'ALICE: a multiprocessor reduction machine for the parallel evaluation of applicative languages', *ACM/MIT Conference on Functional Programming Languages and Computer Architecture*, New Hampshire, ACM Press 1981, pp. 50–62.

Dumitrascu, L. (1990) *Generating FORTRAN programs from Decision Tables*, Editura Academiei, Bucharest, Romania.

Ginsberg, A. (1987) 'A new approach to checking knowledge bases for inconsistency and redundancy', In Gupta, U. (ed.), *Validating and Verifying Knowledge-based Systems*, IEEE Computer Society Press, Los Alamitos, California, U.S.A., pp. 120–125.

Gupta, U. (1990) 'A rule-based intelligent test case generator', Workshop Notes: *AAAI 90 Workshop on Validation and Verification*, Boston, Massachusetts, 4 August 1990. (Distributed by the Author: Department of Decision Sciences, E. Carolina State University, Greenville, North Carolina, U.S.A.)

Hicks, R. and Lee, R. (1988) *VP-Expert for Business Applications*, Holden Day Software, Oakland, California, U.S.A.

Maes, R., Vanthienen, J. and Verhelst, M. (1981) 'Procedural decision support through the use of PRODEMO', *Proceedings of the Second International Conference on Information Systems*, Cambridge, Massachusetts, 7–9 December 1981, ACM Press, New York, U.S.A., pp. 149–158.

Maes, R. and Van Dijk, J. (1988) 'On the role of ambiguity and incompleteness in the design of decision tables and rule-based systems', *The Computer Journal*, **31** (6), 481–489.

Maluszynski, J. and Naslund, T. (1989) 'Fail substitutions for negation as failure', *Proceedings of the North American Conference on Logic Programming*, Cleveland, Ohio, M.I.T. Press, Cambridge, Massachusetts, U.S.A., pp. 461–476.

McCarthy, J. (1962) *Lisp 1.5 Programmer's Manual*, M.I.T. Press, Cambridge, Massachusetts, U.S.A.

Merlevede, P. and Vanthienen, J. (1991) 'A structured approach to formalization and validation of knowledge', *Proceedings of the IEEE/ACM International Conference on Developing and Managing Expert System Programs*, Washington DC, IEEE Computer Society Press, Los Alamitos, California, U.S.A., pp. 149–158.

Metzner, J. R. and Barnes, B. H. (1977) *Decision Table Languages and Systems*, Academic Press, New York, U.S.A.

Murrell, S. and Plant, R. T. (1995) 'A graph-reduction implementation of a production system', *Knowledge-Based Systems*, Butterworth-Heineman, to be published.

Nguyen, T. A., Perkins, W. A. and Laffery, T. J. (1985) 'Checking an expert systems knowledge

base for consistency and completeness', *Proceedings of the Ninth International Joint Conference on A.I.*, Los Angeles, California, 18–23 August 1985, Vol. 1. AAAI Press, pp. 375–378.

Nguyen, T. A. (1987) 'Verifying consistency of production systems', *Proceedings of the 3rd Conference on AI Applications*, Kissimmee, Florida, 23–27 February 1987, IEEE Computer Society Press, Los Alamitos, California, U.S.A., pp. 4–7.

Plaza, J. A. (1992) 'Fully declarative logic programming', *Programming Language Implementation and Logic Programming*, Proceedings 1992, Lecture Notes in Computer Science, Vol. 631, pp. 415–427.

Preece, A. D. and Shinghal, R. (1991) 'COVER: a practical tool for verifying rule-based systems', *AAAI Workshop on Validation and Verification Notes*, AAAI 1991. (Distributed by the Authors: Department of Computer Science, University of Aberdeen, U.K.)

Puuronen, S. (1987) 'A tabular rule checking method', *Proceedings of Avignon87*, Vol. 1, pp. 257–268.

Santos-Gomez, L. and Darnell, M. (1992) 'Empirical evaluation of decision tables for constructing and comprehending expert system rules', *Knowledge Acquisition*, **4**, 427–444.

Stachowitz, R. A., Chang, C. L., Stock, T. S. and Coombs, J. B. (1987) In NASA Conference Publication 2491, *First Annual Workshop on Space Operations Automation and Robotics (SOAR'87)*, Johnson Space Center, Houston, Texas, 5–7 August 1987.

Subramanian, G. H., Nosek, J., Raghunathan, S. P. and Kanitkar, S. S. (1992) 'A comparison of the decision table and tree', *Communications of the ACM*, **35** (1), 89–94.

Suwa, M., Scott, A. C. and Shortliffe, E. H. (1984) 'An approach to verifying completeness and consistency in a rule-based expert system', *AI Magazine*, Fall 1984, pp. 16–21.

Tanaka, M., Aoyama, N., Sugiur, A. and Koseki, Y. (1993) 'Integration of multiple knowledge representations for classification problems', *Proceedings of the Fifth International Conference on Tools with AI*, Boston, Massachusetts, 8–11 November 1993, IEEE Computer Society Press, Los Alamitos, California, U.S.A., pp. 448–449.

Vanthienen, J. and Dries, E. (1993) 'Illustration of a decision table tool for specifying and implementing knowledge-based systems', *Proceedings of the Fifth International Conference on Tools with AI*, Boston, Massachusetts, 8–11 November 1993, IEEE Computer Society Press, Los Alamitos, California, U.S.A., pp. 198–205.

Vanthienen, J. and Robben, F. (1993) 'Developing legal knowledge-based systems using decision tables,' *Proceedings of the Fourth International Conference on AI and the Law*, Amsterdam, 15–18 June, 1993, ACM Press, New York, U.S.A., pp. 282–291.

Vanthienen, J. and Wets, G. (1993) 'Interfacing decision tables with knowledge acquisition formalisms', *Proceedings of the Second World Congress on Expert Systems*, Lisbon, Portugal, 10–14 January 1994, Permagon, pp. 1861–1868.

Vanthienen, J. and Dries, E. (1995) 'Decision tables: refining the concept and a proposed standard', *Communications of the ACM*, to be published.

Verhelst, M. R. (1975) 'La table de decision comme technique a l'useage du management et de l'organisation', Universite Catholique de Louvain, Department Toegepaste Economie, Belgium.

Wallace, M. (1987) 'Negation by constraints: a sound and efficient implementation of negation in deductive databases', *Proceedings of the 1987 Symposium on Logic Programming*, San Francisco, IEEE Computer Society Press, Los Alamitos, California, U.S.A., pp. 253–263.

Yoon, J. P. (1989) 'Techniques for data and rule validation in knowledge-based systems', *COMPASS 89, Proceedings of the 4th Conference on Computer Assurance System Integrity, Software Safety and Process Security*, pp. 62–70.