IN-61
33962
P-57

NASA Contractor Report 195027

TBell: A Mathematical Tool for Analyzing Decision Tables

D. N. Hoover and Zewei Chen
Odyssey Research Associates, Inc., Ithaca, New York

Contract NAS1-18972
November 1994

National Aeronautics and
Space Administration
Langley Research Center
Hampton, Virginia 23681-0001

TBell: A Mathematical Tool for Analyzing Decision Tables [1]

D. N. Hoover
Zewei Chen
Odyssey Research Associates, Inc.
301 Dates Dr.
Ithaca NY 14850-1326
Internet: {hoove,chen}@oracorp.com

November 29, 1994

[1] This document is the Final Report for NASA contract NAS1-18972 (Task 13), sponsored by NASA Langley Research Center. Thanks to Lance Sherry, Ted Ralston, Maureen Stillman, David Guaspari, and Dana Hartman for many suggestions improving this paper. Special thanks to David Guaspari for several important technical contributions.

Contents

Chapter 1

Introduction

We take the point of view that system designers prefer to use formal methods whenever their task is well-defined and a suitable formalism is available. Industry does not need to be sold on formal methods or to be told to use a particular formal method, such as a specification language or theorem prover based on formal logic. Rather we must recognize that many of the methods used in industry are semi-formal or formal. For these methods, we must develop mathematical theory and software to help analyze specifications, and offer advice about good methodology. We expect that such an opportunistic approach will give impressive near term results, stretch the imaginations of formal methodists, make formal methods accessible and beneficial to system designers, and help clarify the long-term aims of formal methods research.

This paper describes how we have applied this point of view to one formal specification method commonly used in systems design, the decision table. A decision table is a tabular format for specifying a complex set of rules that choose one of a number of alternative actions. The problem with this form of specification is that it is not always easy to see that the table indicates exactly one possible action for each possible input scenario, that is, the table defines a (total) function. Determining whether a table actually defines a function is essentially a problem in propositional logic. Canned routines available to us [Mar93, SOR93a, McC94] did not seem useful because they did not give very helpful information about what is wrong with a flawed table (usually a lot).

Consequently we built our own prototype tool, called TBell[1], a window-based decision table editor that performs the following functions:

- Shows for which inputs more than one course of action is indicated.
- Shows for which inputs no course of action is indicated.
- Performs an analysis finding structural defects in the table (correlations between variables that mean that the table cannot define a function).
- Generates Ada code implementing the table.

[1] TBell is short for any of: TaBular Environment, LogicaL; T___Bell, TBell's secret code name, alluding to a fictional character known for being quick, small, clever, informative, etc.; or Tabella, Latin for "a little table."

- Generates English-language text documenting the decision table.

All logical input and output is expressed in terms of tables so that the user does not need to learn another formalism (i.e. symbolic logic). TBell's internal functions are written in SML/NJ, with its window interface written in tcl/tk.

The logical manipulations that TBell performs (involved in all of the functions listed above) are based on finite logic, a generalization of propositional logic, and finite decision diagrams, a generalization of binary decision diagrams designed to support finite-valued (as opposed to Boolean-valued) variables. The finite-valued approach eliminates the need to code finite-valued variables using several Boolean-valued variables, which seems to improve performance and simplifies the translation between formulas or decision tables and finite decision diagrams.

The basic coverage analysis functions are relatively straightforward. Structural analysis, however, depends on a nontrivial mathematical innovation.

Because TBell efficiently performs an important class of analyses using a language (that of decision tables) familiar to practitioners in the field and saves labor by generating code and documentation, we feel that it has good prospects of being used advantageously in industry. Generating code and documentation from the formal specification (the decision table) will also eliminate the common problem of specification and documentation getting out of date when the code is modified. As modifications will be made to the decision table, up-to-date code and documentation will be generated from it.

Our aim in this project has been to produce a formal tool of a different nature from conventional program specification and verification. In fact we believe that the difficulty of using such products (in particular, of learning to do a proof) precludes their being adopted by nonspecialists. Nevertheless, we have profited from using two conventional formal methods systems, PVS [SOR93b] and Penelope [GMP90]. PVS is a general theorem prover; Penelope is a formal verification system for Ada. We used PVS in our preliminary investigations of particular decision tables before we understood that we should build our own tool. Later we used PVS to check the correctness of TBell's results. We used Penelope to check the correctness of generated code by verifying some examples.

This work was carried out in cooperation with Honeywell as part of a NASA project on applying formal methods to the development of avionics software. We thank Lance Sherry for guidance in identifying the problems of most concern to practitioners and in identifying the nature of those problems.

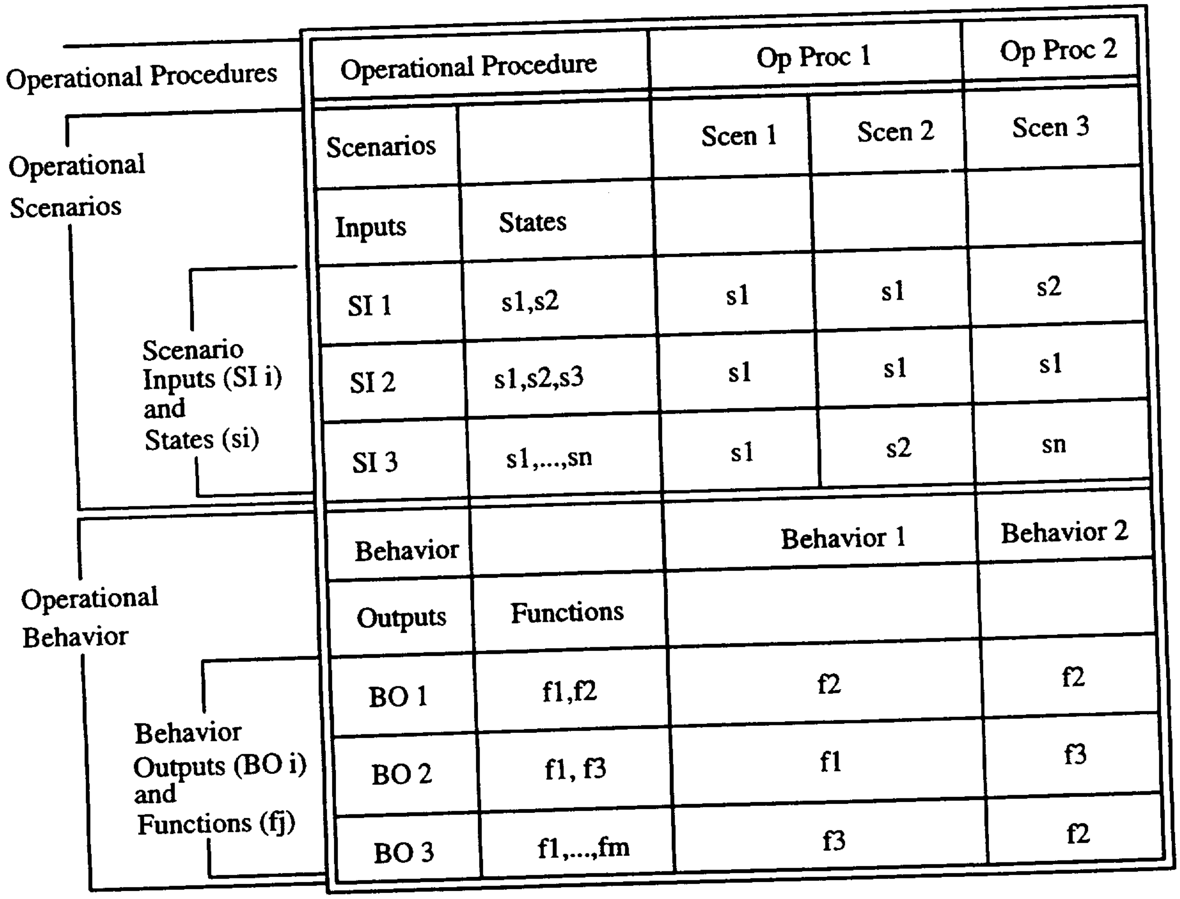
Besides developing a tool to support coverage analysis of decision tables, we developed a new form of decision table, the *partitioned* decision table, which can represent the same information as a decision table much more compactly, in much the same format as it might be described in English or implemented in code. Our tool does not currently support partitioned decision tables, but the logical apparatus necessary has already been included in support of English-language document generation.

Some other tools supporting decision table analysis already exist. The capabilities of some of them are described in Appendix E.

Chapter 2

Decision Tables

A decision table (Figure 2.1, as presented in [She94b]) is a tabular format for specifying a choice of the actions that a system is to take. Each possible action is called an *operational procedure*.


Operational Procedures
Operational Scenarios
Scenario Inputs (SI i) and States (si)
Operational Behavior
Behavior Outputs (BO i) and Functions (fj)

Operational Procedure		Op Proc 1		Op Proc 2
Scenarios		Scen 1	Scen 2	Scen 3
Inputs	States			
SI 1	s1,s2	s1	s1	s2
SI 2	s1,s2,s3	s1	s1	s1
SI 3	s1,...,sn	s1	s2	sn
Behavior		Behavior 1		Behavior 2
Outputs	Functions			
BO 1	f1,f2	f2		f2
BO 2	f1, f3	f1		f3
BO 3	f1,...,fm	f3		f2

Figure 2.1: A decision table with both scenarios and behaviors [She94].

A decision table is divided into two parts.

- A specification of conditions (values of input variables) under which the various operational procedures will be enabled. These enabling conditions are called the *engagement criteria* for the various operational procedures. Each engagement criterion is the union (or disjunction) of more specific situations called *scenarios*, each corresponding to a column of (this half of) the decision table. This part of the decision table is called the *scenarios part*.
- A specification of what actions (changes to values of state variables, output values produced) are associated with each operational procedure. These actions are called *behaviors*. Different parts of the behaviors may be specified either directly or else in terms of further decision tables. This part of the table is called the *behaviors part*.

A decision table is intended to define a function. That is:

- each possible set of values of the input variables satisfies the engagement criterion of exactly one operational procedure; and
- any decision tables used in defining behaviors of operational procedures are functional.

It is clear that the main requirement is the first one, functionality of operational procedure selection, since if that were satisfied for all decision tables in a system specification, then the second requirement would automatically be satisfied. In this project, we regarded each table as an independent entity. In Section 5 we will discuss the possibility of taking advantage of interdependence among tables, such as the fact that a decision table describing part of the behavior of an operational procedure should be able to assume that its inputs satisfy the engagement criterion of that operational procedure.

For the rest of this paper, when we say “decision table” we will mean only the scenarios half.

2.1 Semantics of Decision Tables

As we see from Figure 2.2, a decision table is divided into rows and columns. Their significance can be described as follows.

- First column: a list of parameter or variable names (e.g. `flightphase`) one in each row. (A variable name can be a complex mathematical expression, such as `ac_alt < dap_seg_alt`, but in the table it is still just a variable name.)
- Second column: a list of nonempty lists of state (or value) names, one list in each row. The list in row n represents the set of possible states (or values) of the parameter named in row n. For example, the variable `flightphase` can take any of the values `takeoff`, `climb`, `cruise`, `descent`, or `approach`.

Operational Procedure		Climb					
Input Param	States						
flightphase	takeoff, climb, cruise, descent approach	takeoff climb cruise	takeoff climb cruise	takeoff climb cruise	takeoff climb cruise	takeoff climb cruise	takeoff climb cruise
fcc_alt_capt_hold	on, off	on	on	on	off	off	off
prev_vg_ref_alt = vg_ref_alt	TRUE, FALSE	FALSE	FALSE	FALSE	*	*	*
lvg_ref_alt - cruise_alt l <= cruise_alt_tol	TRUE, FALSE	*	*	*	*	*	*
cruise_alt_valid	TRUE, FALSE	*	*	*	*	*	*
ac_alt < dap_seg_alt	TRUE, FALSE	*	*	*	*	*	*
dap_exists	TRUE, FALSE	*	*	*	*	*	*
dap_overspeed_cond	TRUE, FALSE	*	*	*	*	*	*
dap_capt_hold_cond	TRUE, FALSE	*	*	*	*	*	*
hm_cond	TRUE, FALSE	*	*	*	*	*	*
vg_type = profile	TRUE, FALSE	*	*	*	*	*	*
engine_out	TRUE, FALSE	FALSE	TRUE	TRUE	FALSE	TRUE	TRUE
above_eo_max_alt	TRUE, FALSE	*	FALSE	*	*	FALSE	*
compare(vg_ref_alt, eo_safe_speed_alt)	LT, EQ,GT	*	GT	GT	*	GT	GT
eo_decel_cond	TRUE, FALSE	*	*	FALSE	*	*	FALSE

Figure 2.2: A simple decision table.

- Other columns: the *body* of the table, grouped under headings. The headings are operational procedure names. (In Figure 2.2 the only operational procedure named is `Climb`, for reasons of space.) Each column is called a *scenario*. For each parameter listed in the first column of the table, a scenario lists in the same row a set of values for that variable compatible with that scenario. An asterisk "*" indicates that any value will do (i.e. it is equivalent to listing all the values listed in the second column of the same row).

> The engagement criterion for a given operational procedure consists of all assignments of values to parameters that are compatible with one of the scenarios listed under that operational procedure.

That is what we call a *concrete* decision table. A more abstract mathematical model of a decision table can be given as follows. This model is based on a less rigorous decision table semantics given in [She94a].

Given a set S, let $\mathcal{P}_+(S)$ denote the set of nonempty subsets of S.

An abstract decision table consists of the following.

- A finite set $P = \{p_1, \ldots, p_n\}$ of parameters (all the parameters listed in the first column of the table).
- A finite set V of values (all the values listed anywhere in the second column).
- A map $\sigma : P \to \mathcal{P}_+(V)$. The map σ is called the *signature* of the table. Let $S_i = \sigma(p_i)$, $i = 1, \ldots, n$. (S_i is the set of values listed in the ith row of the second column.)
- A finite set OP of operational procedures. (For present purposes, these are just names without any further interpretation.)
- A *scenario* for a signature σ is a map $s : P \to \mathcal{P}_+(V)$ where for each $p \in P$, $s(p) \subseteq \sigma(p)$. Equivalently, a scenario is an n-tuple $(s_1, \ldots, s_n)$, where $s_i = s(p_i) \subseteq S_i$, $i = 1, \ldots, n$. (Each column of a table body stands for the scenario $(s_1, \ldots, s_n)$ where s_i is the set of values listed in the ith row, or S_i if "*" appears.)

  Let Σ be the set of scenarios for σ.
- The *body* of T is a map $\mathsf{body} : \mathsf{OP} \to \mathcal{P}_+(\Sigma)$ (set of scenarios in which an operational procedure is engaged). (In a concrete table, $\mathsf{body}(\mathsf{opp})$ is the set of scenarios represented by columns listed under opp.)

Given an abstract decision table T, the engagement criterion $\mathsf{eng\_crit}(\mathsf{opp})$ of an operational procedure opp is the predicate

$$\mathsf{eng\_crit}(\mathsf{opp}) \subseteq S_1 \times \ldots \times S_n$$

given by

$$(v_1, \ldots, v_n) \in \mathsf{eng\_crit}(\mathsf{opp})$$

if and only if there exists a scenario $(s_1, \ldots, s_n) \in \mathsf{body}(\mathsf{opp})$ such that

$$(v_1 \in s_1 \,\text{and} \ldots \text{and}\, v_n \in s_n)\,.$$

The *denotation* (that is, mathematical meaning) of the abstract decision table T is the relation

$$R_T \subseteq (S_1 \times \ldots \times S_n) \times \mathsf{OP}$$

given by

$$((v_1,\ldots,v_n),\mathsf{opp}) \in R_T \text{ iff } (v_1,\ldots,v_n) \in \mathsf{eng\_crit(opp)}.$$

That is, operational procedure opp is enabled when the parameters $p_1,\ldots,p_n$ have respective values $v_1,\ldots,v_n$ if and only if $v_1,\ldots,v_n$ are compatible with some scenario of opp.

The abstract table T is *functional* ("correct") if R_T is actually a function, that is, for each $(s_1,\ldots,s_n) \in S_1 \times \ldots \times S_n$ there is exactly one operational procedure opp such that $((s_1,\ldots,s_n),\mathsf{opp}) \in R_T$. The property of being functional is equivalent to the conjunction of the following two properties.

T is *exclusive* if and only if for each $(s_1,\ldots,s_n) \in S_1 \times \ldots \times S_n$ there is at most one operational procedure opp such that

$$((s_1,\ldots,s_n),\mathsf{opp}) \in R_T.$$

T is *exhaustive* if and only if for each $(s_1,\ldots,s_n) \in S_1 \times \ldots \times S_n$ there is at least one operational procedure opp such that $((s_1,\ldots,s_n),\mathsf{opp}) \in R_T$.

An advantage of decision tables is that, at least for small tables, it is possible to check exclusiveness by inspection. Exhaustiveness is much harder.

2.2 Semantics of the Table in Figure 2.2

As an example, we will define the various parts of the semantics for the table in Figure 2.2.

- There are fifteen parameters. Note that some of the names contain white space and/or relation symbols. Nevertheless, they name single parameters.

```
P = {
        flightphase,
        fcc_alt_capt_hold,
        prev_vg_ref_alt = vg_ref_alt,
        |vg_ref_alt - cruise_alt| <= cruise_alt_tol,
        cruise_alt_valid,
        ac_alt < dap_seg_alt,
        dap_exists,
        dap_overspeed_cond,
        dap_capt_hold_cond,
        hm_cond,
        vg_type = profile,
        engine_out,
        above_eo_max_alt,
        compare(vg_ref_alt,eo_safe_speed_alt),
        eo_decel_cond
    }.
```

- There are twelve values.

$$\begin{aligned} V \;=\; \{\; & \texttt{takeoff},\texttt{climb},\texttt{cruise},\texttt{descent},\texttt{approach},\texttt{on},\texttt{off},\\ & \texttt{TRUE},\texttt{FALSE},\texttt{LT},\texttt{EQ},\texttt{GT}\}. \end{aligned}$$

- $\sigma(\texttt{flightphase}) = \{\texttt{takeoff},\ \texttt{climb},\ \texttt{cruise},\ \texttt{descent},\ \texttt{approach}\}$,
  $\sigma(\texttt{fcc\_alt\_capt\_hold}) = \{\texttt{on},\texttt{off}\}$,
  $\sigma(\texttt{engine\_out}) = \{\texttt{TRUE},\ \texttt{FALSE}\}$, etc.

- $\mathsf{OP} = \{\texttt{Climb},\ \texttt{Descent Intermediate Level},\ldots\}$. Only Climb is shown in Figure 2.2.

- Each column in the body of the table represents a scenario. The first column represents the scenario s_1 that satisfies

$$\begin{aligned} &s_1(\texttt{flightphase}) = \{\texttt{takeoff},\ \texttt{climb},\ \texttt{cruise}\},\\ &s_1(\texttt{fcc\_alt\_capt\_hold}) = \{\texttt{off}\},\\ &s_1(\texttt{dap\_exists}) = \{\texttt{TRUE},\ \texttt{FALSE}\},\ \text{etc.} \end{aligned}$$

  (Of course, this is only a partial description of s_1.) Viewed as a tuple, the scenario is the 15-tuple

$$(\{\texttt{takeoff},\ \texttt{climb},\ \texttt{cruise}\},\{\texttt{on}\},\{\texttt{FALSE}\},\{\texttt{TRUE},\ \texttt{FALSE}\},\{\texttt{TRUE},\ \texttt{FALSE}\},\ldots).$$

- $\mathsf{body}(\texttt{Climb}) = \{s_1,\ldots,s_6\}$, where each s_i, $i = 1,\ldots,6$ is the scenario corresponding to the ith column (and in particular s_1 is as defined in the previous item).

- $(v_1,\ldots,v_{15}) \in \mathsf{eng\_crit}(\texttt{Climb})$ iff for $i = 1,\ldots,15$, $v_i \in \sigma(v_i)$, and any one (or more) of the following six conditions holds.
  - $v_1 \in \{\texttt{takeoff},\ \texttt{climb},\ \texttt{cruise}\}$, $v_2 = \texttt{on}$, $v_3 = \texttt{FALSE}$, and $v_{12} = \texttt{FALSE}$.
  - $v_1 \in \{\texttt{takeoff},\ \texttt{climb},\ \texttt{cruise}\}$, $v_2 = \texttt{on}$, $v_3 = \texttt{FALSE}$, $v_{12} = \texttt{TRUE}$, $v_{13} = \texttt{FALSE}$, and $v_{14} = \texttt{GT}$.
  - $v_1 \in \{\texttt{takeoff},\ \texttt{climb},\ \texttt{cruise}\}$, $v_2 = \texttt{on}$, $v_3 = \texttt{FALSE}$, $v_{12} = \texttt{TRUE}$, $v_{14} = \texttt{GT}$, and $v_{15} = \texttt{FALSE}$.
  - $v_1 \in \{\texttt{takeoff},\ \texttt{climb},\ \texttt{cruise}\}$, $v_2 = \texttt{off}$, and $v_{12} = \texttt{FALSE}$.
  - $v_1 \in \{\texttt{takeoff},\ \texttt{climb},\ \texttt{cruise}\}$, $v_2 = \texttt{off}$, $v_{12} = \texttt{TRUE}$, $v_{13} = \texttt{FALSE}$, and $v_{14} = \texttt{GT}$.
  - $v_1 \in \{\texttt{takeoff},\ \texttt{climb},\ \texttt{cruise}\}$, $v_2 = \texttt{on}$, $v_{12} = \texttt{TRUE}$, $v_{14} = \texttt{GT}$, and $v_{15} = \texttt{FALSE}$.

2.3 Partitioned Decision Tables and their Semantics

A disadvantage of decision tables as described above is that they are not very dense in information. Consequently, they can be very large, making it difficult to ascertain anything about them by eye. In this section, we propose a more compact notation, which we call *partitioned*

<table>
<tr><td colspan="3">Operational Procedure</td><td colspan="3">Climb</td><td colspan="3">Descent Intermediate Level</td><td>etc.</td></tr>
<tr><td>Category</td><td>Input Param</td><td>States</td><td colspan="3"></td><td colspan="2"></td><td></td><td></td></tr>
<tr><td>Flight
Phase</td><td>flightphase</td><td>takeoff, climb,
cruise, descent
approach</td><td colspan="3">takeoff
climb
cruise</td><td colspan="2">descent
approach</td><td>approach</td><td></td></tr>
<tr><td rowspan="4">Altitude
Target
Status</td><td>fcc_alt_capt_hold</td><td>on, off</td><td>on</td><td colspan="2">off</td><td>on</td><td>on</td><td>on</td><td></td></tr>
<tr><td>prev_vg_ref_alt
= vg_ref_alt</td><td>TRUE, FALSE</td><td>FALSE</td><td colspan="2">*</td><td>TRUE</td><td>TRUE</td><td>TRUE</td><td></td></tr>
<tr><td>|vg_ref_alt -
cruise_alt| <=
cruise_alt_tol</td><td>TRUE, FALSE</td><td>*</td><td colspan="2">*</td><td>FALSE</td><td>*</td><td>TRUE</td><td></td></tr>
<tr><td>cruise_alt_valid</td><td>TRUE, FALSE</td><td>*</td><td colspan="2">*</td><td>*</td><td>FALSE</td><td>*</td><td></td></tr>
<tr><td rowspan="4">Desc/Appr
Path
Status</td><td>ac_alt < dap_seg_alt</td><td>TRUE, FALSE</td><td colspan="3">*</td><td colspan="2">*</td><td>*</td><td></td></tr>
<tr><td>dap_exists</td><td>TRUE, FALSE</td><td colspan="3">*</td><td colspan="2">*</td><td>*</td><td></td></tr>
<tr><td>dap_overspeed_cond</td><td>TRUE, FALSE</td><td colspan="3">*</td><td colspan="2">*</td><td>*</td><td></td></tr>
<tr><td>dap_capt_hold_cond</td><td>TRUE, FALSE</td><td colspan="3">*</td><td colspan="2">*</td><td>FALSE</td><td></td></tr>
<tr><td>Hold
Manual</td><td>hm_cond</td><td>TRUE, FALSE</td><td colspan="3">*</td><td colspan="2">*</td><td>*</td><td></td></tr>
<tr><td>VG Type</td><td>vg_type = profile</td><td>TRUE, FALSE</td><td colspan="3">*</td><td colspan="2">*</td><td>*</td><td></td></tr>
<tr><td rowspan="4">Engine-
Out
Status</td><td>engine_out</td><td>TRUE, FALSE</td><td>FALSE</td><td>TRUE</td><td>TRUE</td><td colspan="2">*</td><td>*</td><td></td></tr>
<tr><td>above_eo_max_alt</td><td>TRUE, FALSE</td><td>*</td><td>FALSE</td><td>*</td><td colspan="2">*</td><td>*</td><td></td></tr>
<tr><td>compare(vg_ref_alt,
eo_safe_speed_alt)</td><td>LT, EQ, GT</td><td>*</td><td>GT</td><td>GT</td><td colspan="2">*</td><td>*</td><td></td></tr>
<tr><td>eo_decel_cond</td><td>TRUE, FALSE</td><td>*</td><td>*</td><td>FALSE</td><td colspan="2">*</td><td>*</td><td></td></tr>
</table>

Figure 2.3: A partitioned decision table

decision tables. For contrast, we will refer to the usual style of decision table, defined above, as *simple* decision tables. The structure of partitioned tables also corresponds closely to the layout we use for English-language documentation (Section 4.5, patterned after the Honeywell SRD [Hon94]).

Our decision table analysis tool, TBell, does not currently support partitioned decision tables, but it contains the necessary logical apparatus.

Figure 2.3, shows a partitioned decision table. The single macro-column under the heading "Climb" in Figure 2.3 corresponds to all the columns under the same heading in Figure 2.2.

The following features distinguish a partitioned decision table from a simple one:

- Horizontal and vertical double lines partition the table.
- The horizontal double lines partition the variables into *categories*.
- The vertical double lines partition the body of the table into *macro-columns.*
- In the body of the table, double lines mark off boxes we call *macro-entries*. Each macro-entry is divided into micro-rows and micro-columns by single vertical and horizontal lines, like the body of a simple decision table. A macro-entry is like the body of a mini-simple decision table.

The meaning of a partitioned decision table is the same as that of the simple table obtained by replacing each macro-column by the set of simple columns obtained by piecing together one micro-column from each category in the macro-column. This idea can form the basis of a mathematical semantics similar to that given for simple decision tables.

Chapter 3

Logical Basis

TBell uses a generalization of binary decision diagrams (BDDs) that we call *finite decision diagrams* (FDDs). The idea involved is quite simple, but seems to be new or at least not generally known. As we feel that most systems using BDDs will want to adopt it, we describe it here. The underlying idea is that most methods applicable to Boolean or propositional logic generalize to what we call *finite logic*, logic in which the variables can take finitely many values instead of only two. The application to decision tables, and all systems involving a finite state, is obvious since variables in a decision table can take more than two values. The main observation is that most techniques for manipulating propositional logic generalize straightforwardly to finite logic. In particular, the finite logic generalization of the if-then-else expression is the case expression, which obeys the same laws. BDDs are essentially a special if-then-else normal form. FDDs are the corresponding case normal form, and can be manipulated using essentially the same methods.

Another way of looking at FDDs is that they are just decision trees in a particular form, with special rules for manipulating them.

Of course finite logic includes propositional logic because variables in finite logic *can* take just the values **true** and **false**. On the other hand, finite logic can be coded into propositional logic by coding each finite valued variable by several Boolean-valued variables, as has been common practise, but such coding is unnecessary, and translation to and from FDD representations is simpler and more natural without coding into Boolean variables.

In our logical discussions, we write

$$A \equiv B$$

to indicate that A is equivalent to B, that is, each is deducible from the other, and

$$A \Rightarrow B$$

to indicate that B is deducible from A.

3.1 Finite Logic

A *finite language* L consists of a set P of variables together with a signature σ that maps each $p \in P$ to a finite set of values $\sigma(p)$, the *type* of p. A *literal* of L is a formula

$$p \in s \tag{3.1.1}$$

where $s \subseteq \sigma(p)$. (If $s = \sigma(p)$, then (3.1.1) is the identically true proposition, and if $s = \emptyset$, then p is identically false.) A *formula* of L is just a boolean combination of literals.

An assignment (or valuation) of L is a mapping α with domain P such that for all $p \in P$, $\alpha(p) \in \sigma(p)$; α satisfies a literal $p \in s$ (write $\alpha(p \in s) = \textsf{true}$), if $\alpha(p) \in s$. For a complex formula A, $\alpha(A)$ is defined as propositional logic; for example, $\alpha(A \wedge B) = \textsf{true}$ iff $\alpha(A) = \textsf{true}$ and $\alpha(B) = \textsf{true}$. A formula is *valid* if and only if it satisfies every assignment.

Which rules or formulas are valid in a finite logic depends on the signature. For example, in the logic given by the signature of the table in Figure 2.2,

$$\begin{array}{l} \texttt{flightphase} \in \{\texttt{takeoff, climb}\} \vee \texttt{flightphase} \in \{\texttt{cruise}\} \\ \equiv \\ \texttt{flightphase} \in \{\texttt{takeoff, climb, cruise}\} \end{array}$$

and

$$\texttt{comp(vg\_ref\_alt, eo\_safe\_speed\_alt)} \in \{\texttt{LT, EQ, GT}\}$$

is valid. Nevertheless, it appears that all methods applicable to propositional logic generalize easily and naturally to finite logic. We describe finite decision diagrams, a generalization of Boolean decision diagrams, below. We also developed and studied a generalization of resolution, but finite decision diagrams proved more effective.

3.2 Finite Decision Diagrams

Boolean decision diagrams (BDDs) were introduced by Lee [Lee59] and later popularized by Akers [Ake78]. Since Bryant [Bry86] they have been recognized as the most important practical method for representing Boolean expressions. Essentially, a BDD is an ordered if-then-else normal form for Boolean expressions with subformula sharing, depending on only a few simple identities involving if-then-else. Since the laws for manipulating an expression

$$\textsf{if } b \textsf{ then } c \textsf{ else } d$$

have nothing at all to do with the (common) type of c and d, BDDs were later generalized to Algebraic Decision Diagrams [BFG+93] or Multi-Terminal Decision Diagrams [CFM+93], which can be expressions of any type. Here we offer the further observation that if-then-else has a perfectly good finitely branching analog, the *case* expression,

$$\textsf{case } p \textsf{ of } c_1^p \Rightarrow a_1 \mid \ldots \mid c_{k_p}^p \Rightarrow a_{k_p},$$

where p is a finite-valued variable, $c_1^p, \ldots, c_{k_p}^p$ are all the possible values of p, and each of $a_1, \ldots, a_{k_p}$ is an expression of some given type T. Case expressions satisfy the obvious generalizations of all the same laws as if-then-else expressions. We call the corresponding generalization of BDDs Finite Decision Diagrams. They are essentially finitely branching decision trees in a normal form with hashing of common subtrees (so that they are really decision graphs, or diagrams, rather than trees). Here is the precise definition of the normal form.

- A strict linear order $<$ on propositional variables is given.
- Each subterm of an FDD of type T is either an element of T or else of the form

  $$\textsf{case } p \textsf{ of } c_1^p \Rightarrow a_1 \mid \ldots \mid c_{k_p}^p \Rightarrow a_{k_p}$$

  where p is a variable of L and for each variable q occurring in any of $a_1, \ldots, a_{k_p}$, $p < q$.
- An FDD has no vacuous case splits

  $$\textsf{case } p \textsf{ of } c_1^p \Rightarrow a \mid \ldots \mid c_{k_p}^p \Rightarrow a$$

  (such a term is equivalent to just a).
- Subterms are hashed to save space and computation.

We have not yet implemented the hashing in TBell, so our FDDs are really finite decision trees.

The point is that the normal form is unique (equivalent formulas have the same normal form) because of the ordering and the elimination of superfluous case splits. In particular, for Boolean-valued FDDs the normal form of the Boolean expression **true** is just **true** and the normal form of **false** is just **false**. Hence valid or contradictory formulas can be detected simply by putting them in normal form. Equivalent FDDs are equal.

It is easy to operate on FDDs and obtain an FDD representation of the result. The basic idea is shown by the following basic identity of case expressions.

$$\begin{aligned}&f(p_1, \ldots, p_n) \equiv \\ &\quad \textsf{case } p_1 \textsf{ of} \\ &\quad\quad c_1^{p_1} \Rightarrow f(c_1^{p_1}, p_2, \ldots, p_n) \mid \cdots \mid \\ &\quad\quad c_{k_{p_1}}^{p_1} \Rightarrow f(c_{k_{p_1}}^{p_1}, p_2, \ldots, p_n).\end{aligned}$$

Once function applications have been pushed down to the leaves, any vacuous case splits have been eliminated, and new subformulas have been entered in the hashtable, we have an FDD. For binary functions, we use a more complicated identity to preserve the order of the variables. For notational simplicity, we give this identity for if-then-else. The generalization to case expressions and to N-ary is obvious.

$$\begin{aligned}&g(\textsf{if } p \textsf{ then } b \textsf{ else } c, \textsf{if } p' \textsf{ then } b' \textsf{ else } c') \\ &= \textsf{if } p \textsf{ then } g(b, \textsf{if } p' \textsf{ then } b' \textsf{ else } c') \textsf{ else } g(c, \textsf{if } p' \textsf{ then } b' \textsf{ else } c'),\ p < p', \\ &= \textsf{if } p \textsf{ then } g(b, b') \textsf{ else } g(c, c'),\ p = p', \\ &= \textsf{if } p' \textsf{ then } g(\textsf{if } p \textsf{ then } b \textsf{ else } c, b') \textsf{ else } g(\textsf{if } p \textsf{ then } b \textsf{ else } c, c'),\ p' < p.\end{aligned} \tag{3.2.2}$$

Bryant notes that the ternary form is useful for function composition, but it is the binary form that is crucial.

One can prove by induction on the number of variables that a function of finite valued variables is equivalent to an FDD because if $p_1 < \ldots < p_n$ are finite-valued variables then

$$f(p_1,\ldots,p_n) \equiv \mathsf{case}\ p_1\ \mathsf{of}\ c_1^{p_1} \Rightarrow f(c_1^{p_1},p_2,\ldots,p_n) \mid \ldots \mid c_{k_{p_1}}^{p_1} \Rightarrow f(c_{k_{p_1}}^{p_1},p_2,\ldots,p_n).$$

(This generalizes Shannon normal form.)

Note that, in theory, the variables in a decision diagram need not be finite-valued, but operations on the resulting diagrams would not be computable. One could, however, generalize further by permitting the variables to be arbitrary-valued but requiring only finitely many distinct cases at each node. Finiteness is not required at all for the values at the leaves, however.

All of TBell's operations are based on FDD manipulations. All are straightforward, except for structural analysis.

An example. Here is a simple example of how FDDs work. Suppose that σ is a signature such that $\sigma(p) = \{a, b, c\}$ and $\sigma(q) = \sigma(r) = \{\mathsf{TRUE},\ \mathsf{FALSE}\}$. We will abbreviate a formula such as $q \in \{\mathsf{TRUE}\}$ by q and $q \in \{\mathsf{FALSE}\}$ by $\neg q$.

Consider the formula (boolean valued function of p, q, r) P given by

$$p \in \{b, c\} \text{ and } (q \text{ or } r).$$

There are two ways to represent P as an FDD (with the variables in alphabetical order). The first applies the idea of Shannon Normal Form given above.

$$\begin{array}{ll}
 & p \in \{b,c\} \text{ and } (q \text{ or } \neg r) \\
\equiv & \mathsf{case}\ p\ \mathsf{of} \\
 & \quad a \Rightarrow \mathsf{FALSE} \text{ and } (q \text{ or } \neg r) \mid \\
 & \quad b,c \Rightarrow \mathsf{TRUE} \text{ and } (q \text{ or } \neg r) \\
\equiv & \mathsf{case}\ p\ \mathsf{of} \\
 & \quad a \Rightarrow \mathsf{FALSE} \mid \\
 & \quad b,c \Rightarrow (q \text{ or } \neg r) \\
\equiv & \mathsf{case}\ p\ \mathsf{of} \\
 & \quad a \Rightarrow \mathsf{FALSE} \mid \\
 & \quad b,c \Rightarrow (\mathsf{case}\ q\ \mathsf{of}\ \mathsf{TRUE} \Rightarrow \mathsf{TRUE} \mid \mathsf{FALSE} \Rightarrow \neg r) \\
\equiv & \mathsf{case}\ p\ \mathsf{of} \\
 & \quad a \Rightarrow \mathsf{FALSE} \mid \\
 & \quad b,c \Rightarrow (\mathsf{case}\ q\ \mathsf{of}\ \mathsf{TRUE} \Rightarrow \mathsf{TRUE} \mid \mathsf{FALSE} \Rightarrow \mathsf{case}\ r\ \mathsf{of}\ \mathsf{TRUE} \Rightarrow \mathsf{FALSE} \mid \mathsf{FALSE} \Rightarrow \mathsf{FALSE})
\end{array}$$

The other way to transform P into an FDD is to represent its atomic components as FDDs and then apply eq. (3.2.2) to build up the BDD representation of P. Thus,

$$q \equiv \mathsf{case}\ q\ \mathsf{of}\ \mathsf{TRUE} \Rightarrow \mathsf{TRUE} \mid \mathsf{FALSE} \Rightarrow \mathsf{FALSE}$$

and

$$\neg r \equiv \textsf{case } r \textsf{ of } \mathsf{TRUE} \Rightarrow \mathsf{FALSE} \mid \mathsf{FALSE} \Rightarrow \mathsf{TRUE}.$$

Thus,

$$\begin{aligned}
& q \text{ or } \neg r \\
\equiv\ & (\textsf{case } q \textsf{ of } \mathsf{TRUE} \Rightarrow \mathsf{TRUE} \mid \mathsf{FALSE} \Rightarrow \mathsf{FALSE}) \text{ or} \\
& (\textsf{case } r \textsf{ of } \mathsf{TRUE} \Rightarrow \mathsf{FALSE} \mid \mathsf{FALSE} \Rightarrow \mathsf{TRUE}) \\
\equiv\ & (\textsf{case } q \textsf{ of} \\
& \mathsf{TRUE} \Rightarrow \mathsf{TRUE}\text{or}(\textsf{case } r \textsf{ of } \mathsf{TRUE} \Rightarrow \mathsf{FALSE} \mid \mathsf{FALSE} \Rightarrow \mathsf{TRUE}) \mid \\
& \mathsf{FALSE} \Rightarrow \mathsf{FALSE} \text{ or } (\textsf{case } r \textsf{ of } \mathsf{TRUE} \Rightarrow \mathsf{FALSE} \mid \mathsf{FALSE} \Rightarrow \mathsf{TRUE}) \\
\equiv\ & (\textsf{case } q \textsf{ of} \\
& \mathsf{TRUE} \Rightarrow (\textsf{case } r \textsf{ of } \mathsf{TRUE} \Rightarrow \mathsf{TRUE} \text{ or } \mathsf{FALSE} \mid \mathsf{FALSE} \Rightarrow \mathsf{TRUE} \text{ or } \mathsf{TRUE}) \mid \\
& \mathsf{FALSE} \Rightarrow (\textsf{case } r \textsf{ of } \mathsf{TRUE} \Rightarrow \mathsf{FALSE} \text{ or } \mathsf{FALSE} \mid \mathsf{FALSE} \Rightarrow \mathsf{FALSE} \text{ or } \mathsf{TRUE}) \\
\equiv\ & (\textsf{case } q \textsf{ of} \\
& \mathsf{TRUE} \Rightarrow (\textsf{case } r \textsf{ of } \mathsf{TRUE} \Rightarrow \mathsf{TRUE} \mid \mathsf{FALSE} \Rightarrow \mathsf{TRUE}) \mid \\
& \mathsf{FALSE} \Rightarrow (\textsf{case } r \textsf{ of } \mathsf{TRUE} \Rightarrow \mathsf{FALSE} \mid \mathsf{FALSE} \Rightarrow \mathsf{TRUE}) \\
\equiv\ & (\textsf{case } q \textsf{ of} \\
& \mathsf{TRUE} \Rightarrow \mathsf{TRUE} \mid \\
& \mathsf{FALSE} \Rightarrow (\textsf{case } r \textsf{ of } \mathsf{TRUE} \Rightarrow \mathsf{FALSE} \mid \mathsf{FALSE} \Rightarrow \mathsf{TRUE}).
\end{aligned}$$

To get the representation for P, observe that

$$p \in \{b, c\} \equiv \textsf{case } p \textsf{ of } a \Rightarrow \mathsf{FALSE} \mid b, c \Rightarrow \mathsf{TRUE},$$

then apply eq. (3.2.2) with "and" instead of "or."

Chapter 4

Tool Description

This section describes the various functions of TBell. TBell presently supports only simple decision tables, but we intend to add support for partitioned decision tables.

The logical analysis, code generation, and English-language generation modules of TBell are all written in Standard ML of New Jersey (SML/NJ). Its window interface is written in tcl/tk.

Using SML as the programming language for TBell permitted rapid, almost bug-free development of the program. Furthermore, the implementation of decision tables corresponds almost exactly to the mathematical definition of symbol tables given in Section 2 (including concrete and abstract tables as well as the semantics of tables). Performance seems to be quite memory dependent. Our FDD module performs comparably to a more mature and sophisticated BDD package written in C, perhaps because it treats FDDs directly instead of coding them as BDDs. But that is one of the advantages of SML—one can concentrate on the mathematics and the algorithms instead of the coding.

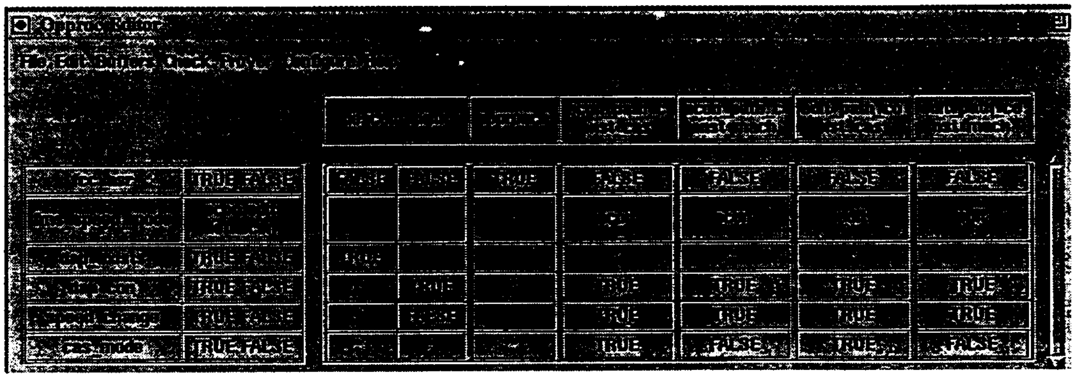
The logical analyzer has three functions: detecting overlap (exclusiveness failure) between engagement criteria of different operational procedures, detecting non-coverage (exhaustiveness failure), and detecting specific structural defects in a decision table that prevent it from being functional. Logical analysis is a purely pushbutton affair, as is code generation. English-language documentation requires some additional information about how to describe variables in English and how to group related variables.

One begins by invoking TBell's window-based decision table editor and entering the table to be analyzed. This table may be stored and re-edited at a later time. An example decision table, derived (with one change) from Section 10.2.5.5 of [Hon94], a case of speed scenario selection, is shown in Figure 4.1.

4.1 Exclusiveness Testing

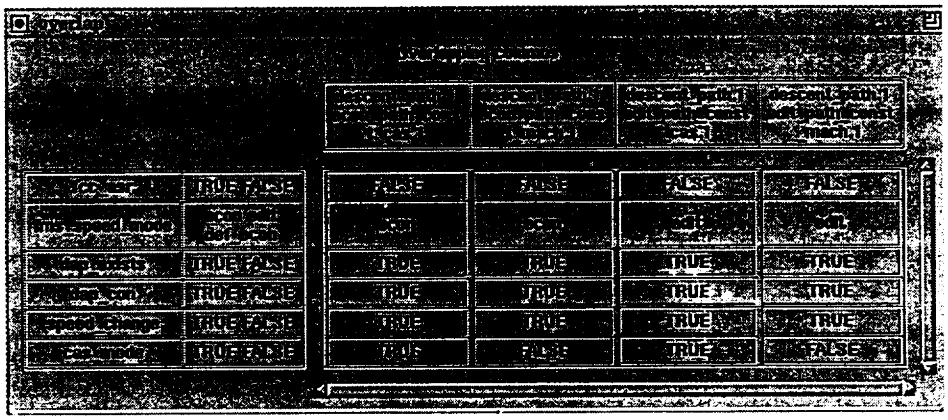
TBell can test for overlap at three granularities: whether two given columns overlap, whether the engagement criteria of two given operational procedures overlap, or whether there is any pair of operational procedures whose engagement criteria overlap. In each case, the output format

ORIGINAL PAGE IS OF POOR QUALITY



		[illegible]		[illegible]	[illegible]	[illegible]	[illegible]	[illegible]
[illegible]	TRUE FALSE	FALSE	FALSE	TRUE	FALSE	FALSE	FALSE	FALSE
[illegible]	[illegible]				[illegible]	[illegible]	[illegible]	[illegible]
[illegible]	TRUE FALSE	TRUE						
[illegible]	TRUE FALSE		TRUE		TRUE	TRUE	TRUE	TRUE
[illegible]	TRUE FALSE		FALSE		TRUE	TRUE	TRUE	TRUE
[illegible]	TRUE FALSE				TRUE	FALSE	TRUE	FALSE

Figure 4.1: Speed scenario selection table.



		[illegible]	[illegible]	descent_path.1 econ_path_const_ cas.1	descent_path.1 econ_path_const_ mach.1
[illegible]	TRUE FALSE	FALSE	FALSE	FALSE	FALSE
[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]
[illegible]	TRUE FALSE	TRUE	TRUE	TRUE	TRUE
[illegible]	TRUE FALSE	TRUE	TRUE	TRUE	TRUE
speed_change	TRUE FALSE	TRUE	TRUE	TRUE	TRUE
cas_mode	TRUE FALSE	TRUE	FALSE	TRUE	FALSE

Figure 4.2: Speed scenarios: overlap table.

is the same: a listing of all pairs of overlapping columns (belonging to distinct operational procedures) among the pairs tested, as in Figure 4.2. Columns are given names of the form opp.*n*, indicating the *n*th column listed under opp. The columns listed under descent_path.1 econ_path_const_cas.1 indicate *exactly* the overlap between these two columns.

4.2 Exhaustiveness Testing

The exhaustiveness test returns a table consisting of a set of columns defining scenarios not covered by the decision table. In logical terms, the columns of a simple decision table can be looked at as clauses (conjunctions of literals) in a finite-valued (as opposed to Boolean-valued) logic—see Section 3.1. The non-coverage table returned is essentially a disjunctive normal form (disjunction of clauses) of the negation of the disjunction of the columns in the body of the input table. Figure 4.3 shows the non-coverage table for the decision table given in Figure 4.1. The table returned is not unique. As the logical analyzer presently works, the non-coverage table is determined by the order of the variables in the input table. There exist heuristics for

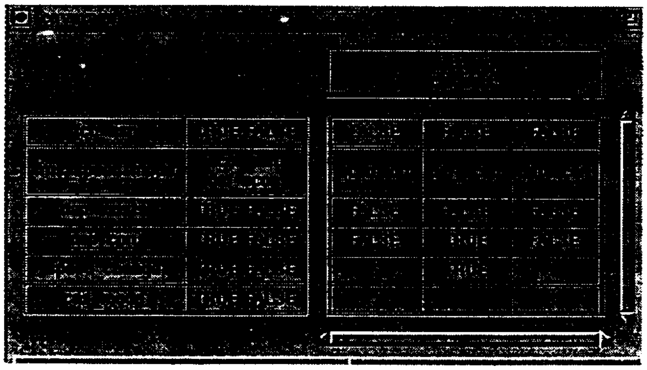


Figure 4.3: Speed scenarios: scenarios not covered.

varying the order of variables [Rud93] that could produce a more compact output.

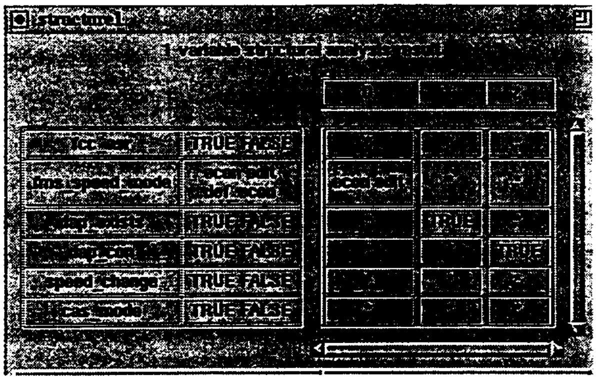
The table returned may be tacked onto the input table rather than returned as a separate table. In the future, doing so may facilitate formally recognizing some of the columns not covered as *illegal* scenarios, that is, scenarios that, we claim, inputs to the table will never satisfy. We will discuss this the idea of illegal scenarios further in Section 5.2.

4.3 Structural Analysis

Because overlap is relatively easy to detect by eye, there are usually few overlaps in a table. Furthermore, the overlap table produced by TBell points directly to the culprits responsible. Non-coverage, on the other hand, is much more difficult to deal with and, in practise, we find that tables often leave many scenarios uncovered. Hence the non-coverage table produced by TBell may be large, and one may be at a loss where to start mending the input table. Furthermore, one cannot point to any particular columns responsible for non-coverage, since it is the whole table that is responsible. For that reason we have developed a form of structural analysis that can detect specific defects in tables that we have noticed occur often in practise. It amounts to pointing to a few variables, rather than columns, responsible for non-functionality. The method we use to search for structural defects is mathematically based. We have proved that if a table contains a defect of this kind, our algorithm will find it.

We detect the following class of defects. Suppose a simple decision table T and variables $p_1, \ldots, p_k$ are given. Suppose also, for the sake of the present discussion, that the table we are given has only one column for each operational procedure. Suppose that there is some logical formula B, not identically true or identically false, such that for every column s of the table, either

- every assignment of values to the variables $p_1, \ldots, p_n$ that is compatible with s satisfies B, or else
- every assignment of values to the variables $p_1, \ldots, p_n$ that satisfies B is compatible with s.



structure1

1 variable structural analysis result

		[illegible]	[illegible]	[illegible]
[illegible]	TRUE FALSE	[illegible]	[illegible]	[illegible]
fms_speed_mode	econ edit def_econ	econ edit	[illegible]	[illegible]
dap_exists	TRUE FALSE	[illegible]	TRUE	[illegible]
[illegible]	TRUE FALSE	[illegible]	[illegible]	TRUE
speed_change	TRUE FALSE	[illegible]	[illegible]	[illegible]
cas_mode	TRUE FALSE	[illegible]	[illegible]	[illegible]

Figure 4.4: Speed scenarios: results of structural analysis on one variable.

Then T cannot be functional. The point is that though the table has columns that in some sense contain or are contained by B, it has no columns complementary to B. Therefore, one of the following two conditions must pertain.

- All columns contained by B are contained by columns that contain B. Hence T is not exclusive.
- For some column contained by B, the column obtained from it by complementing B is not covered by T. Hence T is not exhaustive.

If we drop the assumption that each operational procedure has only one column, the same idea is valid, but the argument is more complicated.

The user can ask TBell whether a given input table has a structural defect of this kind related to a particular set of variables or whether it has a defect related to any set of variables of a given cardinality. TBell returns a table or set of tables each representing a formula B as described above. For example, if our input table is the table in Figure 4.1 and we ask for information about defects involving one variable, then we get the set of tables in Figure 4.4. They tell us some problems with the input table: the only specific value `dap_exists` and `dap_con` are ever assigned is **true**, never **false**, and that `fcc_asr` occurs only with values **econ** and `edit`, never with the value `def_econ`.

If we asked for structural analysis involving two variables, we would get additional information such as that two variables occur only with the same sign, suggesting that they are actually equivalent conditions and that only one of them should occur in the table. Structural analysis involving more variables yields more complex information.

We recommend that analysis of a table begin with overlap analysis. Once all overlaps have been eliminated, proceed to structural analysis on one variable. Once there are no more problems with single variables, proceed to structural analysis on two variables, etc. Finally, do coverage analysis and make final corrections to the table.

In general, one should correct all structural defects involving n variables before proceeding to structural analysis on $n+1$ variables, because otherwise all n-variable defects will be repeated somehow as $n + 1$-variable defects.

Structural analysis is sufficient to guarantee functionality of an exclusive table, because if a table is not exhaustive then the disjunction of its columns is a formula of the kind that will be found by structural analysis.

4.4 Code Generation

It is easy to efficiently generate code (here, Ada code) from decision tables using the internal representation employed by TBell's logical analyzer. Essentially TBell combines the FDDs representing the engagement criteria of the various operational criteria into one big $\mathcal{P}(\mathsf{OP})$-valued FDD in which each path through the FDD ends in the set of all operational procedures whose engagement criteria contain that path. An FDD is just a big nested case statement, so we translate it into a big nested Ada case statement. When a path ends in a singleton {opp}, return opp. If a path ends in the emptyset, raise the exception Undefined. If a path ends a set with more than one member, raise the exception Ambiguous. Of course, if one generates code from a *functional* decision table, these exceptions will never arise. The point is that we can generate code that implements a table whether or not the table is correct (functional). One might wish to do so if one believed that the bad cases would in fact never arise, or if one wished to prototype a program using an early version of a decision table. Such a prototype will automatically indicate a flaw in the table by raising an exception.

The giant case statement is encapsulated in a function subprogram to which the decision table variables are input as global Ada variables. Code generated from Figure 4.1 is shown in Figure 4.5. TBell also generates declarations and wraps everything up in a compilable Ada package, though we expect that the generated subprogram will be used in a context other than the package generated by TBell.

The enormous-case-statement translation appears to produce near-optimal code implementing a decision table. It may be possible to optimize the size and performance of the code by using heuristics to choose a better order for the decision variables. (Currently the order is the same as the order in which the variables are listed in the decision table.)

We plan to provide an option to generate more readable code. This more readable code will evaluate engagement criteria of individual operational procedures and then either return an operational procedure or raise an exception, as appropriate. The readable code for each engagement criterion will be essentially an Ada version of the English-language documentation described in the next section.

4.5 Generating English-Language Documentation

One of the principal documents describing the Honeywell code that we worked from on our project was the Software Requirements Document (SRD). In what we might describe as relatively formal English, it described engagement criteria and behaviors associated with various classes of operational procedures at various levels of a flight control system. The engagement criteria in the SRD were easy to translate (or, more properly, transcribe) into code or into a formal specification in a language like PVS or Larch (modulo a few ambiguities and cases where

```
function dp_spd_scn return spd_scn is
begin
  if fcc_asr then
    return approach;
  else
    case fms_speed_mode is
      when econ =>
        if dap_exists then
          if dap_con then
            if speed_change then
              raise Ambiguous;
            else
              return descent_path;
            end if;
          else
            return descent_path;
          end if;
        else
          if dap_con then
              ...
```

Figure 4.5: Generated Ada code.

1. **FCC Approach Speed Request.**
   FCC Approach Speed Request (fcc_asr) is FALSE.
2. **Descent/Approach Path Status.**
   Any of the following requirements are satisfied.
   (a) Descent/Approach Path Exists (**dap_exists**) is TRUE.
   (b) All of the following requirements are satisfied.
      i. Descent/Approach Path Exists (**dap_exists**) is FALSE.
      ii. Descent/Approach Path Under Construction (**dap_con**) is TRUE.
      iii. Speed Changed (**speed_change**) is FALSE.

Figure 4.6: Generated documentation.

the text seemed to be garbled). Since the SRD text was so formal, it seemed worth attempting to generate this text directly from the decision tables. Figure 4.6 contains an extract from the result.

TBell generates LaTeX source. We plan to add options to generate documentation in Microsoft Word and Word Perfect format.

The logical form of the generated text is identical to that of a partitioned decision table. That is, the variables are partitioned into categories (corresponding to headings like "Descent/Approach Path Status" and "FCC Approach Speed Request"). In the text, the word "any" (resp. "all") signifies a disjunction (resp. conjunction) of the list of conditions following it. Thus, the text is a disjunction of conjunctions of formulas, each formula itself a disjunctive normal form containing variables of only one category.

In order to generate this English text, the user must supply the categories as well as the phrases describing the variables. If the user does not supply categories, then none will be used. If the user does not supply descriptive phrases for variables, then the variable name will be used (the same name as in the table, included in round brackets if there is a descriptive phrase).

Chapter 5

Research Directions

The research described in this paper has been going on for only a short time and there is much yet to be done. Here we will list some of the more substantial research topics that remain.

5.1 More General Queries about Decision Tables

Our logical analysis algorithms are adequate to test any desired property of a decision table, not just coverage properties. One might want to ask questions that would validate the contents of a decision table, such as: "Operational procedure opp will not be selected if condition A holds." The problem is to design a good language for asking such questions.

5.2 Accounting for Illegal Scenarios

Our practical work with Honeywell showed that in many cases certain scenarios (combinations of values of input variables) are considered to be impossible. For example, they might be physically impossible. Alternatively a certain decision table T' may be part of the definition of the behavior of a certain operational procedure opp occurring in a decision table T; hence inputs to T' always satisfy the engagement criterion of opp, defined by T.

The main deficiency in the use of decision tables as outlined in this report is that the configurations considered illegal are nowhere clearly documented. We propose to remedy this deficiency by creating a dummy operational procedure, called *Illegal*. All illegal configurations must be recorded as scenarios belonging to operational procedure *Illegal*. Engagement criteria of all legitimate operational procedures are relative to the legal scenarios (those complementary to the engagement criterion of Illegal). All functions of TBell should generalize straightforwardly.

5.3 Behaviors

Our discussion in this report covers only the selection half of a decision table, which says which operational procedure will be engaged in a given situation. The other half of a decision table describes what effect each operational procedure will have on the state of the system and what outputs it will produce. Part of the behavior may be described by further decision tables. We would like to support the behaviors half of the decision table as well. Exclusiveness and exhaustiveness testing are not directly relevant for behaviors, though they are relevant for any dependent tables used to define parts of behaviors; code and English document generation are relevant. The most basic way to support behaviors is simply to include them in the table and to generate code and documentation.

5.4 Systems of Interrelated Decision Tables

In order to offer a higher level of support to the behaviors part of a decision table, we would like to take account of the fact that a decision table that describes part of the behavior associated with an operational procedure of a parent table will receive only inputs that satisfy the engagement criterion of that operational procedure. Thus any input configurations not satisfying that engagement criterion can be justified by context as illegal configurations of the dependent decision table.

These considerations call for a system that deals not with single decision tables but with systems of interrelated decision tables, the set of illegal configurations of a dependent symbol table being justified according to context.

Going further in this direction, one can imagine a formalism for successive uses of decision tables with a state change in between. This idea points toward either a predicate transformation system (as in [GMP90]) involving decision tables or, if the whole system can be regarded as finite-state, integration with a model-checking system (such as [CLM89]). Integrating with a model-checking system would be especially natural since such systems use methods similar to ours, namely finite logic and FDDs. In either case, ability to justify declaring certain configurations illegal would be enhanced by the ability to prove invariants of the system containing our decision tables (such as the invariant that illegal configurations never occur in the system).

5.5 Algorithms for Partitioning Decision Tables

In order to construct a partitioned decision table or to generate English-language documentation for a simple decision table, the user must supply a partition of variables into categories that give a reasonable structure to the information in the table. We would like to find a way to generate such partitions automatically.

5.6 Methodology for Developing Correct Decision Tables

All of the methods relating to correctness discussed in this report deal with detecting errors in decision tables. In theory it would be better to develop decision tables in an orderly fashion without the errors. We might imagine a method starting with a small table having a small number of variables and operational procedures and gradually refining the table into a complex one by splitting variables and operational procedures. It is hard, however, to see how such a procedure would be adequate to develop the complex decision tables we encounter in practise. Perhaps a better method would be to develop tables as compositions of simpler tables (i.e. sequences of simpler decisions). A third possibility, based on structural analysis, would be to add columns to a table one by one, maintaining exclusiveness, until all structural flaws were eliminated.

Bibliography

[Ake78] S. B. Akers. Binary decision diagrams. *IEEE Transactions on Computers*, C-27:509–516, June 1978.

[BFG+93] I. Bahar, E. Frohm, C. Gaona, G. D. Hachtel, E. Macii, A. Pardo, and F. Somenzi. Algebraic decision diagrams and their applications. In *ICCAD '93*, 1993.

[Bry86] R. E. Bryant. Graph-based algorithms for boolean function manipulation. *IEEE Transactions on Computers*, pages 677–691, August 1986.

[CFM+93] E. M. Clarke, M. Fujita, P. C. McGeer, K. McMillan, J. C.-Y. Yang, and X. Zhao. Multi-terminal binary decision diagrams: An efficient data structure for matrix representation. Technical report, Computer Science Department, Carnegie-Mellon University, February 1993.

[CLM89] E. M. Clarke, D. E. Long, and K. L. McMillan. Compositional model checking. In *Proceedings of the Fourth Annual Symposium on Logic in Computer Science.* IEEE Computer Society Press, June 1989.

[GMP90] David Guaspari, Carla Marceau, and Wolfgang Polak. Formal verification of Ada programs. *IEEE Transactions on Software Engineering*, 16:1058–1075, September 1990.

[Hon94] Honeywell, Inc. *Apparatus and Method for Controlling the Vertical Profile of an Aircraft*, Aug. 16 1994. United States Patent #5,337,982.

[Lee59] C. Y. Lee. Representation of switching circuits by binary-decision programs. *Bell Systems Technical Journal*, 38:985–999, July 1959.

[Mar93] Carla Marceau. *Penelope Reference Manual.* ORA, December 1993.

[McC94] W. W. McCune. Otter 3.0 reference manual and guide. Technical Report ANL-94/6, Argonne National Laboratory, 9700 South Cass Ave., Argonne IL 60439-4801 USA, January 1994.

[Rud93] Richard Rudell. Dynamic variable ordering for ordered binary decision diagrams. In *ICCAD '93*, 1993.

[She94a] Lance Sherry. The operational procedure information model. Technical report, Honeywell, Inc., 1994.

[She94b] Lance Sherry. A structured approach to requirements specification for software-based systems using operational procedures. In *IEEE Digital Avionics Systems Conference*, 1994.

[SOR93a] N. Shankar, S. Owre, and J. M. Rushby. *A Tutorial on Specification and Verification Using PVS.* Computer Science Laboratory, SRI International, Menlo Park CA 94025 USA, March 1993.

[SOR93b] N. Shankar, S. Owre, and J. M. Rushby. A tutorial on specification and verification using PVS. Technical report, SRI International, Menlo Park CA 94025 USA, March 1993.

Appendix A

PVS Formalization

The following is the PVS file containing a formalization in PVS of the engagement criteria given in [Hon94] Section 10.2.5.5 (speed scenario selection for the Descent Path operational procedure). The theory is that expressed by the table in Figure 4.1 (except that here we use the name other instead of def_econ).

The two theorems exhaustive and exclusive verify that the missing columns and the overlaps pointed out by TBell are in fact correct. The PVS proof of each was simply

```
(then* (auto-rewrite-theory "VG")(do-rewrite)(iff)(bddsimp))
```

It was our practise while developing TBell to check its results by doing these kinds of proofs in PVS.

```
VG : THEORY
  BEGIN

  speed_mode : TYPE  =  {econ, edit, other}
  fcc_asr : boolean
  dap_exists : boolean
  dap_con : boolean
  speed_change : boolean
  fms_speed_mode : speed_mode
  cas_mode : boolean
  descent_path : boolean  =  ¬ fcc_asr  ∧  (dap_exists  ∨  (dap_con  ∧ ¬ speed_change))

  approach : boolean  =  fcc_asr

  econ_path_const_cas :
    boolean  =  ¬ fcc_asr  ∧  (fms_speed_mode  =  econ)  ∧  dap_con  ∧  speed_change  ∧  cas_mode

  econ_path_const_mach :
    boolean  =  ¬ fcc_asr  ∧  (fms_speed_mode  =  econ)  ∧  dap_con  ∧  speed_change  ∧  ¬ cas_mode
```

```
edit_path_const_cas :
 boolean  = ¬ fcc_asr  ∧ (fms_speed_mode  = edit) ∧ dap_con  ∧ speed_change  ∧ cas_mode

edit_path_const_mach :
 boolean  = ¬ fcc_asr  ∧ (fms_speed_mode  = edit) ∧ dap_con  ∧ speed_change  ∧ ¬ cas_mode

exhaustive :
 THEOREM
  ¬
   (descent_path
      ∨ approach
       ∨ econ_path_const_cas
        ∨ econ_path_const_mach  ∨ edit_path_const_cas  ∨ edit_path_const_mach)
    ⇔
   ((fcc_asr  = FALSE
      ∧ (fms_speed_mode  = econ  ∨ fms_speed_mode  = edit)
       ∧ dap_exists  = FALSE  ∧ dap_con  = FALSE)
     ∨
     (fcc_asr  = FALSE
       ∧ fms_speed_mode  = other
        ∧ dap_exists  = FALSE  ∧ dap_con  = TRUE  ∧ speed_change  = TRUE)
      ∨
      (fcc_asr  = FALSE
        ∧ fms_speed_mode  = other  ∧ dap_exists  = FALSE  ∧ dap_con  = FALSE))

exclusive :
 THEOREM
  (descent_path  ∧ approach) = FALSE
    ∧ (descent_path  ∧ econ_path_const_cas)
     =
    (fcc_asr  = FALSE
      ∧ fms_speed_mode  = econ
       ∧ dap_exists  = TRUE
        ∧ dap_con  = TRUE  ∧ speed_change  = TRUE  ∧ cas_mode  = TRUE)
     ∧ (descent_path  ∧ econ_path_const_mach)
      =
     (fcc_asr  = FALSE
       ∧ fms_speed_mode  = econ
        ∧ dap_exists  = TRUE
         ∧ dap_con  = TRUE  ∧ speed_change  = TRUE  ∧ cas_mode  = FALSE)
      ∧ (descent_path  ∧ edit_path_const_cas)
       =
      (fcc_asr  = FALSE
        ∧ fms_speed_mode  = edit
         ∧ dap_exists  = TRUE
          ∧ dap_con  = TRUE  ∧ speed_change  = TRUE  ∧ cas_mode  = TRUE)
       ∧ (descent_path  ∧ edit_path_const_mach)
        =
        (fcc_asr  = FALSE
           ∧ fms_speed_mode  = edit
             ∧ dap_exists  = TRUE
               ∧ dap_con  = TRUE  ∧ speed_change  = TRUE  ∧ cas_mode  = FALSE)
         ∧
        (approach
           ∧
          (econ_path_const_cas
              ∨ econ_path_const_mach
                ∨ edit_path_const_cas  ∨ edit_path_const_mach))
          = FALSE
          ∧
         (econ_path_const_cas
             ∧
           (econ_path_const_mach  ∨ edit_path_const_cas  ∨ edit_path_const_mach))
            = FALSE
           ∧
          (econ_path_const_mach  ∧ (edit_path_const_cas  ∨ edit_path_const_mach))
              = FALSE
              ∧ (edit_path_const_cas  ∧ edit_path_const_mach)  = FALSE

  END VG
```

Appendix B

Generated Code

The following code was generated by TBell from the table in Figure 4.1 (again with **def_econ** changed to **other**).

Several things about the code should be noted.

- TBell does not prettyprint the generated code at all. The indentation in code below was obtained by running it through Penelope [Mar93].
- This particular example of code compiled without modification. That is not generally the case, for the following reasons.
  - A decision table variable does not have to be an Ada variable (see, for example, Figure 2.2). To generate code, decision table variables should indeed be Ada expressions. If a decision table variable is a complex Ada expression, like

    ```
    a + b >= c
    ```

    then the generated code will contain an improper declaration

    ```
    a + b >= c :  boolean;
    ```

    but will not contain any declarations of the variables `a`, `b`, and `c`. The improper declaration of an expression has to be removed and the variable declarations have to be added by hand.
  - The function `compare` may need to be declared differently or more than once with different types if it is used to compare other than integer quantities. The purpose of `compare` is do to comparisons when an answer of the form "greater," "less," or "equal" is required.

For the following reasons, however, it does not seem important to try to generate declarations properly through TBell.

  - Doing so would not save any labor or improve assurance that the code is correct, since essentially the same information in a similar form would have to be entered via TBell.
  - Since the inputs to the decision routine are passed via the variables, those variables have to receive values somewhere, probably somewhere else. Consequently their declarations will in fact have to be put somewhere else as well, so there is really no way to avoid manually modifying the declarations.

In a word, the value of generating code from TBell is in the subprogram that implements the table, rather than in the packaging around it.

The code as generated here essentially implements a decision tree that corresponds to the table implemented by the code. Scenarios for which the table does not indicate any operational procedure raise the exception `Undefined`; scenarios for which the table indicates more than one operational procedure raise the exception `Ambiguous`.

Implementing the code as a decision tree gives near-optimal execution speed, but is not very readable. If for some reason the code is to be modified or checked by hand, a way of generating more structured code would be indicated. We plan to provide an option to generate code structured like the generated documentation shown in Appendix C. We have not yet implemented such an option, but the fact that we can generate the documentation shows that there are no unsolved technical problems involved.

In principle, we believe that the proper approach is to modify the decision table instead of the code (except for the enveloping declarations) and generate the most efficient code.

Here, then, is the Ada code generated by TBell from the table in Figure 4.1.

```
-- Warning! ***************************************** !
--
-- 1) The package TEST_PAK is only an example: you will probably want to
-- put the subprogram in your own package somewhere.
--
-- 2) The type declarations are syntactically correct (unless
-- you used "others" to stand for unspecified members of an
-- enumeration type), but you may want them to appear in some other
-- package.
--
-- 3) The object declarations may not be syntactically correct.  In
-- particular they may be of the form either
--
-- (*)      expression : type;
-- or
-- (**)      identifier_1.identifier_2. ... . identifier_n : type;
-- instead of
-- identifier : type;
-- Nevertheless, they do correctly indicate the type of the given
-- expression.
-- Some object declarations should be in another package.  Those of
-- form (**) must be in another package, where they must take the form
-- identifier_n : type;
-- "Declarations" of expressions (form (*)) will have to be broken up
-- into declarations of the component variables.
-- End Warning ***************************************************

package EXAMPLE_PAK is
  type spd_scn is (descent_path, approach, econ_path_const_cas,
  econ_path_const_mach, edit_path_const_cas, edit_path_const_mach);
  type speed_mode is (econ, edit, other);
  fcc_asr : boolean;
  fms_speed_mode : speed_mode;
  dap_exists : boolean;
  dap_con : boolean;
  speed_change : boolean;
  cas_mode : boolean;
  Undefined, Ambiguous : exception;
  function dp_spd_scn return spd_scn;
end EXAMPLE_PAK;

package body EXAMPLE_PAK is
  -- WARNING: you may need to declare *compare* with other input types
  type COMP is (LT, EQ, GT);
  function compare(x, y : in integer) return COMP

  is

  begin
    if (x<y) then
      return LT;
    elsif (x=y) then
      return EQ;
    else
      return GT;
    end if;
  end compare;
  function dp_spd_scn return spd_scn
```

```
is

begin
  if fcc_asr then
    return approach;
  else
    case fms_speed_mode is
      when econ =>
        if dap_exists then
          if dap_con then
            if speed_change then
              raise Ambiguous;
            else
              return descent_path;
            end if;
          else
            return descent_path;
          end if;
        else
          if dap_con then
            if speed_change then
              if cas_mode then
                return econ_path_const_cas;
              else
                return econ_path_const_mach;
              end if;
            else
              return descent_path;
            end if;
          else
            raise Undefined;
          end if;
        end if;
      when edit =>
        if dap_exists then
          if dap_con then
            if speed_change then
              raise Ambiguous;
            else
              return descent_path;
            end if;
            else
              return descent_path;
            end if;
          else
            if dap_con then
              if speed_change then
                if cas_mode then
                  return edit_path_const_cas;
                else
                  return edit_path_const_mach;
                end if;
              else
                return descent_path;
              end if;
            else
              raise Undefined;
            end if;
          end if;
        when other =>
          if dap_exists then
            return descent_path;
          else
            if dap_con then
              if speed_change then
                raise Undefined;
              else
                return descent_path;
              end if;
            else
              raise Undefined;
            end if;
          end if;
      end case;
    end if;
  end dp_spd_scn;
end EXAMPLE_PAK;
```

Appendix C

Documentation Generated by TBell

The following is the documentation for the table in Figure 4.1 as generated by TBell's SML module. Text generation is not currently available through TBell's window interface.

The LaTeX source of the text below is exactly as generated by TBell, except for adding a "*" to keep the first heading from affecting the section numbering of this report.

In order to generate the following documentation, it is necessary to supply the following information in addition to the table.

- A descriptive phrase for each variable and each operational procedure. In the text, this phrase always precedes the (parenthesized) name as used in the table.
- The variables must be divided into categories, to give some structure to the description and to make it more compact. The names of the categories are used as headings, here "FCC Approach Speed Request," "FMS Speed Mode," "Descent/Approach Path Status" and "CAS Mode."

Given this information, TBell automatically reorganizes the table and generates LaTeX source for the documentation.

The style of the documentation generated by TBell closely imitates the documentation provided by Honeywell, which is a kind of quasi-formal English, very suitable for specifying code. Automatically generating documentation from a table has the advantage that the correctness of the documentation (relative to the table) is guaranteed and that the usage of terms can be made absolutely consistent.

The main problem with the Honeywell documentation was understanding what the possible values of the parameters were. Some sort of declaration section should be added to the generated documentation.

C.1 Vertical Guidance Speed Targets (Descent Path)

C.1.1 Descent Path (descent_path)

Engagement Criteria

All of the following requirements are satisfied.

1. **FCC Approach Speed Request.**

   FCC Approach Speed Request (fcc_asr) is FALSE.

2. **Descent/Approach Path Status.**

   Any of the following requirements are satisfied.

   (a) Descent/Approach Path Exists (dap_exists) is TRUE.

   (b) All of the following requirements are satisfied.

      i. Descent/Approach Path Exists (dap_exists) is FALSE.

      ii. Descent/Approach Path Under Construction (dap_con) is TRUE.

      iii. Speed Changed (speed_change) is FALSE.

C.1.2 Approach (approach)

Engagement Criteria

FCC Approach Speed Request (fcc_asr) is TRUE.

C.1.3 Econ Path Under Construction—CAS Regime (econ_path_const_cas)

Engagement Criteria

All of the following requirements are satisfied.

1. **FCC Approach Speed Request.**

   FCC Approach Speed Request (fcc_asr) is FALSE.

2. **FMS Speed Mode.**

   FMS Speed Mode (fms_speed_mode) is econ.

3. **Descent/Approach Path Status.**

   All of the following requirements are satisfied.

   (a) Descent/Approach Path Under Construction (dap_con) is TRUE.

   (b) Speed Changed (speed_change) is TRUE.

4. **CAS Mode.**

   CAS Mode (cas_mode) is TRUE.

C.1.4 Econ Path Under Construction—Mach Regime (econ_path_const_mach)

Engagement Criteria

All of the following requirements are satisfied.

1. **FCC Approach Speed Request.**
   FCC Approach Speed Request (fcc_asr) is FALSE.
2. **FMS Speed Mode.**
   FMS Speed Mode (fms_speed_mode) is econ.
3. **Descent/Approach Path Status.**
   All of the following requirements are satisfied.
   (a) Descent/Approach Path Under Construction (dap_con) is TRUE.
   (b) Speed Changed (speed_change) is TRUE.
4. **CAS Mode.**
   CAS Mode (cas_mode) is FALSE.

C.1.5 Edit Path Under Construction—CAS Regime (edit_path_const_cas)

Engagement Criteria

All of the following requirements are satisfied.

1. **FCC Approach Speed Request.**
   FCC Approach Speed Request (fcc_asr) is FALSE.
2. **FMS Speed Mode.**
   FMS Speed Mode (fms_speed_mode) is edit.
3. **Descent/Approach Path Status.**
   All of the following requirements are satisfied.
   (a) Descent/Approach Path Under Construction (dap_con) is TRUE.
   (b) Speed Changed (speed_change) is TRUE.
4. **CAS Mode.**
   CAS Mode (cas_mode) is TRUE.

C.1.6 Edit Path Under Construction—Mach Regime (edit_path_const_mach)

Engagement Criteria

All of the following requirements are satisfied.

1. **FCC Approach Speed Request.**

   FCC Approach Speed Request (fcc_asr) is FALSE.

2. **FMS Speed Mode.**

   FMS Speed Mode (fms_speed_mode) is edit.

3. **Descent/Approach Path Status.**

   All of the following requirements are satisfied.

   (a) Descent/Approach Path Under Construction (dap_con) is TRUE.

   (b) Speed Changed (speed_change) is TRUE.

4. **CAS Mode.**

   CAS Mode (cas_mode) is FALSE.

Appendix D

Verification of Generated Code Using Penelope

The following is the complete Penelope verification of the code generated by TBell for the table in Figure 4.1. This verification constitutes a complete check that this particular piece of code is a correct implementation of the table. The specification is a transcription into Penelope's specification language, Larch/Ada, of the PVS formalization given in Appendix A.

```
--! Verification status: Verified
--| Larch
VG: trait
  sort SpeedMode is (econ, edit, other)
  sort OpProc is (descent_path, approach, econ_path_const_cas,
  econ_path_const_mach, edit_path_const_cas, edit_path_const_mach)
  introduces
    proper: OpProc -> Bool
    eng_crit: Bool, SpeedMode, Bool, Bool, Bool, Bool, OpProc -> Bool
    undefined, ambiguous: Bool,
        SpeedMode, Bool, Bool, Bool, Bool, OpProc -> Bool

  asserts
    forall fcc_asr, dap_exists, dap_con, speed_change, cas_mode:Bool,
    fms_speed_mode:SpeedMode, opp:OpProc
      proper_def: (proper(opp)
        =
          ((opp=descent_path)
           or
            ((opp=approach)
             or
        (((opp=econ_path_const_cas) or (opp=econ_path_const_mach))
         or
          ((opp=edit_path_const_cas) or (opp=edit_path_const_mach))))))
eng_crit_def: (eng_crit(fcc_asr, fms_speed_mode, dap_exists, dap_con,
      speed_change, cas_mode, opp)
  =
    (if (opp=descent_path)
      then ((not fcc_asr)
         and
          (dap_exists or (dap_con and (not speed_change))))
      else (if (opp=approach)
          then fcc_asr
          else (if (opp=econ_path_const_cas)
              then ((not fcc_asr)
                 and
                  ((fms_speed_mode=econ)
                   and
                    (dap_con and (speed_change and cas_mode))))
              else (if (opp=econ_path_const_mach)
                  then ((not fcc_asr)
                     and
                      ((fms_speed_mode=econ)
                       and
                        (dap_con
                         and
                          (speed_change and (not cas_mode)))))
                  else (if (opp=edit_path_const_cas)
                      then ((not fcc_asr)
                         and
                          ((fms_speed_mode=edit)
                           and
                            (dap_con and (speed_change and cas_mode))))
                      else (if (opp=edit_path_const_mach)
                          then ((not fcc_asr)
                             and
                              ((fms_speed_mode=edit)
                               and
                                (dap_con
                                 and
                                  (speed_change and (not cas_mode)))))
                          else false)))))))
 descent_path: (eng_crit(fcc_asr, fms_speed_mode, dap_exists, dap_con,
   speed_change, cas_mode, descent_path)
 =
  ((not fcc_asr) and (dap_exists or (dap_con and (not speed_change)))))
appr: (eng_crit(fcc_asr, fms_speed_mode, dap_exists, dap_con,
   speed_change, cas_mode, approach)=fcc_asr)
econ_path_const_cas: (eng_crit(fcc_asr, fms_speed_mode, dap_exists,
   dap_con, speed_change, cas_mode, econ_path_const_cas)
 =
  ((not fcc_asr)
   and
   ((fms_speed_mode=econ)
    and
    (dap_con and (speed_change and cas_mode)))))
econ_path_const_mach: (eng_crit(fcc_asr, fms_speed_mode, dap_exists,
   dap_con, speed_change, cas_mode, econ_path_const_mach)
 =
  ((not fcc_asr)
   and
   ((fms_speed_mode=econ)
    and
    (dap_con and (speed_change and (not cas_mode))))))
edit_path_const_cas: (eng_crit(fcc_asr, fms_speed_mode, dap_exists,
   dap_con, speed_change, cas_mode, edit_path_const_cas)
 =
  ((not fcc_asr)
   and
   ((fms_speed_mode=edit)
    and
    (dap_con and (speed_change and cas_mode)))))
edit_path_const_mach: (eng_crit(fcc_asr, fms_speed_mode, dap_exists,
   dap_con, speed_change, cas_mode, edit_path_const_mach)
 =
  ((not fcc_asr)
   and
   ((fms_speed_mode=edit)
    and
    (dap_con and (speed_change and (not cas_mode))))))
undefined: (undefined(fcc_asr, fms_speed_mode, dap_exists, dap_con,
   speed_change, cas_mode, descent_path)
 =
  (ømf op:OpProc::
   (not eng_crit(fcc_asr, fms_speed_mode, dap_exists, dap_con,
             speed_change, cas_mode, op))))
      ambiguous: (ambiguous(fcc_asr, fms_speed_mode, dap_exists, dap_con,
            speed_change, cas_mode, descent_path)
        =
          (øme op1, op2:OpProc::
            ((op1/=op2)
             and
              (eng_crit(fcc_asr, fms_speed_mode, dap_exists, dap_con,
                  speed_change, cas_mode, op1)
               and
                eng_crit(fcc_asr, fms_speed_mode, dap_exists, dap_con,
                  speed_change, cas_mode, op2)))))


--| end Larch


--| with trait VG ;
package EXAMPLE_PAK is
  type OP_PROC_TYPE is (descent_path, approach, econ_path_const_cas,
  econ_path_const_mach, edit_path_const_cas, edit_path_const_mach);
  type fms_speed_mode_TYPE is (econ, edit, other);
  fcc_asr : BOOLEAN;
  fms_speed_mode : fms_speed_mode_TYPE;
  dap_exists : BOOLEAN;
  dap_con : BOOLEAN;
  speed_change : BOOLEAN;
  cas_mode : BOOLEAN;
  Undefined, Ambiguous : exception;
  function dp_spd_scn return OP_PROC_TYPE;
  --| where
  --|      global fcc_asr, fms_speed_mode, dap_exists, dap_con, speed_change,
            cas_mode : in ;
  --|      return opp such that eng_crit(fcc_asr, fms_speed_mode, dap_exists,
              dap_con, speed_change, cas_mode, opp);
  --|      raise
            Ambiguous
            <=> in
            ambiguous(fcc_asr, fms_speed_mode, dap_exists, dap_con,
                speed_change, cas_mode, descent_path);
  --|      raise
            Undefined
            <=> in
            undefined(fcc_asr, fms_speed_mode, dap_exists, dap_con,
                speed_change, cas_mode, descent_path);
  --| end where;

end EXAMPLE_PAK;

package body EXAMPLE_PAK is
  -- WARNING: you may need to declare *compare* with other input types
  function dp_spd_scn return OP_PROC_TYPE
    --| where * * *
    --|      global fcc_asr, fms_speed_mode, dap_exists, dap_con,
              speed_change, cas_mode : in ;
    --|      return opp such that eng_crit(fcc_asr, fms_speed_mode,
                dap_exists, dap_con, speed_change, cas_mode, opp);
    --|      raise
              Ambiguous
              <=> in
              ambiguous(fcc_asr, fms_speed_mode, dap_exists, dap_con,
                  speed_change, cas_mode, descent_path);
    --|      raise
              Undefined
              <=> in
              undefined(fcc_asr, fms_speed_mode, dap_exists, dap_con,
                  speed_change, cas_mode, descent_path);
    --| end where;
    --! VC Status: proved
    --! BY instantiation of ambiguous in trait VG as rewrite rule
    --! BY instantiation of undefined in trait VG as rewrite rule
    --! BY instantiation of proper_def in trait VG as rewrite rule
    --! BY instantiation of eng_crit_def in trait VG as rewrite rule
    --! BY limited simplification, limited simplification
    --! BY synthesis of IF,
    --!    BY prenex simplification, simplification
    --!    BY synthesis of EXISTS exhibiting approach
    --!    BY limited simplification
    --!    BY synthesis of TRUE
    --! AND
    --!    BY synthesis of AND
    --!       BY synthesis of FORALL/IMPLIES
    --!       BY right substitution, in 2 then thinning,
    --!       BY limited simplification
--!       BY synthesis of IF,
--!          BY synthesis of IF,
--!             BY synthesis of AND
--!                BY cases, using cas_mode, then rewriting
--!                CASE TRUE
--!                   BY synthesis of EXISTS exhibiting descent_path,
econ_path_const_cas
--!                   BY limited simplification
--!                   BY synthesis of TRUE
--!                CASE FALSE
--!                   BY synthesis of EXISTS exhibiting descent_path,
econ_path_const_mach
--!                   BY limited simplification
--!                   BY synthesis of TRUE

--!                BY prenex simplification
--!                BY synthesis of EXISTS exhibiting descent_path
--!                BY limited simplification
--!                BY synthesis of TRUE

--!             □
--!          AND
--!             BY prenex simplification
--!             BY synthesis of EXISTS exhibiting descent_path
--!             BY limited simplification
--!             BY synthesis of TRUE
--!       AND
--!          BY synthesis of FORALL/IMPLIES
--!          BY synthesis of IF,
--!             BY synthesis of IF,
--!                BY prenex simplification
--!                BY synthesis of EXISTS exhibiting econ_path_const_cas
--!                BY limited simplification
--!                BY synthesis of TRUE
--!             AND
--!                BY prenex simplification
--!                BY synthesis of EXISTS exhibiting econ_path_const_mach
--!                BY limited simplification
--!                BY synthesis of TRUE
--!          AND
--!             BY prenex simplification
--!             BY synthesis of EXISTS exhibiting descent_path
--!             BY limited simplification
--!             BY synthesis of TRUE

--!       BY synthesis of FORALL/IMPLIES
--!       BY right substitution, in 2 then thinning,
--!       BY limited simplification
--!       BY synthesis of IF,
--!          BY synthesis of IF,
--!             BY synthesis of AND
--!                BY cases, using cas_mode, then rewriting
--!                CASE TRUE
--!                   BY synthesis of EXISTS exhibiting descent_path,
edit_path_const_cas
--!                   BY limited simplification
--!                   BY synthesis of TRUE
--!                CASE FALSE
--!                   BY synthesis of EXISTS exhibiting descent_path,
edit_path_const_mach
--!                   BY limited simplification
--!                   BY synthesis of TRUE

--!                BY prenex simplification
--!                BY synthesis of EXISTS exhibiting descent_path
--!                BY limited simplification
--!                BY synthesis of TRUE

--!             □
--!          AND
--!             BY prenex simplification
--!             BY synthesis of EXISTS exhibiting descent_path
--!             BY limited simplification
--!             BY synthesis of TRUE
--!       AND
--!          BY synthesis of FORALL/IMPLIES
--!          BY synthesis of IF,
--!             BY synthesis of IF,
--!                BY prenex simplification
--!                BY synthesis of EXISTS exhibiting edit_path_const_cas
--!                BY limited simplification
--!                BY synthesis of TRUE
--!             AND
--!                BY prenex simplification
```

```
  --!                  BY synthesis of EXISTS exhibiting edit_path_const_mach
  --!                  BY limited simplification
  --!                  BY synthesis of TRUE
  --!            AND
  --!               BY prenex simplification
  --!               BY synthesis of EXISTS exhibiting descent_path
  --!               BY limited simplification
  --!               BY synthesis of TRUE

  --!         BY synthesis of FORALL/IMPLIES
  --!         BY right substitution, in 2 then thinning,
  --!         BY limited simplification
  --!         BY synthesis of IF,
  --!            BY prenex simplification
  --!            BY synthesis of EXISTS exhibiting descent_path
  --!            BY limited simplification
  --!            BY synthesis of TRUE
  --!         AND
  --!            BY prenex simplification
  --!            BY synthesis of EXISTS exhibiting descent_path
  --!            BY limited simplification
  --!            BY synthesis of TRUE

  --!     □

is


begin
  if fcc_asr then
    return approach;
  else
    case fms_speed_mode is
      when econ =>
        if dap_exists then
          if dap_con then
            if speed_change then
              raise Ambiguous;
            else
              return descent_path;
            end if;
    else
      return descent_path;
    end if;
  else
    if dap_con then
      if speed_change then
        if cas_mode then
          return econ_path_const_cas;
        else
          return econ_path_const_mach;
        end if;
      else
        return descent_path;
      end if;
    else
      raise Undefined;
    end if;
  end if;
when edit =>
  if dap_exists then
    if dap_con then
      if speed_change then
        raise Ambiguous;
      else
        return descent_path;
      end if;
    else
      return descent_path;
    end if;
  else
    if dap_con then
      if speed_change then
        if cas_mode then
          return edit_path_const_cas;
        else
          return edit_path_const_mach;
        end if;
      else
        return descent_path;
      end if;
    else
      raise Undefined;
```

```
            end if;
          end if;
        when other =>
          if dap_exists then
            return descent_path;
          else
            if dap_con then
              if speed_change then
                raise Undefined;
              else
                return descent_path;
              end if;
            else
              raise Undefined;
            end if;
          end if;
      end case;
    end if;
  end dp_spd_scn;
end EXAMPLE_PAK;
```

Appendix E

A Survey of Other Decision Table Tools

ORA has conducted a survey of existing decision table tools. We have not actually used any of these tools. The descriptions below are compiled from reviews and marketing information.

All the tools surveyed appear to do basic coverage analysis (exclusive and exhaustiveness checks), and a number of them appear to have fancier approaches to generating code than TBell has. A number of them simplify decision tables. In the context of TBell, that simplification would be done by converting tables into the internal FDD representation and then back into tables, a trivial operation. Some of the descriptions mention "flowcharts," apparently meaning to express the idea of a decision diagram.

None of the other tools appear to do anything like TBell's structural analysis or to generate English-language documentation. None seem to support anything like the partitioned decision tables we have suggested. Some are components of larger program development systems.

Deva and Deva-Pro (Binary Triangles, La Mesa, CA)

DEVA-PRO 4.0 is a decision table evaluation program for strategic logic design and development. It allows users to build huge decision tables and to check them for completeness and contradictions. The program also measures the amount of tight logic in the decision table and standardizes the decision table. It also creates two different tight flowchart representations of the logic in the decision table, one of which is easy to turn into source code. The software enables users to simplify the decision table, which means that they can automatically find the smallest tight flowchart that represents the logic in their decision table. It is very useful to be able to simplify the table's logic, especially at the abstract design stages, and thereby eliminate the subtle, hard-to-find bugs. The program is protected by Rainbow Technologies' hardware key. 200K RAM required. Price: $369.95.

Logic Gem (Logic Technologies, Inc.)

Logic GEM 1.0 Programmer's edition is primarily a decision-table editor and logic interpreter for the IBM PC, PS/2 or compatible with 640K of RAM and DOS 2.0 or later. It can also generate code in C, BASIC, dBASE III Plus, FORTRAN, and PASCAL. It is best at disclosing patterns in code. The user interface is somewhat rough and its menu structure somewhat illogical. Logic GEM is a good program for programmers who can use decision-tables, or for those wanting to learn. Price: $198.

REFINE/COBOL 1.0 PC Version (Reasoning Systems, Inc., Palo Alto, CA)

REFINE/COBOL is an interactive, graphical re-engineering tool that allows programmers to assess, redocument, quality-check, and convert existing COBOL applications. REFINE/COBOL extracts business rules based on decision tables. It modularizes large COBOL applications into manageable programs creating CALLs, USING parameters, ENTRYs, and linkage and working storage sections. The system performs program slicing to determine data dependencies for change impact analysis, rationalizes data names across all programs in a JCL job, and reformats COBOL source code. The system generates structure charts, set-use tables, flow charts, and physical data model and data-flow diagrams. The system displays reports online graphically and/or textually, and prints them on PostScript-compatible printers. REFINE/COBOL also provides an interactive code that links reports to code.

Syslog Automation Methodology - SAM (Syslog, Montreal, Quebec)

SAM is an automated software design and development environment that is based on the relational decomposition of software into primitive components. The SAM II logic verifier analyzes the consistency and completeness of decision logic. The dependencies that frequently exist among conditions are taken into account.

Logic Gem (Sterling Castle Software, Marina Del Ray, CA)

Logic Gem is a logic CASE tools program from. This package offers a variety of options for generating program code from tabular-format rules and a well-focused set of powerful analytic aids.

Logic Gem is a program development tool that allows the user to analyze program logic by means of decision tables. A decision table is defined as a matrix associating a series of conditions with a set of actions. Logic Gem, aided by its integral Logic Editor, has capabilities for helping construct decision tables, evaluating the matrix and detecting inconsistencies, detecting and eliminating ambiguities, and generating code in C, Pascal, line-numbered or structured BASIC, FORTRAN, or dBASE III Plus. By transforming IF ... THEN logic into a decision table, Logic Gem can make sure that program logic is complete by verifying that all possible states have corresponding actions. A big advantage of the software is that it reveals unsuspected patterns in a code. While its user interface could be smoother and its menu structure more logical, Logic Gem is valuable for decision table programmers. Price: $99.

Teamwork (Cadre Technologies Inc., Providence, R.I.)

Cadre Technologies Inc.'s Teamwork is a CASE package that operates on workstations using UNIX, VMS, HP Apollo Domain, and OS/2 operating systems and takes advantage of X-Windows for its platforms. It is a multi-user product for a client/server architecture. Teamwork's tools aid real-time CASE development for process specifications, control specifications, decision tables, state/event matrices and process activation tables. Teamwork/RT (for real-time) is a graphics module for Ward-Mellor and Hatley-Pirbhai extensions to data flow diagrams, and provides complete tables of notation for state and process specs. Teamwork/SIM simulates the actions of the model. Price: $29,000.

Turbo CASE and MacBubbles (Structsoft, Inc. and Starsys, Inc.)

Structsoft's Turbo CASE and Starsys's MacBubbles are two CASE packages for the Apple Macintosh. Both have menus with many command shortcuts. Programmers can decompose processes into child process diagrams, decision tables, or minispecs. Data-dictionary entries contain linkage information, and the program can switch context in mid-task. MacBubbles supports only Yourdon/DeMarco dataflow diagrams and has a MacDraw-like interface. Price: Turbo Case—$495; MacBubbles—$780.

DTABL (IBM, System Engineering Services, Armonk, NY)

DTABL maintains a library of limited-entry decision tables that can be compiled into high-quality procedural code. Decision tables can be validated for consistency and completeness. The compiler within DTABL attempts to select the best (or near best) flowchart. In addition to its compiler, DTABL contains an interactive table editor and table verifier that indicates incomplete or inconsistent logic rules. The system also provides questionnaire processing, which interactively guides the user through a series of yes/no questions (conditions) to the proper set of directions (actions). Object code can be generated in APL, PL/1, Cobol, or Algol.

FilePro Plus (The Small Computer Company)

FilePro Plus is a DOS character-based development system available for most UNIX systems and some other platforms. Instead of a programming language, users enter processing commands into DECISION TABLES containing a series of IF ... THEN statements to determine processing in a program. It uses a proprietary database format, and can import and export only DBF files. Although FilePro offers an assortment of tools for screen design, it lacks an easy way of laying out fields. Price: $699.

REPORT DOCUMENTATION PAGE

Form Approved
OMB No. 0704-0188

Public reporting burden for this collection of information is estimated to average 1 hour per response, including the time for reviewing instructions, searching existing data sources, gathering and maintaining the data needed, and completing and reviewing the collection of information. Send comments regarding this burden estimate or any other aspect of this collection of information, including suggestions for reducing this burden, to Washington Headquarters Services, Directorate for Information Operations and Reports, 1215 Jefferson Davis Highway, Suite 1204, Arlington, VA 22202-4302, and to the Office of Management and Budget, Paperwork Reduction Project (0704-0188), Washington, DC 20503.

1. AGENCY USE ONLY ***(Leave blank)***

2. REPORT DATE
November 1994

3. REPORT TYPE AND DATES COVERED
Contractor Report

4. TITLE AND SUBTITLE
TBell: A Mathematical Tool for Analyzing Decision Tables

5. FUNDING NUMBERS
C NAS1-18972

WU 505-64-50-04

6. AUTHOR(S)
D. N. Hoover and Zewei Chen

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)
Odyssey Research Associates, Inc. (ORA)
301 Dates Drive
Ithaca, NY 14850-1313

8. PERFORMING ORGANIZATION REPORT NUMBER
TM-94-0073

9. SPONSORING / MONITORING AGENCY NAME(S) AND ADDRESS(ES)
National Aeronautics and Space Administration
Langley Research Center
Hampton, VA 23681-0001

10. SPONSORING / MONITORING AGENCY REPORT NUMBER
NASA CR-195027

11. SUPPLEMENTARY NOTES
Langley Technical Monitor: C. Michael Holloway
Task 13 Report

12a. DISTRIBUTION / AVAILABILITY STATEMENT
Unclassified-Unlimited

Subject Category 61

12b. DISTRIBUTION CODE

13. ABSTRACT ***(Maximum 200 words)***

This paper describes the development of mathematical theory and software to analyze specifications that are developed using decision tables. A decision table is a tabular format for specifying a complex set of rules that choose one of a number of alternative actions. The report also describes a prototype tool, called TBell, that automates certain types of analysis.

14. SUBJECT TERMS
formal methods, decision tables, technology transfer, binary decision diagrams, formal specification

15. NUMBER OF PAGES
57

16. PRICE CODE
A04

17. SECURITY CLASSIFICATION OF REPORT
Unclassified

18. SECURITY CLASSIFICATION OF THIS PAGE
Unclassified

19. SECURITY CLASSIFICATION OF ABSTRACT

20. LIMITATION OF ABSTRACT