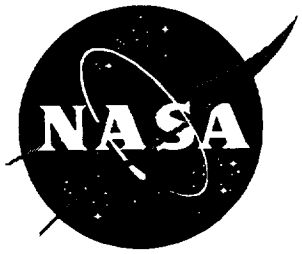
NASA Contractor Report 4723

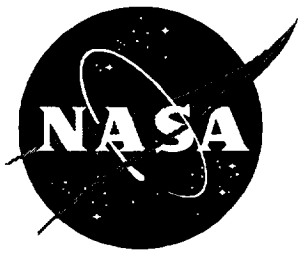


Applications of Formal Methods to Specification and Safety of Avionics Software

D. N. Hoover, David Guaspari, and Polar Humenn

Contract NAS1-20335
Prepared for Langley Research Center

April 1996



NASA Contractor Report 4723

Applications of Formal Methods to Specification and Safety of Avionics Software

D. N. Hoover, David Guaspari, and Polar Humenn
Odyssey Research Associates, Inc. • Ithaca, New York

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

Prepared for Langley Research Center
under Contract NAS1-20335

April 1996

Printed copies available from the following:

NASA Center for AeroSpace Information
800 Elkridge Landing Road
Linthicum Heights, MD 21090-2934
(301) 621-0390

National Technical Information Service (NTIS)
5285 Port Royal Road
Springfield, VA 22161-2171
(703) 487-4650

Abstract

This report treats several topics in applications of formal methods to avionics software development. Most of these topics concern decision tables, an orderly, easy-to-understand format for formally specifying complex choices among alternative courses of action.

The topics relating to decision tables include: generalizations of decision tables that are more concise and support the use of decision tables in a refinement-based formal software development process; a formalism for systems of decision tables with behaviors; an exposition of Parnas tables for users of decision tables; and test coverage criteria and decision tables. We outline features of a revised version of ORA's decision table tool, Tablewise, which will support many of the new ideas described in this report.

We also survey formal safety analysis of specifications and software.

Contents

[1]We thank Steve Miller for numerous explanations that have made this chapter possible. Any errors that it still contains are the fault of the authors.

List of Figures

Chapter 1

Introduction

The theme of the research done on this project[1] and on its predecessor [12] is the use of formal methods and formally based tools to produce precise, understandable specifications and to help automate parts of the software production process, such as formulating specifications, checking specifications, deriving code and documentation from specifications, and testing of specifications and code. We have particularly concentrated on decision tables because, as discussed in Sherry [25], they are easy to understand, suitable for specifying iterative reactive systems of the kind common in avionics, and have a natural refinement methodology associated with them (*missions* to *operational procedures* to *scenarios* to *behaviors* to dependent missions).

In the predecessor to this task we developed Tablewise, a tool supporting formal specification of code using decision tables and generation of code from decision tables [12, 13]. Tablewise is a basic tool supporting editing of decision tables, testing decision tables for consistency (at most one outcome assigned under each possible scenario) and completeness (some outcome assigned to every possible scenario), a form of structural analysis that localizes faults responsible for tables being incomplete or inconsistent, and generation of Ada or C language code from decision tables.

Under this task, we have tried to broaden the scope of the project by identifying a broader range of ideas in formal methods that can be of use in avionics software development in general or in decision table methods or Tablewise in particular. The topics we have addressed are the following listed by chapter. For the most part, each chapter was originally written as a separate report on a particular topic.

- (Chapter 2) Extensions of decision tables.
  - Incorporating preconditions into decision tables. This is a method of indicating the context that a decision assumes and also of annotating relationships between the in-

[1]This research was carried out under NASA Langley Research Center Contract NAS1-20335, Technical Monitor Michael Holloway. We thank Lance Sherry for the substantial input he has made to the content of this report. We also thank Steve Miller for helpful and instructive discussions about testing, and David Rosenthal for his many useful comments.

put variables that mathematically guarantee that certain scenarios cannot occur and therefore can be excluded in analysis of the table.

- Permitting input variables to be partitioned into groups resembling records (as the term is used in programming languages). The resulting *partitioned decision tables* can express the same information as a "flat" decision table far more concisely.
- Assertion tables, which represent more general logical formulas than decision tables. Assertion tables permit one to write assertions about a specification that can be used either to generate a decision table or else to check whether a given decision table says what one thinks it should say.

Our algorithms for consistency and completeness analysis extend straightforwardly to decision tables that may be partitioned or have preconditions. Consistency and completeness analysis is essentially done automatically as part of generating a decision table from assertions. We had to generalize our notion of structural analysis in order to apply it to partitioned decision tables.

- (Chapter 3) Placing decision tables in a more general formal program development methodology. We propose a tabular specification method that permits a complex system to be specified using a family of decision tables. These tables specify not only how to choose one of a number of courses of action (the *selection*), but also what each course of action actually is (the *behavior*).

  The resulting system is essentially a variant of Parnas's A-7 specification method [10, 28], using decision tables with behaviors rather than Parnas's tables as the main definitional element, and using other tabular forms to declare system variables.

  Our system also bears a significant resemblance to Leveson's RSML [17] minus its statecharts. Leveson uses decision tables (selection part), divided up into pieces to define state transitions in her statecharts. Essentially, a state transition (arc in the statechart) in RSML corresponds to an operational procedure in ours. In Leveson's approach, as in its parent, Harel's statecharts [7], the overall state is represented partly by locations in the statechart and partly by values of state variables. In ours, as in Parnas's A-7 method, the state is represented entirely by state variables.

  A promising idea that we have not worked out here is the idea of a decision table as an object with an internal state (its operational procedure) and some associated methods (its behaviors). We think it will provide a basis for a compelling connection between state machine specification methods and decision tables.

- (Chapter 4) In the context of his A-7 specification method, Parnas uses a variety of specification tables. On the ground that these may be useful instead of or in conjunction with decision tables, we review the varieties of Parnas tables, explaining them in terms of decision tables.

- (Chapter 5) We explore the problems of generating code that is testable according to a certain test coverage criterion (MC/DC) and show how to generate such code along with a test suite meeting the coverage requirement from a decision table.

  We also explore the idea of a test coverage criterion for decision tables so that they can be tested in order to validate them as requirements.

- (Chapter 6) We present a short survey of formal and quasi-formal methods as applied to software safety.

- (Appendix A) We are developing a new version of Tablewise incorporating new methods reported here. Appendix A is a short description of what will be new in Tablewise 2.

We believe that the most important contributions reported here are the table extensions in Chapter 2 and the work on testing tables and test generation in Chapter 5.

1.1 Basics of Decision Tables

Operational Procedures

Operational Scenarios

Scenario Inputs (SI_i) and States (s_i)

Operational Behavior

Behavior Outputs (BO_i) and Functions (f_j)

Operational Procedure		Op Proc 1		Op Proc 2
Scenarios		Scen 1	Scen 2	Scen 3
Inputs	States			
SI_1	s_1, s_2	s_1	s_1	s_2
SI_2	s_1, s_2, s_3	s_1	s_1	s_1
SI_3	$s_1, \ldots, s_n$	*	s_2	s_n
Behavior		Behavior 1		Behavior 2
Outputs	Functions			
BO_1	f_1, f_2	f_2		f_2
BO_2	f_1, f_3	f_1		f_3
BO_3	$f_1, \ldots, f_m$	f_3		f_2

Figure 1.1: A decision table with both selection and behaviors parts [25].

A decision table (Figure 1.1, as presented in [25]) is a tabular format for specifying a choice of the actions that a system is to take. Overall, the decision table specifies a *mission*. The possible courses of action that will be used to implement the mission under different circumstances are called

operational procedures (or *opprocs*, for short). A decision table is divided into two parts. The top half specifies the conditions under which each operational procedure will be selected. This half of the table is called the *selection part*. The selection part is composed of two things.

- The *signature* lists input variables whose values affect the selection of an operational procedure and the possible values, or states, that those variables may take.

  In Figure 1.1, the input variables are SI_1, SI_2, and SI_3; the possible states of SI_2, are s_1, s_2, and s_3.

- The *body* associates an *engagement criterion* with each operational procedure. Each engagement criterion is a set of *scenarios*, and each scenario is a list associating a set of possible values to each input variable.

  In Figure 1.1, the engagement criterion of Operational Procedure 1 consists of Scenario 1 and Scenario 2. Operational Procedure 1 will be selected if either of those scenarios holds. Scenario 1 holds if SI_1, SI_2 have value s_1. The "$*$" opposite SI_3 means that it may have any value ("don't care").

  A scenario or an engagement criterion corresponds in a natural way to a logical formula that is true of the inputs if and only if the scenario or engagement criterion holds. We will frequently abuse terminology by speaking as if the scenario or engagement criterion and the corresponding formula were the same thing.

The bottom half of the decision table is the *behaviors part*. It specifies just what course of action, or *behavior*, is associated with each operational procedure. A behavior is an assignment of values to a number of *behavior outputs*. The values of behavior outputs are called *functions* (as in the word "functionality," rather than the phrase "mathematical function"). A function might be to assign a particular value to some quantity or to assign an algorithm for computing a given quantity from sensor inputs during the period that the operational procedure is in force. Some of the functions might be defined by reference to other decision tables.

Note that while several scenarios can be associated with an operational procedure, an operational procedure has only one behavior.

In all of this report except Chapter 3, we consider only the selections part of a decision table. The main reason is that the selections part is where the logical content of a decision table lies; therefore logical methods of analysis can be most easily and directly applied to the selections part. Nevertheless, decision tables with behaviors appear to have important connections with state machine based specification methods, such as statecharts [7, 17].

1.2 Summary of Table Forms

In Chapters 2 and 3 of this report we define a rather large number of tabular forms. We present a summary of them here. In Chapter 4 we describe in addition a number of forms defined by Parnas [22], but we do not describe them here, since they are relevant only to that chapter.

Selection table. The selection half of a decision table.

(Recursively) partitioned decision table. A decision table whose input variables have been grouped into records like those of Pascal or Ada.

Decision tables with preconditions. Decision tables in which certain opprocs (*Always, Never, Precond, Illegal*) indicate conditions which are or are not expected to hold of the input variables of a decision table.

Assertion table. A variant of a selection table that describes conditions under which various opprocs must or must not be selected.

Assertion result table. A table showing ways in which a decision table violates assertions made in an assertion table.

Type declaration table. A table defining names as abbreviations for complex types (essentially a table of type definitions).

Variable declaration table. A tabular representation of variable declarations.

Function table (functional decision table). A decision table with a single behavior output, called *output*, equipped with a list of formal parameter declarations that indicate how the table should be translated into a function subprogram.

1.3 Logical Notation

In mathematical contexts we use the usual logical notations for Boolean operations. For the convenience of the non-mathematical reader, we summarize them here.

Symbol	Meaning
$A \wedge B$	A and B (conjunction, meet)
$A \vee B$	A or B (disjunction, join)
$\neg A$	not A (negation, complement)
$A \Rightarrow B$	if A then B (implication)
$A \Leftrightarrow B$	A if and only if B (equivalence)

Chapter 2

Three Extensions of Decision Table Syntax and Semantics

This chapter gives the syntax and semantics for three extensions of decision tables: partitioned decision tables, decision tables with preconditions or illegal scenarios, and assertion tables.

Partitioned decision tables are decision tables in which input variables may be (recursively) grouped into records. This grouping has numerous advantages.

- Grouping input related variables helps organize a decision table and makes it more understandable.
- A partitioned decision table is typically less cluttered and can be far more compact than a "flat," simple decision table.
- Partitioned decision tables correspond closely to a well-structured "formal" English language specification that uses logical constructs in an orderly fashion.

As presented here, partitioned decision tables are more general than previously described [12, 13, 11]. In those presentations an entire table was partitioned to a fixed depth. Here, variables are individually structured into a structure of subvariables which is essentially the same as the concept of a record type in programming languages.

Use of preconditions allows one to indicate scenarios that are always or never expected to occur, clarifying the meaning of a table and making it possible for analysis to avoid reporting flaws in a table that can never be exercised.

The purpose of assertion tables is twofold.

- First they allow one to make assertions about what a decision table specifies. One can then check whether the table says what one thinks it says by testing whether it satisfies these assertions.

- Second, one can design a table by using assertions to describe what the table should say. One can then generate the table from the assertions, successively adding, removing, strengthing or weakening assertions until a consistent and complete table is obtained.

Assertion tables present an important opportunity to apply the method of *refinement* [29, 6] to the development of decision table-based specifications.

2.1 Recursively Partitioned Decision Tables

Ordinary "flat" decision tables have input variables that take values in an unstructured finite set. Their type is what programming language definitions call an enumerated type or an enumeration type. A standard way of structuring types is by forming *record types*. An variable of a record type is essentially a collection of variables (*fields*) of simpler types. A value of the record variable consists of an assignment of values to its fields. We can define record types based on enumeration types as follows. We assume that a class *Value* of atomic values and a class *Vble* of variable names are given.

$TYPE ::= ENUM\_TYPE \mid RECORD\_TYPE$

$ENUM\_TYPE ::= (w_1, \ldots, w_n), \quad w_i \in$ *Value* distinct

$RECORD\_TYPE ::= (v_1 : T_1, \ldots, v_n : T_n), \quad v_i \in$ *Vble* distinct, $T_i \in TYPE, i = 1, \ldots, n.$

A record type is also called a *signature*, because it is a sequence of variables with their types.

Observe that a record type as shown above is like the signature of a decision table, whose variables are $v_1, \ldots, v_n$ with types $T_1, \ldots, T_n$. If $T_1, \ldots, T_n$ were all simple types, it is clear in the context of decision tables what an assertion about a value of that type should be: an engagement criterion with entries suitable to $T_1, \ldots, T_n$. Formally, abstract entries, scenarios, and engagement criteria are defined as follows.

- An *entry* for a simple type $T = (w_1, \ldots, w_n)$ is a set $s \subseteq \{w_1, \ldots, w_n\}$.
- A *scenario* for a sequence of types $(T_1, \ldots, T_m)$ is a sequence $(e_1, \ldots, e_m)$ where for $i = 1, \ldots, m$, e_i is an entry for T_i.
- An *engagement criterion* for $(T_1, \ldots, T_m)$ is a set of scenarios for $(T_1, \ldots, T_m)$.
- An *entry* for a record type $(v_1 : T_1, \ldots, v_m : T_m)$ is an engagement criterion for $(T_1, \ldots, T_m)$.

The semantics of a recursively structured decision table is exactly what one would expect given the standard idea that a scenario represents the conjunction of its entries and an engagement criterion represents the disjunction of its scenarios.

Formally, the logical formula corresponding to an entry for a variable v of type T or a scenario or engagement criterion corresponding to a signature is defined as follows.

- If v is a variable, $T = (w_1, \ldots, w_n)$ and $s \subseteq \{w_1, \ldots, w_n\}$ then the formula of s for v and T is

  $$v \in s$$

  or equivalently

  $$v = w_{i_1} \vee \ldots \vee v = w_{i_m}$$

  where $s = \{w_{i_1}, \ldots, w_{i_m}\}$.

- If $\sigma = (v_1 : T_1, \ldots, v_n : T_n)$ is a signature and $s = (e_1, \ldots, e_n)$ is a scenario for $(T_1, \ldots, T_n)$, then the formula of s for σ is

  $$\phi_1 \wedge \ldots \wedge \phi_n$$

  where, for $i = 1, \ldots, n$, ϕ_i is the formula of e_i for v_i and t_i.

- If $E = \{s_1, \ldots, s_m\}$ is an engagement criterion for the signature σ, then the formula of E for σ is

  $$\psi_1 \vee \ldots \vee \psi_m$$

  where, for $j = 1, \ldots, m$, ψ_j is the formula of s_j for σ.

- If ϕ is a formula and v is a variable, then ϕ^v is the formula obtained by replacing each variable name v' occurring in ϕ by $v.v'$.

- If v is a variable of a record type $T = (v_1 : T_1, \ldots, v_n : T_n)$ and e is an entry for T, namely an engagement criterion for $(v_1 : T_1, \ldots, v_n : T_n)$, the formula of e for v and T is

  $$\phi^v$$

  where ϕ is the formula of e for T (considered as a signature).

In practise, we will avoid prefixing the name of a record-valued variable to the names of its fields by choosing distinct field names for fields of distinct variables.

As usual, a decision table (abstractly) consists of a signature σ and a body B. The body B is a function associating each of a set of operational procedures with an engagement criterion for σ. The table specifies that an operational procedure *opp* be selected whenever the formula of its engagement criterion for σ holds.

Operational Procedure			Takeoff		Climb		Climb_Int_Level	Cruise
Input Variables		States						
flightphase		climb cruise	climb		climb		climb	cruise
Altitude Status	AC_Alt > 400	TRUE FALSE	TRUE		*		*	*
	compare(AC_Alt, Acc_Alt)	LT EQ GT	LT		EQ GT		*	*
Altitude Target Status	Alt_Capt_Hold	TRUE FALSE	FALSE	TRUE	FALSE	TRUE	TRUE	TRUE
	compare(Alt_Target, prev_Alt_Target)	LT EQ GT	*	GT	*	GT	*	*

Figure 2.1: A recursively partitioned decision table.

Operational Procedure		Takeoff		Climb		Climb_Int_Level	Cruise
Input Variables	States						
Flightphase	climb cruise	climb	climb	climb	climb	climb	cruise
AC_Alt > 400	TRUE FALSE	TRUE	TRUE	*	*	*	*
compare(AC_Alt, Acc_Alt)	LT EQ GT	LT	LT	EQ GT	EQ GT	*	GT
Alt_Capt_Hold	TRUE FALSE	FALSE	TRUE	FALSE	TRUE	TRUE	TRUE
compare(Alt_Target, prev_Alt_Target)	LT EQ GT	*	GT	*	GT	*	EQ

Figure 2.2: A simple decision table.

2.1.1 An Example

Figure 2.1, adapted from [24], gives an example of a recursively partitioned decision table equivalent to the simple decision table in Figure 2.2.

In Figure 2.1, one of the top-level input variables, *flightphase*, is simple and the other two, *Altitude Status* and *Altitude Target Status*, are record variables with simple components. The grouping of *AC_Alt> 400* and *compare(AC_Alt,Acc_Alt)* into *Altitude Status* clarifies their meaning, but has not been used otherwise since their entries are essentially the same as if they had been simple variables. The entries for *Altitude Target Status* under *Takeoff* and *Climb* have been grouped into a mini-engagement criterion. Doing so avoids duplicating the entries for *flightphase* and *Altitude Status*, simplifying and further clarifying the structure of the table.

According to Figure 2.1, the engagement criterion of *Takeoff* is equivalent to the formula

$$\begin{aligned} &\mathit{flightphase} = \mathit{climb} \wedge \mathit{AC\_Alt} > 400 \wedge \mathit{AC\_Alt} < \mathit{Acc\_Alt} \wedge \\ &(\neg \mathit{Alt\_Capt\_Hold} \vee (\mathit{Alt\_Capt\_Hold} \wedge \mathit{Alt\_Target} > \mathit{prev\_Alt\_Target})). \end{aligned}$$

Engagement criteria in partitioned decision tables correspond closely to a well-structured English-language specification. For example, the engagement criterion of *Takeoff* can be written as follows.

> Operational procedure *Takeoff* will be engaged if all of the following conditions hold.
>
> 1. The *flightphase* is *climb*.
> 2. The *aircraft altitude* is above 400 feet.
> 3. The *aircraft altitude* is less than the *acceleration altitude*.
> 4. Either of the following conditions hold.
>   (a) *Altitude capture hold* is *false*.
>   (b) Both of the following conditions hold.
>   i. *Altitude capture hold* is *true*.
>   ii. *Altitude target* is above the *previous altitude target*.

As Figure 2.3 shows, tables in which several groups of variables can be grouped into records can offer a great saving in space over equivalent simple decision tables. (Note that in this table we have not bothered including names for the records, which are not essential if all field names are distinct.)

2.1.2 Decision Diagram Representations and Algorithms for Partitioned Decision Tables

Partitioned decision tables can be manipulated using decision diagram algorithms similar to those for simple decision tables that we described in [13, 12]. The decision diagrams associated with partitioned decision tables are, however, enriched by giving a record structure to the decision variables, as partitioned decision tables do.

Given the enriched decision diagram algorithms, consistency and completeness analysis are essentially the same for partitioned decision tables as for simple ones. The notion of structural analysis had, however, to be generalized in order to be applicable to partitioned decision tables.

The enriched decision diagram algorithms can potentially support similar analysis and manipulation of Parnas tables [22], described below in Chapter 4.

2.2 Decision Tables with Preconditions or Illegal Configurations

It seems that, for some decision tables, certain configurations of "illegal" or "impossible" values are expected never to occur. Looked at from another point of view, the table is expected to be used only when some precondition holds which excludes the illegal configurations.

Operational Procedure		*OP*		
Input Variables	*States*			
A	$T\ F$	T	F	
B	$T\ F$	T	$*$	
α	$T\ F$	T	F	F
β	$T\ F$	$*$	T	$*$
γ	$T\ F$	$*$	$*$	F

Operational Procedure		*OP*					
Input Variables	*States*						
A	$T\ F$	T	T	T	F	F	F
B	$T\ F$	T	T	T	$*$	$*$	$*$
α	$T\ F$	T	F	F	T	F	F
β	$T\ F$	$*$	T	$*$	$*$	T	$*$
γ	$T\ F$	$*$	$*$	F	$*$	$*$	F

Figure 2.3: Another partitioned decision table and an equivalent simple decision table.

A natural way to require or exclude certain conditions is by adding dummy operational procedures *Precond*, *Illegal*, *Always* and *Never*. Their intended meaning is as follows.

- The engagement criterion of *Never* denotes conditions which are *guaranteed* never to hold when the table is invoked, and which may be completely ignored in analyzing the table. For example, an overlap between two engagement criteria that is contained in the engagement criterion of *Never* will not be reported in an inconsistency analysis. In generating code, no action need be taken on inputs satisfying the engagement criterion of *Never*, because there will never be any.

  The two tables in Figure 2.4 are equivalent. They define the three-valued comparison *compare*. In this case, *Never* annotates a mathematical dependency between input variables: it is not possible for $x < y$ and $x > y$ to both to be true.

- *Always* is similar to *Never* but states restrictions in a positive form, i.e., conditions that will always hold. Scenarios lying outside the *Always* condition are entirely ignored.

- *Precond* and *Illegal* are respectively like *Always* and *Never* but are less strict. Their engagement criteria represent conditions that are *expected* always/never to hold when the table is invoked. Just in case, however, overlaps of engagement criteria that fall outside *Precond* or inside *Illegal* will be reported by consistency analysis, and in generated code, scenarios outside *Precond* or inside *Illegal* will raise an exception.

One can imagine further ways to treat preconditions, but there must be an end of such things. The four kinds of precondition annotations can, of course, be used in combination. They are provided in positive/negative pairs simply because sometimes it is simpler to express the negation of a condition as an engagement criterion, and sometimes it is easier to express the condition itself.

In general, excessive use of *Precond* and *Illegal*, or *Always* and *Never*, except when they annotate mathematical relations among input variables, make decision tables less reusable. If a table is correct only if used in a context that guarantees certain preconditions, then it cannot be used without modification in another context that does not guarantee the same preconditions.

In general extending our algorithms to tables with precondition annotations is trivial. Sometimes, however, one might want to express an engagement criterion in the simplest possible way given that it does not matter whether impossible scenarios are included or excluded. What is a good algorithm for performing such a simplification?

Operational Procedure		*LT*	*EQ*	*GT*	*Never*
Input Variables	*States*				
$x < y$	$T\ F$	T	F	F	T
$y < x$	$T\ F$	F	F	T	T

Operational Procedure		*LT*	*EQ*	*GT*	*Never*
Input Variables	*States*				
$x < y$	$T\ F$	T	F	$*$	T
$y < x$	$T\ F$	$*$	F	T	T

Figure 2.4: Two equivalent decision tables with precondition annotations.

2.3 Assertion Tables

Assertion tables permit one to specify more general relations between operational procedures and logical conditions than are possible in decision tables. In an assertion table, sets of operational procedures may be associated with several engagement criteria, and in several ways: in an *if*, in an *only if*, or in an *iff* (if and only if) relation.

- An *if* relation means that some opproc in the set must be selected if the engagement criterion holds, but may also be selected under other conditions.
- An *only if* relation means that an opproc in the set may be selected only if the engagement criterion holds, but does not have to be selected just because the engagement criterion holds.

- An *iff* relation is the combination of *if* and *only if*—the engagement criterion states the exact conditions under which some opproc in the set will be selected.

These different relations can be used to make assertions about what a decision table is meant to be like. These assertions can be used either to check whether a decision table satisfies them or to construct the least specific decision table that does satisfy them.

Figure 2.5 gives an example of an assertion table. It says the following things.

- If the flightphase is *climb* then one of the opprocs *Takeoff*, *Climb*, or *Climb Int Level* will be selected. One of these opprocs will be selected only if the flightphase is *climb*.
- *Takeoff* may be selected only if $AC\_Alt < Acc\_Alt$.
- *Climb* may be selected only if $AC\_Alt \geq Acc\_Alt$.
- *Climb Int Level* will be selected if and only if *flightphase* is *climb*, Alt_Capt_Hold is true, and $Alt\_Target \geq prev\_Alt\_Target$.
- If the flightphase is *cruise* then the operational procedure *Cruise* must be chosen.

		1	2	3	4	5
Assertions		*Takeoff, Climb, Climb Int Level*	*Takeoff*	*Climb*	*Climb Int Level*	*Cruise*
Input Variables	*States*	*iff*	*only if*	*only if*	*iff*	*if*
flightphase	*climb* *cruise*	*climb*	*	*	*climb*	*cruise*
$AC\_Alt < Acc\_Alt$	T F	*	T	F	*	*
Alt_Capt_Hold	T F	*	*	*	T	*
$Alt\_Target > prev\_Alt\_Target$	T F	*	*	*	F	*

Figure 2.5: An assertion table.

2.3.1 Checking a Table Against Assertions

Checking a selection table against an assertion table produces a result table listing violations of the assertions.

The selection table in Figure 2.6 is a sort of botched condensation of the table in Figure 2.1. Testing the table in Figure 2.6 against the assertion table in Figure 2.5 produces the result table in Figure 2.7. The result table indicates the following things.

- The first column says that Assertion 1 is violated because the indicated scenario is not assigned an operational procedure by the selection table.

Operational Procedure		*Takeoff*		*Climb*		*Climb Int Level*	*Cruise*
Input Variables	*States*						
flightphase	*climb* *cruise*	*climb*		*climb*		*climb*	*cruise*
AC_Alt < Acc_Alt	*T F*	*T*		*T*		*	*
Alt_Capt_Hold	*T F*	*F*	*T*	*F*	*T*	*T*	*T*
Alt_Target > prev_Alt_Target	*T F*	*	*T*	*	*T*	*	*

Figure 2.6: A table to test against assertions.

- No violation of the second assertion is listed, indicating that it is satisfied.
- The third assertion is violated by both scenarios under *Climb* in the selection table.
- The fourth assertion is violated because the engagement criterion of *Climb Int Level* fails to exclude the case *Alt_Target > prev_Alt_Target*.
- The fifth assertion is violated because the engagement criterion of *Cruise* restricts *Alt_Capt_Hold*.

Of course, most of these errors are silly and have been contrived to illustrate all the different kinds of possible errors.

Assertion Results		Assertion.1 (unassigned)	Assertion.3 Climb.1	Assertion.3 Climb.2	Assertion.4 Climb Int Level.1	Assertion.5 Cruise.1
Input Variables	States					
flightphase	climb cruise	climb	climb	climb	climb	cruise
AC_Alt < Acc_Alt	T F	F	T	T	*	*
Alt_Capt_Hold	T F	F	F	T	T	F
Alt_Target > prev_Alt_Target	T F	*	*	T	T	*

Figure 2.7: Results of assertion check.

2.3.2 Generating Selection Tables from Assertions

A natural way to develop a selection table is to write down principles it is supposed to embody as assertions, then build a table that satisfies those assertions. In fact, the selections table can be generated automatically from assertions. Figure 2.8 shows the table generated from the assertion table in Figure 2.5. This generated table is in fact complete and correct.

In general, however, a set of assertions will not fully specify a complete and correct decision table. Typically, we would expect a designer to proceed incrementally, as follows.

1. Write down an assertion.

Operational Procedure		*Takeoff*		*Climb*		*Climb Int Level*	*Cruise*
Input Variables	*States*						
flightphase	*climb* *cruise*	*climb*		*climb*		*climb*	*cruise*
AC_Alt < Acc_Alt	*T F*	*T*		*F*		*	*
Alt_Capt_Hold	*T F*	*F*	*T*	*F*	*T*	*T*	*
Alt_Target > prev_Alt_Target	*T F*	*	*T*	*	*T*	*F*	*

Figure 2.8: A table generated from assertions.

2. Generate and examine the induced selection table.
3. Add or modify assertions accordingly, or else correct the table by hand.
4. Go back to step 2.

To illustrate this process, consider the weaker set of assertions in Figure 2.9. Those assertions generate the decision table in the same Figure. In that decision table *Takeoff* and *Climb* have not been disambiguated and no restrictions have been placed in the opproc for the scenario in which *flightphase* is *cruise* and *Alt_Capt_Hold* is *false*. These deficiencies might be corrected by revising the assertion table to be the same as that in Figure 2.5.

2.3.3 Semantics of Decision Tables and Assertion Tables

In this section we will define and justify our algorithm for generating decision tables from assertion tables. Justifying the algorithm means defining the semantics of assertion tables and decision tables and showing that the semantics of an assertion table and the semantics of the decision table are equivalent.

Consequently, we will go into some detail about the semantics of decision tables and assertion tables, providing enough detail to prove that our algorithm preserves semantics. Thinking about decision tables in relation to assertion tables has also caused us to revise our earlier definition of decision table semantics [12]. The changes only affect tables that are incomplete or incorrect and therefore are irrelevant to the finished product of decision table development. We made the changes for the following reasons:

- to help design a sensible algorithm for generating a decision table from assertions;
- to make reducing inconsistency and incompleteness in a decision table correspond to refining it (strengthening the formula it denotes);
- to make refinement of an assertion table (i.e. adding assertions) correspond to refinement of the decision table it generates.

Assertions		*Takeoff, Climb, Climb Int Level*	*Climb Int Level*	*Cruise*
Input Variables	*States*	*iff*	*iff*	*if*
flightphase	*climb* *cruise*	*climb*	*climb*	*cruise*
AC_Alt < Acc_Alt	*T F*	*	*	*
Alt_Capt_Hold	*T F*	*	*T*	*T*
Alt_Target > prev_Alt_Target	*T F*	*	*F*	*

Operational Procedure		*Takeoff* *Climb*		*Climb Int Level*	*Cruise*	*
Input Variables	*States*					
flightphase	*climb* *cruise*	*climb*		*climb*	*cruise*	*cruise*
AC_Alt < Acc_Alt	*T F*	*	*	*	*	*
Alt_Capt_Hold	*T F*	*F*	*T*	*T*	*T*	*F*
Alt_Target > prev_Alt_Target	*T F*	*	*T*	*F*	*	*

Figure 2.9: A weaker assertion table and the decision table it generates.

Semantics of Assertion Tables

There is only one thing about the semantics of assertion tables that is not clearly annotated in the table: exactly which opprocs may be selected? We adopt the convention that only the opprocs listed in an assertion table are possible outputs. One can always make sure all intended opprocs are there by adding an assertion

$$(\textit{list of all selectable opprocs}) \Rightarrow \textit{true}.$$

Abstractly, an assertion table is a set of triples

(*engagement criterion*, *logical connective*, *set of opprocs*);

The *engagement criterion* is a logical formula in terms of input variables and their possible values. The *logical connective* is one of $\Rightarrow$, $\Leftarrow$, or $\Leftrightarrow$. The meaning $[\![A]\!]$ of an assertion

$$A = (EC, \Rightarrow, \{op_1, \ldots, op_n\})$$

is

$$EC \Rightarrow (OP = op_1 \vee \ldots \vee OP = op_n),$$

where OP denotes the opproc to be selected. The meaning $[\![\mathcal{A}]\!]$ of an assertion table

$$\mathcal{A} = \{A_1, \ldots, A_n\}$$

is

$$[\![A_1]\!] \wedge \ldots \wedge [\![A_n]\!] \wedge (OP = op_1 \vee \ldots \vee OP = op_m)$$

where $\{op_1, \ldots, op_m\}$ is the set of selectable opprocs.

Semantics of Decision Tables

Abstractly, a generalized decision table of the sort generated from assertions is a set of clauses, which are pairs of the form

(*engagement criterion*, *set of opprocs*),

where the engagement criterion is, as it always is abstractly, a logical formula in terms of the table's input variables and their possible values. In a "normal" decision table, the set of opprocs must always contain exactly one element, but in tables generated from assertions, the sets of opprocs may have more than one element or be empty. The difference from the elements of an assertion table is that the logical connective has been left out as understood. But what connective should be understood? The candidates are: $\wedge$, $\Rightarrow$, $\Leftrightarrow$ and $\Leftarrow$. Furthermore, should the interpretations of the element clauses of a decision table be conjoined, as for assertion tables, to form the interpretation of the whole table, or should they be joined together some other way? Two reasonable interpretations of a decision table present themselves. Let

$$\mathcal{D} = \{(EC_1, s_1), \ldots, (EC_m, s_n)\}$$

be a decision table (set of clauses).

- $[\![\mathcal{D}]\!]_1 = ([\![EC_1]\!] \wedge OP \in s_1) \vee \ldots \vee ([\![EC_m]\!] \wedge OP \in s_m)$.
- $[\![\mathcal{D}]\!]_2 = ([\![EC_1]\!] \Rightarrow OP \in s_1) \wedge \ldots \wedge ([\![EC_m]\!] \Rightarrow OP \in s_m)$.

Let us consider the properties of these two interpretations.

1. If $EC_1, \ldots, EC_m$ are exhaustive and mutually exclusive, then $[\![\mathcal{D}]\!]_1$ and $[\![\mathcal{D}]\!]_2$ are equivalent.
2. If EC_i and EC_j overlap but s_i and s_j do not, then $[\![\mathcal{D}]\!]_1$ is ambiguous (allows OP to be either a member of s_i or a member of s_j), but $[\![\mathcal{D}]\!]_2$ is inconsistent.
3. $[\![\mathcal{D}]\!]_1$ says that the input variables must always satisfy $EC_1 \vee \ldots \vee EC_n$. $[\![\mathcal{D}]\!]_2$ leaves the choice of opproc completely unspecified if $EC_1 \vee \ldots \vee EC_n$ does not hold.

4. In $[\![\mathcal{D}]\!]_1$, a clause $(EC, \{\})$ is vacuous—adding or deleting it does not change the interpretation. In $[\![\mathcal{D}]\!]_2$, $(EC, \{\})$ means that the input variables are forbidden to satisfy EC.

5. $[\![\mathcal{D}]\!]_1$ becomes stronger (is refined) if each $[\![EC_i]\!]$ is made stronger (smaller) and each s_i is made stronger (smaller). $[\![\mathcal{D}]\!]_2$ becomes stronger if each $[\![EC_i]\!]$ is made weaker (larger) and each s_i is made stronger (smaller).

Elsewhere [13] we have given $[\![\mathcal{D}]\!]_1$ as our interpretation of decision tables.

On overlap, we are inclined to believe that $[\![\mathcal{D}]\!]_1$ is the best interpretation: if engagement criteria overlap, that probably means we have failed to decide exactly which opproc to choose in some cases. Incompleteness, on the other hand, probably means that we have forgotten about some cases and have therefore failed to specify them. That corresponds to $[\![\mathcal{D}]\!]_2$. We therefore adopt the following modification of $[\![\mathcal{D}]\!]_1$ as our official interpretation of decision tables.

$$[\![\mathcal{D}]\!] = (EC_1 \wedge OP \in s_1) \vee \ldots \vee (EC_m \wedge OP \in s_m) \vee (\neg EC_1 \wedge \ldots \wedge \neg EC_m).$$

That is, on inputs satisfying some EC_i, $i = 1, \ldots, m$, OP must be selected compatibly with one of the clauses of the table. Otherwise the opproc selection is unspecified.

This interpretation preserves both overlap as ambiguity and incompleteness as failure to specify some cases. In $[\![\mathcal{D}]\!]$, a clause $(EC, \{\})$ also becomes a contradiction if EC is not contained in the union of the other engagement criteria. It is also the case that $[\![\mathcal{D}]\!]$ is equivalent to $[\![\mathcal{D}]\!]_1$ as long as $\mathcal{D}$ the engagement criteria of different opprocs in $\mathcal{D}$ are mutually exclusive.

Generating Decision Tables from Assertions

The interpretation of a decision table is a formula and so is the interpretation of a set of assertions. If a decision table is generated from a set of assertions, then those formulas should be equivalent. We achieve this by the following translation algorithm, each step of which preserves equivalence of interpretations. Tablewise uses a more efficient decision diagram algorithm to produce the same result.

1. Replace each assertion of the form $(EC, \Leftrightarrow, s)$ by two assertions $(EC, \Rightarrow, s)$ and $(EC, \Leftarrow, s)$.

2. Replace each assertion of the form $(EC, \Leftarrow, s)$ by $(\neg EC, \Rightarrow, s^c)$, where s^c is the set of all opprocs which do not belong to s but are mentioned in the table.

3. Replace each pair of assertions $(EC, \Rightarrow, s)$, $(EC', \Rightarrow, s')$ by the three assertions

$$(EC \wedge \neg EC', \Rightarrow, s),$$
$$(EC \wedge EC', \Rightarrow, s \cap s'),$$
$$(EC' \wedge \neg EC, \Rightarrow, s'),$$

discarding any of these for which the first component is equivalent to *false*. Repeat until for every such pair of assertions, $EC \wedge EC'$ is false.

4. For each set s of opprocs, if

$$(EC_1, \Rightarrow, s), \ldots, (EC_n, \Rightarrow, s)$$

are all the assertions with third component s, replace them by

$$(EC_1 \vee \ldots \vee EC_n, \Rightarrow, s).$$

5. Form a decision table $\mathcal{D}$ by eliminating the middle component of each assertion.

The final set of assertions $\mathcal{A}_f$ has the property that for any two distinct assertions $(EC, \Rightarrow, s)$ and $(EC', \Rightarrow, s')$ in it, $s \neq s'$ and $EC \wedge EC'$ is equivalent to *false*. We have

$$[\![\mathcal{A}_f]\!] = [\![\mathcal{D}]\!]_2 \equiv [\![\mathcal{D}]\!],$$

the equality by definition and the equivalence by our remark above that $[\![\mathcal{D}]\!]$ and $[\![\mathcal{D}]\!]_2$ are equivalent when the engagement criteria of different opprocs $\mathcal{D}$ are mutually exclusive, as is the case here.

Preconditions and Forbidden Scenarios

We require that assertions related to preconditions (*Always*, *Never*, *Precond*, *Illegal*) take one of the following forms.

$$(EC, \Leftarrow, \textit{positive restriction})$$

and

$$(EC', \Rightarrow, \textit{negative restriction})$$

That is, such an assertion must state either that a condition EC must hold or that a condition EC' must not hold.

In generating a decision table from assertions, we replace an assertion about a positive restriction, such as, $(EC, \Leftarrow, \textit{Precond})$, by a negative form, in this case $(\neg EC, \Rightarrow, \textit{not-Precond})$. It is necessary to do so because the semantics of positive preconditions is dual to that of negative preconditions and ordinary opprocs, but in generating tables we need to treat all opprocs and preconditions the same.

2.3.4 Refinement of Assertion Tables and Decision Tables

We mentioned above that we have modified our definition of decision tables and have set up our relation between decision tables and assertion tables in order to be compatible with a notion of refinement. Here, we will outline what refinement is, why it is important, and how refinement applies to decision tables.

A *specification* of a simple computer program or subprogram has the form (P, R), where P, the precondition predicate, is a logical formula in terms of the inputs to the program, and R, the input-output relation, is a logical formula in terms of both the inputs and the outputs. For example, **Z**

schemas [27] are always of this form—the part above the line is P and the part below the line is R. P defines the competency of the subprogram, that is, the conditions under which it is valid to invoke it. R specifies, though perhaps only partially, what effect the subprogram will have when it is invoked. Together, P and R amount to much the same thing as the single formula $P \Rightarrow R$, which can be considered as an input/output relation. Sometimes, however, R may not make sense or may be ill-defined when P fails, so it is best to indicate P separately.

An important operation on specifications is that of *refinement*. One refines a specification by making it logically more precise. We say that a specification (P', R') refines (P, R) if

$$P' \wedge R' \Rightarrow R.$$

That is, within its competence, R' is consistent with and at least as specific as R.

Refinement is an accepted formal method for developing specifications. One starts with a rather loose (weak) specification and proceeds to refine (strengthen) it until it is strong enough to translate easily into code.

What is refinement for decision tables? First, what is the specification associated with a decision table? There are two definitions: strict, counting only *Always* and *Never* as real restrictions and treating *Precond*, or rather *not-Precond*, and *Illegal* as ordinary opprocs; and liberal, treating *Precond* and *Illegal* as restrictions.

Let D be a decision table

$$\begin{aligned} D = \{ \quad & (op_1, EC_1), \ldots, (op_n, EC_n), \\ & (Always, EC_{Always}), \\ & (Never, EC_{Never}), \\ & (Precond, EC_{Precond}), \\ & (Illegal, EC_{Illegal}) \}, \end{aligned}$$

where $op_1, \ldots, op_n$ are distinct and not among *Always*, *Never*, *Precond*, and *Illegal*.

- The *strict* specification consists of

  $$P = EC_{Always} \wedge \neg EC_{Never}$$

  and

  $$R = EC_1 \wedge OP = op_1 \vee \ldots \vee EC_n \wedge OP = op_n \vee not\text{-}Precond \wedge \neg EC_{Precond} \vee Illegal \wedge EC_{Illegal}.$$

- The *liberal* specification consists of

  $$P = EC_{Always} \wedge \neg EC_{Never} \wedge EC_{Precond} \wedge \neg EC_{Illegal}$$

  and

  $$R = EC_1 \wedge OP = op_1 \vee \ldots \vee EC_n \wedge OP = op_n.$$

In either sense, adding assertions refines the specification (either kind) of the generated decision table.

Chapter 3

Tables with Behaviors

Elsewhere in this report we deal with decision tables that only choose a course of action, called an operational procedure, that shall be active under a given set of conditions, called its engagement criterion. But an operational procedure does cause something to happen; that something is called a *behavior*. A behavior consists of outputs, whose values are called *functions*, as in the phrase "what is the function of this object," rather than the phrase "mathematical function."

Once one begins to treat behaviors in decision tables, it becomes natural to use decision tables to define the different possible behaviors. One would want, then, to define a framework supporting systems of decision tables that refer to each other. This chapter is our preliminary essay in this direction. The ideas worked out here closely resemble Parnas's A-7 requirements model [10, 28]. The idea of regarding a decision table as an object, described below, suggests ways of going beyond what we do here and possibly making connections to explicitly state machine-based specification methods such as Harel's statecharts [7] or Leveson's RSML [17].

A classical way to view a decision table is as a means of denoting a subprogram in some programming language. In that case, a behavior consists of simply a value to be returned as a result (if a function subprogram) or an assignment of values to out variables or globals (if a procedure subprogram).

A more modern way to look at a decision tables with behaviors is as an object or state machine whose main internal state variable is the operational procedure. The function of the object may be to provide methods for computing certain values. How those values are computed depends on the current value of the operational procedure. From time to time the operational procedure itself would be determined as indicated by the decision table, either periodically or as triggered by some sort of event.

For example, suppose we have a system whose operational procedure is determined once per second, but whose function is to compute some control values fifty times each second.

It might be that under operational procedure O, the control value x will be computed as y^2, where y is the current value of some sensor, while under operational procedure O' the value x displayed is computed as $(w + z)/2$, where w and z are the values of two other sensors. What the operational

procedure determines is not the value of x, but the formula to be used to compute x fifty times during the second in which that operational procedure is in force.

These two points of view are not entirely at odds—they can be reconciled by imagining a subprogram that returns a mathematical function for computing the value of x, as can easily be, indeed, is best done in an "impure" functional programming language like ML. We remark also that an object-oriented programming language is not actually required to treat a decision table as representing an object.

Our point of view in this chapter will be the more old fashioned idea that a decision table is meant to specify a subprogram in a programming language. One reason for this choice is that we came to understand the idea of a decision table as an object after most of this work had been done, and have not had time to rethink it. The other reason is that the subprogram point of view is adequate to represent the idea of a function embodied by methods if mathematical functions can be returned as values.

For concreteness, we will imagine the programming language to be Ada.

Considering behaviors and regarding decision tables as subprograms requires that we face a number of problems.

- How are behaviors to be represented?
- Will it be one subprogram or more? In Ada or C++ we would implement the object oriented point of view by having one subprogram to select the operational procedure and one for each behavior output.
- How will the signature of the subprogram(s) be represented?
- The signature of a selection table acts as a set of declarations of its input variables or parameters. But those "variables" need not be or correspond to variables of a programming language—they can be more general expressions. The programming language variables in them need to be declared somewhere.
- Complex systems are normally assemblages of subprograms, organized according to some module structure. Presumably we would want the same for complex systems specified by families of decision tables.

We will spend the rest of this chapter addressing these problems.

If we were to carry the idea of systems of decision tables with behaviors to its logical conclusion, we would build a small design specification or programming language (or both) that included decision tables. In terms of Ada, a logical thing to do would be to embed decision tables into Ada as a way of defining subprograms. Such ideas are considered by Metzner [19]. We have not pursued things so far, however, partly because of the work it would involve, and partly because, from the point of view of design, we would first like to explore (elsewhere) the relationship of decision tables to statecharts, RSML, and object oriented methodology. Accordingly, we will keep the programming

language aspects of things as simple as possible. Nevertheless, thinking in terms of Ada gives a concreteness that helps point out some potential problems, so we will do so.

Since we originally did this work, we have had second thoughts about whether it is a good idea to try to connect decision tables closely to an existing programming language. Not to do so, however, would require making decision tables into their own language—perhaps no improvement—and would require code generation to be more sophisticated. Of course, this is by no means impossible, but would require much more thought and work.

This work is based on simple decision tables in which variables have simple types, and has not been upgraded to support partitioned decision tables and record types.

3.1 Table Definitions and Their Environments

A system of decision tables will be a list of the following kinds of things in any order:

- type declarations;
- variable declarations;
- decision tables (including selection, behavior, and subprogram signature).

We will define each of these in turn in the following sections. They may occur in any order, subject types and variables being declared before use. In order to permit recursion, we do not require tables to be defined before use.

3.2 Type declarations

A *type declaration* is essentially an Ada enumeration type declaration. It is of the form:

$$id : \quad (id_1, \ldots, id_n)$$

where $id, id_1, \ldots, id_n$ are Ada identifiers, $id_1, \ldots, id_n$ distinct; id is the type name, $id_1, \ldots, id_n$ the name of its elements. The type name id may not be *integer* or *boolean* (which are considered predefined) or any type name that has occurred earlier in the list of type declarations.

We have excluded *real* as a type, assuming that floating-point types will not be used because of the difference between machine arithmetic and mathematical arithmetic of real numbers.

Type declarations will be grouped in tabular form as in Figure 3.1. Descriptive phrases for the type and its elements could be added for use in generating documentation. Each type must have at least one element listed.

Type Name	Elements	Type phrase	element phrases
Flightphase	climb, cruise	flightphase type	climb, cruise
Comparison	LT, EQ, GT	comparison	less than, equal, greater than

Figure 3.1: A type declaration table.

A type declaration table induces a set *Type* of types, consisting of all types mentioned in the list, a set *Value* of values, consisting of all values listed, and a mapping $valset : Type \to \mathcal{P}_+(Value)$, where for $\tau \in Type$, $valset(\tau)$ is the set of values listed beside τ in the type declaration table.

The set *Value* is not necessarily the value set of any decision table, but any decision table defined in an environment with value set *Value* may use only values in *Value*.

3.3 Variable Declarations

A variable declaration is a pair

id : *type_name*

where *id* is the variable name, an identifier that has not occurred earlier as a variable name or as the name of an element of any type (including *TRUE, FALSE*), and *type_name* is the name of one of the types previously declared (or predefined). Like type declarations, variable declarations can be grouped together and represented in tabular form along with descriptive phrases, as in Figure 3.2.

Variable Name	Type	Description
flightphase	Flightphase	flightphase
ac_alt	integer	aircraft altitude
acc_alt	integer	acceleration altitude
alt_capt_hold	boolean	altitude target being captured or maintained
alt_target	integer	altitude target
prev_alt_target	integer	previous altitude target

Figure 3.2: A variable declaration table.

As for types, the variable declaration table defines a set *Variable* of variables, consisting of all the variables declared, a function $type : Variable \to Type$, mapping each variable, to the type listed beside it, and a function $valset : Variable \to \mathcal{P}_+(Value)$ given by $valset(v) = valset(type(v))$.

Note that *Variable* is not the same as the set *Vble* of "input variables" of a decision table. The "input variables" are actually expressions, defined below.

3.4 Expressions

Since expressions other than variables may appear in decision tables either as "input variables" or as values to be assigned to output variables, we need to define expressions before going on to define decision tables for a given environment (class of type and variable declarations).

We could use any reasonable notion of expression. For concreteness, we give a simple definition adequate for the examples we have encountered.

- An integer expression is either:
  - an integer variable or constant;
  - $abs(E)$, $(-E)$, $E_1 + E2$, $E_1 - E_2$, or $E_1 * E_2$, where E, E_1, E_2 are integer expressions.
- A Boolean expression is either:
  - a Boolean variable or constant;
  - $E_1 < E_2$, $E_1 \leq E_2$, $E_1 = E_2$, $E_1 \geq E_2$, or $E_1 > E_2$, where E_1 and E_2 are integer expressions;
  - *not* B, B *and* C, or B *or* C, where B and C are Boolean expressions.
- An expression of any other type is a variable or constant of that type.

For any type, function tables of that type, which we will define below, are also expressions.

Our definition gives each expression e a type $type(e)$. We can also define $valset(e) = valset(type(e))$.

3.5 States and Assignments

Given a set of variables V and an assignment $valset : V \rightarrow \mathcal{S}$ of a nonempty value set to each variable, a *state* for *valset* is a mapping $\sigma : V \rightarrow S$ such that for each $v \in V$, $\sigma(v) \in valset(V)$.

3.6 Decision Tables with Behaviors

The *environment* of a decision table declaration consists of the type and variable declarations earlier in the system as well as all decision tables.

A decision table with behaviors consists of the following items.

- A name, not already used in the environment.
- A set OP of operational procedures.

- A selections signature whose simple variables are all expressions of a type whose elements are the same as the set of values assigned by the signature. (In fact, the set of values is already determined by the type of the expression as determined by the environment.)
- A selections body, that is, a mapping *eng_crit* from OP to finite sets of scenarios.
- A set of variables O declared in the environment (the output variables).
- An assignment e for O (a map e defined on O such that for each $v \in O$, $e(v)$ is either an expression of the type of v or else a decision table in the environment that has v as an output variable. If $e(v)$ is a decision table, that table $e(v)$ is said to be a *dependent table* of the table in whose behaviors it occurs.

The semantics of the table is a binary relation R on states, defined as follows.

A pair of states $(\sigma, \sigma') \in R_T$ if for each table T' that occurs as an output expression in T there is a state $\sigma_{T'}$ such that $(\sigma, \sigma_{T'}) \in R_{T'}$ such that

- there is some opproc $op \in OP$ such that *eng_crit*(op) is true when variables are given the values assigned to them by σ; and
- for any variable v in the environment,

$$\begin{aligned} \sigma'(v) &= \sigma(v), && v \notin O \\ &= \sigma(e(v)), && e(v) \text{ an expression,} \\ &= \sigma_{T'}(v) && e(v) = T'. \end{aligned}$$

Notes:

- $(\sigma, \sigma') \in R_T$ means that if σ is the initial state, σ' is a possible output state. If T is complete and consistent, then there is only one possible output state, and R_T can be regarded as a function.
- We are assuming that each dependent table is invoked only once in an invocation of T, even though it may occur more than once in T. Furthermore, if a dependent table has output variables that do not occur in T, then their values do not change in the state.

3.7 Checking

A decision table with behaviors is complete (consistent) if it and all its (recursively) dependent tables are complete (consistent).

One would expect that in checking completeness and consistency of a dependent decision table, the (disjunction of the) engagement criteria of the opprocs under which it occurs could be used as

a precondition. In the examples we have looked at however, the variables relevant to dependent tables have been all or mostly different from those in the master table, so this kind of dependency checking seems, so far, to be irrelevant. Rather, the situation suggests an organization into objects so that the variables of different tables are (mostly) the state variables of different objects.

3.8 Function Tables

The use of dependent tables above was slightly odd. The master table corresponds to a procedure subprogram: it changes the state. A dependent table, however, acts more like function subprograms: it only changes the state of the output variable in whose row it occurs. The fact that "output expressions" are not just expressions, but either expressions or tables, suggests that we should augment expressions with tables corresponding to function subprograms, and eliminate the special case in output expressions.

Use of function tables also makes it possible to define expressions that we could not define before. For instance, the *compare* function, which appears in Figure 2.1, takes values of type *Comparison*, an enumeration type, is defined as follows.

$$\begin{aligned} \mathit{compare}(x, y) &= LT, \quad x < y \\ &= EQ, \quad x = y \\ &= GT, \quad x > y \end{aligned}$$

But according to our definition, the only expressions of an enumeration type are constants and variables, so for distinct integer variables x and y, $\mathit{compare}(x, y)$ cannot in general be defined.

A function table consists of the following items.

- A name.
- An output type.
- A variable declaration table, declaring the arguments of the table considered as a function subprogram.
- A decision table with behaviors that has only one output variable, called *output*. The environment of this table is the environment of the function table itself with the local variable declarations added to the environment variable declarations (superceding the environment if there are any name clashes).

  For each opproc *op*, the expression assigned to *output* under *op* must be of the output type.

A function table named T with arguments $x_1, \ldots, x_n$ of types $\tau_1, \ldots, \tau_n$ can be invoked as an expression by writing $T(E_1, \ldots, E_n)$ where $E_1, \ldots, E_n$ are expressions of types $\tau_1, \ldots, \tau_n$, respectively.

Figure 3.3 shows what a table defining the function *compare* would look like.

Function table		*compare*			
x	*integer*				
y	*integer*				
output	*Comparison*				
Selection					
Input Variables	*States*				*Illegal*
$x < y$	*TRUE, FALSE*	*TRUE*	*FALSE*	*FALSE*	*TRUE*
$x = y$	*TRUE, FALSE*	*FALSE*	*TRUE*	*FALSE*	*TRUE*
Behavior					
output	*Comparison*	*LT*	*EQ*	*GT*	

Figure 3.3: Function table for the function *compare*.

Notes.

- Given type and variable declarations, the listing of states (possible values) of input expressions of a decision table is redundant, though helpful. Listing the possible functions (expressions) that can be assigned to an output does not seem useful. Instead, we have listed the type, which is helpful in filling in the table.
- "Ordinary" (not functional) decision tables would look like functional ones except that the title would be something like "Procedure Table," there would be no list of local declarations, there would be headings giving operational procedure names, and there could be more than one output.
- Note that in the *compare* table, we have used *Illegal* to indicate that it is impossible for both $x < y$ and $x = y$ to be true. A potential enhancement would be to build in basic checking for order and equality.

Chapter 4

A Guide to Parnas Tables

Originally Parnas's A-7 specification method [10] relied on English-language specifications. Since then it has come to be more oriented toward formal specifications expressed in the form of tables [28]. In the report [22], Parnas describes a number of different kinds of tables for use in representing specifications or logical formulas in general.

The purpose of this chapter is to summarize Parnas's tables in a way easy for users of decision tables to understand and to examine ideas in Parnas's work that may be applicable to our work on decision tables.

The part of Parnas's report [22] that is most difficult to understand is the definitions of tables of more than two dimensions, which cannot easily be illustrated by a diagram. Our approach will be to avoid higher dimensions and abstract definitions, explaining by example. We suggest that readers who are familiar with decision tables and are interested in thoroughly understanding Parnas's report should first read this chapter, then Parnas's report concentrating on imagining what generalizations of tables to higher dimensions would be like, then Parnas's report another time carefully reading the definitions.

The table definitions are not, of course, the whole of Parnas's A-7 method. That method is similar to the specification framework described in Chapter 3, but with any of Parnas's tables used instead of only decision tables with behaviors.

4.1 Taxonomy of Parnas Tables

In all, Parnas defines ten kinds of tables:

- normal function tables;
- inverted function tables;
- vector function tables;

- normal relation tables;
- inverted relation tables;
- vector relation tables;
- mixed vector tables;
- predicate expression tables;
- characteristic predicate tables;
- generalized decision tables.

Besides these, he defines a number of forms (conjunction grids, union grids, substitution grids, and concatenation grids) designed to reduce clutter in favor of structure in the rows of tables. In this chapter we will discuss only the tables listed, not the ancillary forms.

Since Parnas's tables include several kinds of function tables, we will use the term "functional decision table" to denote what were called "function tables" in Chapter 3.

4.2 Background: Data States and Black-Box Program Specifications

The purpose of Parnas tables (and decision tables, for that matter) is to specify programs. Let us start, then, by reviewing the basic ideas of program specification.

By a data state of a program we mean a function $\sigma : \mathit{Variable} \rightarrow \mathit{Value}$ that assigns a value of the appropriate type to each variable of the program.

A *black box* specification of a subprogram is composed of two parts:

- a *precondition* $P(\sigma)$, which is a state predicate (set of states) indicating on which input states the subprogram can be expected to behave reasonably;
- an *input-output relation* $R(\sigma, \sigma')$ such that:
  - for any data state σ, if $P(\sigma)$ then there exists σ' such that $R(\sigma, \sigma')$;
  - whenever the subprogram is invoked in an initial state σ that satisfies $P(\sigma)$, the program will terminate and its final state σ' will satisfy $R(\sigma, \sigma')$.

Often there is only one intended output to a program on a given input; in that case the relation R can be replaced by a function F such that $F(\sigma) = \sigma'$ whenever $R(\sigma, \sigma')$.

Using an input/output relation instead of a function allows nondeterminism—a subprogram can be permitted to produce a variety of different outputs. Sometimes one may really want such a program,

but more often this is a specification device to avoid specifying at an early stage exactly what a subprogram should do (design nondeterminism—Heimdahl and Leveson [9] argue against this practise). Usually, though, a deterministic specification (with a function F) is easier to understand than a nondeterministic one (with a relation R).

Normally, σ and σ' will occur in R only in a form such as $\sigma(x)$, where x is a variable. The σ is mostly just clutter, so it is usual to let the variable itself stand for its value in a given state. The problem here is that we are interested in values of variables in two states, before and after execution of the subprogram. Most often this is done by writing x for the value of the variable x before execution and x' for the value after execution (this has the advantage that P does not have to have any primes in it). Parnas, however, writes 'x for the value of x before execution and x' for the value of x after execution. Parnas also calls R the *characteristic predicate* of the subprogram.

The classic specification of this kind is the **Z** schema [27], which consists of exactly the definition of such a P (the part above the line) and R (the part below the line), together with the necessary type and variable declarations.

Floating-point computations provide natural examples of nondeterministic specifications because one normally specifies the result only up to roundoff error. Consider, for example, a subprogram *sqrt(x,y)* such that, on input $x \geq 0$, on return x has not changed but y is the square root of x with relative error at most some constant ϵ. Its precondition P and input/output relation R would be

$$P(x) \iff x \geq 0,$$

$$R(x, y, x', y') \iff x = x' \wedge |y' - \sqrt{x}| < \epsilon\sqrt{x}.$$

4.3 The Tables

In the interest of understandability, we will mainly consider tables that can be represented in two dimensions. Mostly we will consider the different kinds of Parnas function tables and how they would correspond to functional decision tables.

4.3.1 Normal Function Tables

Consider Figure 4.1, which is Figure 1 in Parnas [22].

The table defines a function of $f(x, y)$ using different formulas for the result depending on the conditions on x and y listed along the side (for x) and across the top (for y) of the table. For instance, if $x < 3$ and $y > 27$ then

$$f(x, y) = y + \sqrt{-(x - 3)}.$$

Since this kind of a table defines a function, it should correspond to functional decision table. Figure 4.2 shows such a functional decision table.

Remarks:

	$y = 27$	$y > 27$	$y < 27$
$x = 3$	$27 + \sqrt{27}$	$54 + \sqrt{27}$	$y^2 + 3$
$x < 3$	$27 + \sqrt{-(x-3)}$	$y + \sqrt{-(x-3)}$	$y^2 + (x-3)^2$
$x > 3$	$27 + \sqrt{x-3}$	$2 * y + \sqrt{x-3}$	$y^2 + (3-x)^2$

Figure 4.1: A normal function table.

Function table		compare			
x	real				
y	real				
output	real				
Selection					
Input Variables	States				
compare(x,3)	LT, EQ, GT	LT,GT	LT	LT	EQ
compare(y,27)	LT, EQ, GT	LT	EQ	GT	LT
Behavior					
output	real	$y^2 + (x-3)^2$	$27 + \sqrt{3-x}$	$y + \sqrt{3-x}$	$y^2 + 3$
(continuation)					
compare(x,3)	LT, EQ, GT	EQ	EQ	GT	GT
compare(y,27)	LT, EQ, GT	EQ	GT	EQ	GT
Behavior					
output	real	$27 + \sqrt{27}$	$54 + \sqrt{27}$	$27 + \sqrt{x-3}$	$2 * y + \sqrt{x-3}$

Figure 4.2: A functional decision table corresponding to Figure 4.1.

- The table in Figure 4.1 does not directly specify whether x and y are formal parameters of the function or global variables. We have treated them as formal parameters.
- In this example, where almost every pair of conditions on inputs has a different output expression, Parnas's form is more compact and more perspicuous than the functional decision table.
- On the other hand, when there are more than two arguments, we have problems representing a normal function table on two dimensional paper.

One approach would be to present the table one two-dimensional slice at a time. An example with a third variable z, for which the relevant predicates are $z = 0$, $z < 0$, and $z > 0$, is shown in Figure 4.3.

An alternative approach would be to simply divide the variables into two groups, "stacking" them when one or both of the groups contains more than one variable. Figure 4.4 gives the stacked equivalent of the table in Figure 4.3.

(Both slices and stacking are our idea—they do not come from Parnas [22].)

- Corresponding to the requirement of consistency for a decision table, Parnas has a notion of a normal function being *proper*, namely that the conditions designating different outputs do not overlap.
- Note that the output expressions are arranged so that the argument of square root is always positive. The precondition of square root is that its argument is nonnegative.

$z > 0$	$y = 27$	$y > 27$	$y < 27$
$x = 3$	$27 + \sqrt{27z}$	$54 + \sqrt{27z}$	$y^2 + 3z$
$x < 3$	$27 + \sqrt{-(x-3)z}$	$y + \sqrt{-(x-3)z}$	$y^2 + (x-3)^2 z$
$x > 3$	$27 + \sqrt{x-3z}$	$2 * y + \sqrt{x-3z}$	$y^2 + (3-x)^2 z$

$z < 0$	$y = 27$	$y > 27$	$y < 27$
$x = 3$	$27 - \sqrt{27z}$	$54 - \sqrt{27z}$	$y^2 + 3z$
$x < 3$	$27 + \sqrt{-(x-3)z}$	$y - \sqrt{-(x-3)z}$	$y^2 + (x-3)^2$
$x > 3$	$27 + \sqrt{x-3}$	$2 * y - \sqrt{x-3z}$	$y^2 - (3-x)^2 z$

$z = 0$	$y = 27$	$y > 27$	$y < 27$
$x = 3$	27	54	y^2
$x < 3$	27	$y + \sqrt{-(x-3)}$	y^2
$x > 3$	27	$2 * y + \sqrt{x-3}$	y^2

Figure 4.3: A normal function table with three input variables, presented in slices.

4.3.2 Inverted Function Tables

The inverted form of function table is suitable when conditions on the variables are more numerous than the different formulas for output. The name "inverted" comes from the fact that, compared

		$y = 27$	$y > 27$	$y < 27$
$z > 0$	$x = 3$	$27 + \sqrt{27z}$	$54 + \sqrt{27z}$	$y^2 + 3z$
	$x < 3$	$27 + \sqrt{-(x-3)z}$	$y + \sqrt{-(x-3)z}$	$y^2 + (x-3)^2 z$
	$x > 3$	$27 + \sqrt{x-3z}$	$2 * y + \sqrt{x-3z}$	$y^2 + (3.-x)^2 z$
$z < 0$	$x = 3$	$27 - \sqrt{27z}$	$54 - \sqrt{27z}$	$y^2 + 3z$
	$x < 3$	$27 + \sqrt{-(x-3)z}$	$y - \sqrt{-(x-3)z}$	$y^2 + (x-3)^2$
	$x > 3$	$27 + \sqrt{x-3}$	$2 * y - \sqrt{x-3z}$	$y^2 - (3-x)^2 z$
$z = 0$	$x = 3$	27	54	y^2
	$x < 3$	27	$y + \sqrt{-(x-3)}$	y^2
	$x > 3$	27	$2 * y + \sqrt{x-3}$	y^2

Figure 4.4: A normal function table with three input variables, two of them "stacked" on the left.

to the normal function table, the output values and the conditions for one of the input variables have changed places. An example is Figure 4.5, which is Parnas's Figure 2.

	$x + y$	$x - y$	$x * y$
$x = 3$	$y < 3$	$y = 3$	$y > 3$
$x < 3$	$y < x$	$y > x$	$y = x$
$x > 3$	$y < -x$	$y > -x$	$y = -x$

Figure 4.5: An inverted function table.

Here, the conditions on x are written along the side, the conditions on y are in the body of the table, and the output expressions are along the top. For example, if $x < 3$ and $y = x$ then the output is $x * y$.

This arrangement permits a different set of conditions on y to be used for each condition on x. A different notion of properness applies: the conditions on x (the left hand column) must be complete and consistent and each set of conditions on y must be complete and consistent.

Recasting Figure 4.5 as a functional decision table gives Figure 4.6. Recasting Figure 4.5 as a normal function table would yield a more awkward three dimensional table.

Notes:

- Although in this case the inverted function table is rather neat, we can see that the circumstances that make it so are rather special: even though each condition on x is associated with a different set of three conditions on y, there are still only three output expressions.

Function table		*anonymous*								
x	*integer*									
y	*integer*									
output	*integer*									
Selection										
Input Variables	*States*									
compare(x,3)	*LT, EQ, GT*	*EQ*	*LT*	*GT*	*EQ*	*LT*	*GT*	*EQ*	*LT*	*GT*
compare(y,x)	*LT, EQ, GT*	*LT*	*LT*	*	*EQ*	*GT*	*	*GT*	*EQ*	*
compare(y,$-x$)	*LT, EQ, GT*	*	*	*LT*	*	*	*GT*	*	*	*EQ*
Behavior										
output	*integer*	$x+y$			$x-y$			$x*y$		

Figure 4.6: Decision table corresponding to Figure 4.5.

- It is perhaps even more obvious here than for normal function tables that properness is not just a matter of propositional logic, but may depend on other mathematical properties, usually properties of order.

4.3.3 Vector Function Tables

Vector function tables are to normal function tables as procedure decisional tables (those with several named outputs) correspond to functional decision tables. That is, they are basically the same thing except that they have several named outputs instead of a single unnamed outputs.

These tables have the drawback that only tables with a single input variable can be represented directly in two dimensions. In that case they essentially are decision tables. For example, consider Figure 4.7, which is Parnas's Figure 3.

	$w<0$	$w=0$	$w>0$
x	$x+y+q$	$x+2-q$	$x-w$
y	$y+2$	$x+y$	$x+y+2$
z	$z-w$	z	$z+w$

Figure 4.7: A vector function table.

As we understand it, a vector function table indicates an operation changing the values of the

output variables. Thus, in the row of Figure 4.7 corresponding to the output x, we understand x and y (and q) in the expression $x + y + q$ to mean the values of those variables before invocation of the table (or of the subprogram specified by the table).

Figure 4.8, the functional decision table corresponding to Figure 4.7 is nearly the same.

If we used the "stacking" representation for vector tables with several input variables, we would end up with tables quite similar to what we call Parnas-style decision tables, discussed below and shown in Figure 4.15.

Decision table		*compare*		
w	*integer*			
x	*integer*			
y	*integer*			
z	*integer*			
Selection				
Input Variables	*States*			
compare(w,0)	*LT, EQ, GT*	*LT*	*EQ*	*GT*
Behavior				
x	*integer*	$x + y + q$	$x + 2 - q$	$x - w$
y	*integer*	$y + 2$	$x + y$	$x + y + 2$
z	*integer*	$z - q$	z	$z + w$

Figure 4.8: Decision table corresponding to Figure 4.7.

4.3.4 Normal, Inverted, and Vector Relation Tables; Mixed Vector Tables

Normal relation tables are like normal function tables except that the entries of the body of the table, instead of being expressions of the desired output type, are Boolean expressions (logical formulas) containing a special symbol R standing for the output ("result"). The idea is that any value satisfying the formula is an acceptable output if that formula is selected. For example consider Figure 4.9, which is Parnas's Figure 4. If $x < 3$ and $\sqrt{y} > 27$ (and implicitly $y \geq 0$), then any output R satisfying $x^2 = \mathrm{R}^2$ (i.e., $\mathrm{R} = \pm x$) is acceptable. If $x = 3$ and $y < 0$ then any value of R is acceptable, since any value of R satisfies the formula *true*. If $x < 3$ and $y < 0$ then no value of R is acceptable.

Inverted and vector relation tables are analogous to inverted and vector function tables except that, as for normal relation tables, acceptable outputs are indicated by formulas that they must satisfy instead of expressions giving their value.

For vector relation tables, the output variables are named, so the formulas use the name of the

	$\sqrt{y} < 27$	$\sqrt{y} > 27$	$y < 0$
$x = 3$	$x^2 + y^2 = \mathrm{R}^2$	$x^2 = y^2$	*true*
$x < 3$	$y^2 = \mathrm{R}^2$	$x^2 = \mathrm{R}^2$	*false*
$x > 3$	$x^2 = \mathrm{R}^2$	$x = \mathrm{R} > 3$	$x^2 + y^2 = \mathrm{R}^2$

Figure 4.9: A normal relation table.

appropriate output instead of R. In my view, this makes vector relation tables somewhat confusing, because some occurrences of variables in the formulas specifying the outputs indicate values of variables before invocation and others values after invocation, and the only way to tell which is which is to see which variable the row belongs to. For instance, in Figure 4.10, which is Parnas's Figure 6, it appears that in the x row, x stands for the value after, but in the y row it stands for the value before invocation of the table.

	$w < 0$	$w = 0$	$w > 0$
x	$x = w$	$x^2 = 4$	$x^2 = w$
y	$y^2 = x + 2$	$y = x + 2$	$y = \lvert x \rvert + 2$
z	$z^2 = x^2 + y^2 + w^2$	$z^2 = x^2 + y$	$z = 5$

Figure 4.10: A vector relation table.

These problems are fixed in characteristic predicate tables, described below.

Mixed vector tables allow some output values to be specified by an expression and others by a formula, with a notation to indicate which are which.

We could make a decision table analog of relation tables by allowing formulas instead of expressions to define outputs.

4.3.5 Predicate Expression Tables

In form, predicate expression tables are like normal function tables, except that the entries of the body are logical formulas instead of terms. This class of tables provides a means to represent logical formulas two dimensionally without identifying some values as inputs and others as outputs.

A two-dimensional table with formulas $A_1, \ldots, A_m$ listed in the left-hand header, formulas $B_1, \ldots, B_n$ in the headers across the top, and formulas $C_{i,j}$, $1 \leq i \leq m$, $1 \leq j \leq n$, in the body denotes the

formula

$$\bigvee_{1 \le i \le m, 1 \le j \le n} A_i \wedge B_j \wedge C_{i,j}.$$

Here $\vee$ means "or" (disjunction) and $\wedge$ means "and" (conjunction).

A particular example is shown in Figure 4.11, which is compressed from Parnas's Figure 8. Figure 4.11 is equivalent to the following formula.

$$\begin{aligned}
&(x \le 3 \wedge w \le 0 \wedge y = 5) \\
&\quad \vee \quad (x \le 3 \wedge w > 0 \wedge y + x = w) \\
&\quad \vee \quad (x > 3 \wedge w \le 0 \wedge y > 7) \\
&\quad \vee \quad (x > 3 \wedge w > 0 \wedge y - x = 6)
\end{aligned}$$

	$w \le 0$	$w > 0$
$x \le 3$	$y = 5$	$y + x = w$
$x > 3$	$y > 7$	$y - x = 6$

Figure 4.11: A predicate expression table.

4.3.6 Characteristic Predicate Tables

Characteristic predicate tables are similar to predicate expression tables except that each variable x occurring in it occurs in two versions, 'x and x', never plain x, standing for the values of x before and after some subprogram is invoked. Thus a characteristic predicate table is suitable for defining the input/output relation of a procedure subprogram, the *characteristic predicate* of the subprogram. (In our discussion in section 4.2, we wrote plain x instead of 'x.)

A characteristic predicate table is in some ways an alternate (and, in our opinion, better) way to write a vector relation table.

Figure 4.12 is essentially Parnas's Figure 9. It says, for example that if the initial value of x is 3 and the initial value of w is 0, then the final values of x and w are both equal to the initial value of y.

Note that in Figure 4.12 the variables occurring in the headers are both "before" variables. We would expect this to be the normal case, though there is no rule that says header variables must be of that kind.

	$'w < 0$	$'w = 0$	$'w > 0$
$'x = 3$	$x' = 'x \wedge w' = 'x$	$x' = 'y \wedge w' = 'y$	$x' = 'y \wedge w' = 'y$
$'x < 3$	$y' = 'x$	$y' = 'y$	$w' = 'w$
$'x > 3$	$y'^2 = 4$	$x' + w' = 'y$	$y' = 'x$

Figure 4.12: A characteristic predicate table.

4.3.7 Generalized Decision Tables

The main difference between generalized decision tables and decision tables as we would write them is that the input expressions need not be finite valued and the entries of the body are predicates on that input expression rather than lists of permitted values. The symbol # indicates where the input expression for a given row goes in a predicate in that row. Parnas also permits just one output (as for a functional decision table) and writes the output values for each column at the top.

Figure 4.13 is Parnas's Figure 10. Figure 4.14 is the equivalent "ordinary" decision table.

	$x + y$	$x - y$
$x * y$	$\# < 20$	$\# < 20$
x/y	$\# \geq 20$	$\# = 20$
x^2	*true*	$\# > 20$

Figure 4.13: A generalized decision table.

Remarks:

- In the generalized decision table, the list of input expressions is, strictly speaking, unnecessary, since one can use them to replace the instances of # in the row, then delete the input expression listing, as in Figure 4.15. In fact, we think that would be more in the spirit of the other tables, whose principle characteristic is the use of logical formulas as entries.
- Writing predicates as entries in the body avoids making up functions like *compare*, whose meaning may not be immediately apparent to the reader. For this reason alone we think that this form of decision table is worth considering. The fact that entries to the table are predicates may pose some (minor) problems for our checking algorithms, but may also provide some new ideas about structured decision tables and structured decision diagrams.

Function table		*anonymous*	
x	*integer*		
y	*integer*		
output	*integer*		
Selection			
Input Variables	*States*		
$x * y < 20$	*TRUE, FALSE*	*TRUE*	*TRUE*
compare$(x/y,20)$	*LT, EQ, GT*	*EQ, GT*	*EQ*
$x^2 > 20$	*TRUE, FALSE*	$*$	*TRUE*
Behavior			
output	*integer*	$x + y$	$x - y$

Figure 4.14: Functional decision table equivalent to Figure 4.13.

$x + y$	$x - y$

$x * y < 20$	$x * y < 20$
$x/y \geq 20$	$x/y = 20$
true	$x^2 > 20$

Figure 4.15: The "true" Parnas-style decision table.

4.4 Conclusions

- Compared to decision tables, Parnas tables economize on space by eliminating the signature, but to some degree pay back by having more complex entries (formulas).

  Eliminating the signature has the more important benefit of allowing input conditions to be divided into two groups, as in Figure 4.4. Doing so reduces the need to duplicate conditions, as typically must be done in decision tables.

  Note that although in theory the input conditions to a Parnas table could be divided into arbitrarily many groups, but in practise, tables are written in two dimensions, so either the conditions must be put into two groups or else the table printed in slices.

  In the table definitions as given by Parnas and described above, each condition is a single predicate. There is, however, no real reason why such predicates could not be organized in tabular form like an engagement criterion in a decision table, as in a partitioned decision

table. The motivation for tabulating the individual predicates is the same as for using tables in the first place—it organizes the predicates expressed and makes them clearer and easier to understand.

Though we have not discussed them here, Parnas has a system of grids for abbreviating sets of formulas by tabulating them in a number of ways. Tabulating formulas into engagement criteria might be considered an extension of the abbreviation grids.

In general, we think that the idea of permitting general predicates to be table entries is interesting because it allows great flexibility. On the other hand, requiring entries to match a given signature, as in a decision table, helps classify and clarify the information in the entries, forces it to be simple, and makes it easier to perceive logical dependencies between entries. This is particularly true of partitioned decision tables.

- As to form assisting expression, inverted Parnas tables seem to be the most interesting since they provide a neat way to tabulate a decision involving two variables in which the conditions on the second variable are dependent on the first variable. Nevertheless, it does not seem entirely satisfactory because the results that can be obtained cannot also depend on the condition on the first variable as it can in normal function tables. Is there a compact tabular form that combines these features of normal and inverse function tables?

- Parnas tables all define functions or relations. Not having a notion comparable to operational procedure, they do not directly represent objects in the way that we suggested for decision tables in the previous section, though of course objects can always be modeled indirectly by setting state variables on which other functions depend.

  Parnas's A-7 specification method is properly a way of defining the functions and relations needed to define the transition and observer functions used in a complex system composed of state machines, but it does not provide a notation for directly defining the component state machines (objects), as Statecharts and RSML do. A form of object table might bring Parnas's method closer to the others and permit more synergy between the methods.

Chapter 5

Testing Decision Tables and Code Generated from Decision Tables[1]

No matter how rigorously formal methods may be applied, and no matter how completely its results are accepted, testing will still be needed at two points in the process of developing embedded software.

- At the beginning of the development process, the formal specification must be tested in order to validate it, i.e. to ensure that it says the right thing. Examining the results of these tests can also help designers and others understand what the formal specification does say. This testing of the specification amounts to a systematic simulation of a number of cases.
- At the end of the development process, when the software is finally integrated with the system's hardware, the entire system must be tested in order to confirm that the combined system of hardware and software behaves as expected.

Besides these two points, software development standards such as the quasi-regulatory DO-178B [23] specify the use of testing to verify correctness of software, and suggest the use of particular test coverage criteria in this testing. Generating test cases that meet one of these coverage criteria together with the test results specified generated by the decision table may therefore smooth the way to acceptability of code generated from decision tables.

Systematic testing normally requires that one apply a suite of tests selected so as to satisfy some test coverage criterion. The test coverage criterion represents some attempt to test rationally so as to maximize the likelihood of finding (certain types of) errors for a given amount of testing effort.

Section 5.1 describes a number of test coverage criteria mentioned in DO-178B. One of them is modified condition/decision coverage (MC/DC). MC/DC testing is believed to be as effective

[1]We thank Steve Miller for numerous explanations that have made this chapter possible. Any errors that it still contains are the fault of the authors.

in practise as testing according to the most thorough test coverage criterion, multiple-condition coverage (M-CC), but normally requires far fewer tests (typically $N+1$ for N conditions in MC/DC, compared to 2^N tests for M-CC). Drawbacks of MC/DC are that it is more difficult to understand than other test coverage criteria and that not all code even has a test suite providing MC/DC coverage.

Accordingly, this chapter does the following things.

- We discuss testing as a means of validating specifications in the form of decision tables. Testing a decision table by asking what result it would produce on a given input is not a problem. Rather, the problem is deciding on a reasonable test coverage criterion in order to get a good overall picture of what the table specifies. We propose an analog of MC/DC that applies to decision tables (as opposed to code) and an algorithm for generating an appropriate suite of tests.
- We address the need for testing code generated from tables by producing algorithms that generate MC/DC testable code from any decision table together with a test suite providing MC/DC coverage.

Testing is a big subject and we have been able to address only a very small part of it here. Testing or generating a test suite for an individual decision table or the code generated from it is only a small part of what is needed to validate the specification of a complex system or to provide a test suite adequate to test the software implementing that system when it is installed in the hardware it is to control. We believe, however, that it is a step in the right direction.

5.1 Coverage Properties and Associated Terminology

Here are definitions of some important test coverage properties as given in Chilenski and Miller [5]. The first four are quoted by Chilenski and Miller from the Glossary of DO-178B [23], though only MC/DC seems to be mentioned in the text of [23].

- Statement Coverage (SC): every statement in the program has been executed [by the test suite] at least once.
- Decision Coverage (DC): every point of entry and exit in the program has been invoked at least once and every decision in the program has taken all possible outcomes at least once.
- Condition/Decision Coverage (C/DC): every point of entry and exit in the program has been invoked at least once; every decision in the program has taken all possible outcomes at least once; and every condition in a decision in the program has taken all possible outcomes at least once. [I.e., DC plus: every condition in a decision in the program has taken all possible outcomes at least once.]

- Modified Condition/Decision Coverage (MC/DC): every point of entry and exit in the program has been invoked at least once; every condition in a decision in the program has taken all possible outcomes at least once; and each condition has been shown to independently affect the decision's outcome (by varying it while keeping the other conditions in the decision fixed).

- Multiple-Condition Coverage (M-CC): every point of entry and exit in the program has been invoked at least once and all possible combinations of the outcomes of the conditions within each decision have been taken at least once.

We say that a test coverage property A subsumes a test coverage property B if for every program P, any test suite for P satisfying A also satisfies B.

With the exception of M-CC and MC/DC, it is clear that each coverage property on the list subsumes the previous ones. M-CC does not subsume MC/DC, however, because M-CC does not require to show that all conditions have independent effect, as MC/DC does. In fact any program has a test suite satisfying M-CC. That is not the case for MC/DC, because, as we shall see later, in some programs not every condition has independent effect in the decision in which it occurs.

The definitions above use a number of technical terms that need to be explained. For concreteness, we will couch our explanation in terms of Ada programs.

- *Program.* This term must be taken to mean just one particular subprogram containing the conditions and decisions in question, not any context in which it is called or any further subprograms that it calls. The reason is that it will often not be possible to exercise, for example, all possible outcomes of every condition when a subprogram is used in a particular context.

- *Condition, Decision.* DO-178B gives the following definitions.

  Condition: A Boolean expression containing no Boolean operators.

  Decision: A Boolean expression composed of conditions and zero or more Boolean operators. A decision without a Boolean operator is a condition. If a condition occurs more than once in a decision, each occurrence is a distinct condition.

  We make the following observations.

  - Formally, conditions and decisions are *instances* of Boolean expressions rather than Boolean expressions.
  - The definition does not cover Ada `case` statements. According to Miller (private communication), a case statement

```
case COLOR of
  red, green => ...
  blue => ...
end case;
```

should be treated as if it were

```
if COLOR = red or COLOR = green then ...
else if COLOR = blue then ...
end if;
```

- In a statement

  ```
  if A and (B or C) then ...
  ```

  expressions `A and (B or C)`, `B or C`, `A`, `B`, and `C` are all decisions. Normally, one thinks only of the first, `A and (B or C)` (a maximal decision) as the decision, but a little thought will show that it makes no difference to the coverage properties whether decisions are all Boolean expressions or only maximal ones.

- A decision need not occur only in a control statement (`if then else`, `case` or `while`). For example

  ```
  A := B or C;
  ```

  both `A` and `B or C` are decisions.

- An *outcome* of a decision or condition is simply one of the values (`true` or `false`) that it may take.

- *Entry, exit.* In an Ada subprogram, the only point of entry is the beginning of the subprogram, exercised by every test, and the exits are the *return* statements and any exceptions that can be raised but are not handled, and, in case of a procedure subprogram, the end of the body of the subprogram and of each exception handler.

  We do not consider exceptions that are not explicitly raised in the subprogram as exits. As we understand it, critical programs are normally written so as not to raise predefined exceptions, such as `CONSTRAINT_ERROR`, and the property of not raising such exceptions is tested using some form of extremal testing different from what we are discussing here.

- *Statement.* An Ada statement.

- *Independent effect.* The key clause of the definition of MC/DC says:

  > A condition is shown to independently affect a decision's outcome by varying just that condition while holding fixed all other possible conditions.

  Here, we understand the phrase "all other possible conditions" to include at most the conditions occurring in the subprogram containing the particular decision. The word "possible" says that the outcomes of some of these conditions may also be varied, but is vague about the exact conditions under which it is permitted to vary them. We discuss this topic further below in Section 5.3.

  In any case, the two test cases required to show the independent effect of a condition on its decision must both cause control to reach the given decision. For example, the condition A does not have independent effect in the second decision in the following code.

```
if A then ...; elsif not A then ...; end if;
```

A does not independently affect the second decision because the second condition, if reached, can only come out true. This gives us a rule of thumb for making code MC/DC testable: a sequence of `elsif`s must end with a reachable `else`.

5.2 Test Categories vs. Test Cases

Properly speaking, a test case is an assignment of values to the variables (parameters and global variables) of a subprogram. Test coverage criteria, however, are defined in terms of how values of conditions and decisions must be covered rather than directly in terms of values of variables. There is a difference because some of the conditions are defined in terms of relations applied to actual variables, as with the condition `x < y`.

Values of conditions also play the leading role in decision tables. Consequently it makes sense to divide the task of finding a test case of a certain kind into two subtasks: first find a suitable combination of values of conditions, then find values of variables that give the required values to the conditions.

We call an assignment of values to conditions a *test category*. A test case which gives each condition the value assigned to it by a test category C is called a *realization* of C.

Because there can be mathematical relations between conditions, it is not always possible to realize a test category. For example, consider the following code.

```
if x < 0 then if x > 0 then raise Impossible; end if; end if;
```

A *test category* that exercises the statement `raise Impossible` is an assignment of the value `true` to each of the two conditions `x < 0` and `x > 0`.

A *test case* that realizes this test category must assign to `x` a value v that satisfies both $v < 0$ and $v > 0$. Since there is no such value, there is no such test case, and the test category cannot be realized by a test case.

In general, to find whether a test category is realizable by a test case, it is necessary to look into the structure of its conditions. Even then it is, in general, not possible to decide whether there exists any test case realizing a given test category. In some commonly occurring special cases, however, it is possible to determine whether test cases exist and, if so, find them.

- Ada variables of a Boolean or enumeration type.
- a linear inequality $a_1x_1 + \ldots a_nx_n + b \leq 0$, where $a_1, \ldots, a_n, b$ are constants with values known at test time.
- Equivalent to a finite function of the foregoing, but in the form of atomic expressions—no explicit Boolean operations. (E.g. the three-valued comparison, `compare`.)

The decision tables from avionics that we have worked with have all had conditions like this.

In this special case, we can, by a combination of linear programming and propositional logic, effectively determine whether a test category is realizable and, if realizable, find a test case that realizes it (see Nelson and Oppen [20] or Shostak [26]).

We will not further address test category realization here, however, but will concentrate on generating test categories whose realizations, if any, must satisfy certain test coverage requirements.

If all unrealizable test categories are documented among the impossible scenarios under the pseudo-operational procedure *Never*, then no unrealizable test categories will be generated from the table.

Test categories may also be useful because it appears to be easy to piece together a test category for a program with another test category for a subprogram it calls to form a test category for program and subprogram together. It does not seem so easy to do the same for test cases.

5.3 Independent Effect

The meaning of "independent effect" and of the phrase "all other possible conditions" that occurs in its definition requires more detailed discussion. Let us begin with the following simple example.

```
if p = q then ...; elsif p then ...; else ...; end if;
```

In order to show independent effect of the p in the second condition, we need two test categories v and v' that differ on p, say $v(p) = T$, $v'(p) = F$, both of which exercise that instance of p.

Let us make some observations.

- Technically, the p in the first decision is a distinct condition. Clearly it is not possible to change the p in the second decision without changing the other p in the first decision.
- In order to change the p in the second decision, we must also change the q in the first decision.

Thus, the two test categories must be $v(p) = T$, $v(q) = F$, and $v'(p) = F$, $v'(q) = T$.

Why does this pair of test categories show independent effect of the p in the second decision? Because under both, the second decision is executed and the other two conditions (i.e., other than the instance of p in the second condition) that v' changes are not in the second decision at all.

Reasoning in this manner, we arrive at the following definition of independent effect.

A test category v causes a condition p to be evaluated in a decision A if evaluating $v(A)$ does require evaluation of p, taking account of short circuiting. For example, if $v(p) = T$, then in p `or else` q, v causes the evaluation of p, but not q.

A test category v *masks* a set S of conditions in a decision A if $v(A)$ does not depend on the values of conditions in S. That is, treating the members of S as distinct variables distinct from all other variables in A and changing their values arbitrarily would not change $v(A)$.

The test categories v and v' show independent effect of the condition p lying in the decision B if the following conditions hold.

- B is evaluated under both of the test cases v and v'.
- $v(B) \neq v'(B)$.
- Both v and v' mask in B the set of all instances of p other than the one whose independent effect is being shown. Further, for every condition $q \neq p$ that is evaluated in B under both v and v', $v(q) = v'(q)$.

The limitation that only values of conditions evaluated under both v and v' need be preserved is intended to allow for legitimate use of short circuit operators to prevent conditions to be evaluated when they are not well defined. The following code provides an example.

```
x = 0 or else y/x > 100
```

It does not make sense to evaluate the second condition unless $x \neq 0$, so we do not require test cases giving different values to $x = 0$ to give different values to $y/x > 100$.

For better or worse, this rule makes it far easier to demonstrate independent effect when short circuit Boolean operators are used instead of "normal" ones. Consider the following code.

```
if
  p and not q then OP := op1;
elsif
  p or else q then OP := op2;
endif;
```

Call the second decision B. This code is strongly testable. The test cases

$$u(p) = u(q) = T,\ u'(p) = u'(q) = F$$

demonstrate independent effect of the p in B, while the test cases

$$v(p) = v'(p) = v'(q) = F,\ v(q) = T$$

demonstrate independent effect of the q in B. If the `or else` were replaced by `or`, which is not short circuit, changing the value of q between u and u' would not be allowed, so u and u' would not show independent effect of p in B. Indeed, in the code with `or` instead of `or else`, p does not have independent effect.

Now, in fact the instance of p in B is superfluous and the code could be rewritten so as to be MC/DC testable without any evasions. We cannot, however, show that it is always possible to rewrite code so that all conditions have independent effect without introducing short circuit operators in one way or another. This advantage suggests that the definition of independent effect should be weakened so that the use of short circuit operations does not make such a difference. We suggest the following.

(Weakly independent effect) Every condition q in B, other than the instance of p whose independent effect is being shown, such that $v(q) \neq v'(q)$ and q is evaluated in B under both v and v', lies in a subexpression C of B such that $v(C) = v'(C)$.

Short circuit operators still have it easier under this definition, but the difference is not as great and one can always rewrite pure conditional code to be testable in this sense without introducing short circuit operators.

In Section 5.6, we shall make shameless and systematic use of short circuit operations to weaken the requirements that we must meet in order to generate MC/DC testable code from decision tables.

5.4 Pure Conditional Code, Tree Code, Decision Trees, and Context

Starting in this section, we will speak about code generated from decision tables in more technical terms.

For the rest of this chapter, we will allow use only of conditions whose evaluation has no side effects.

5.4.1 Condition statements and Expressions

For simplicity, we will regard

`if` A `then` B `else` C `end if`

as the only form of control statement. All other forms of conditional statement are equivalent to one of these. For example:

- The statement

  `if` A `then` B `end if;`

  is equivalent to

  `if` A `then` B `else null; end if;`

- The statement

  `if` A `then` B `elsif` C `then ...end if;`

  is equivalent to

```
if A then B else if C then ...endif; end if;
```

- The statement

```
case E is
  when A | B .. C => S1;
  when D => S2
  when others => S3;
```

is equivalent to

```
if E = A or (B <= E and E <= C) then S1
else if E = D then S2
     else S3
     end if;
end if;
```

Strictly speaking, a more complex equivalent is required if E has side effects. We always assume that decisions have no side effects.

5.4.2 Boolean Expressions

We assume that the Boolean expressions that may be used are the following.

- Conditions (Boolean expressions that cannot be further decomposed).
- `not` A, A `and` B, A `or` B, A `and then` B, A `or else` B, A `=` B, where A and B are Boolean expressions.
- $A \rightarrow B \mid C$ where A, B, and C are Boolean expressions.

The semantics of the alternative operator $A \rightarrow B \mid C$ is as follows.

> Evaluate A. If the result is *true*, evaluate B and return its value; otherwise evaluate C and return its value.

The semantics of the other Boolean operations are as in Ada.

The reader will observe that $A \rightarrow B \mid C$ is just a form of `if ...then ...else ...end if` that applies to expressions instead of statements. Indeed, in some languages, such as Lisp or ML, there is no difference. Hence we will often refer to the alternative as the *conditional operator* in order to emphasize its close relationship to the conditional statement. The C language has a conditional operator, written A `?` B `:` C.

5.4.3 Straight Line and Pure Conditional Code

First, observe that the code generated from a decision table will always be *pure conditional code*, defined as follows.

- *Straight line code* is a sequence of assignments possibly terminated by a return statement, containing no non-constant Boolean expressions.
- *Pure conditional code* is either:
  - a statement sequence S of straight line code; or
  - A statement

    `if` A `then` S_1 `else` S_2 `end if;`

    where S_1 and S_2 are pure conditional code (and the decision A has no side effects).
- Tree code is the special case of pure conditional code in which each decision is a condition.
- Opproc-by-opproc code is of one of the following forms, where $S_1, \ldots, S_n$ are straight line code.

```
if D1 then S1;
elsif D2 then S2;
  ...
elsif Dn then Sn;
```

or

```
if D1 then S1;
elsif D2 then S2;
  ...
else Sn;
```

5.4.4 Context

The *context* of an expression B in a larger expression A or a statement S is a logical formula representing the strongest property that can be guaranteed always to be true when B is evaluated as part of evaluating A or S. Context is a sort of strongest precondition operator [6], and can be defined for statements as well as expressions. To define it in generality requires the general apparatus of predicate transformation, as discussed in [6]. For Boolean expressions without side effects and pure conditional statements, however, context is easy to define.

The importance of context is the following. In all our discussions of code and testability, we are assuming that predefined exceptions are not raised. What is it, however, that guarantees that

evaluation will not raise an exception? the context in which the expression is evaluated. For example, the expression `m div n`, for integer variables `m` and `n` will not raise an exception if it is evaluated in a context that implies $n \neq 0$.

Note that here, when we define the context of an expression B, we do mean an expression, not an instance of an expression B. The definition can, however, easily be modified to define the context of a condition or of any set of instances of expressions. The definition also applies when either A or B is an expression, statement, or statement sequence. (The most convenient point of view for us would be that of ML or C, in which everything is an expression and statements do not form a separate syntactic category, but expressions evaluated for side effects may be informally referred to as statements.)

Figure 5.1 gives the definition of the context of one expression in another. The definition of a condition, or any instance of an expression, is similar.

$context(B, A) = false$, if B does not occur in A

$context(B, B) = true$

$context(B,$ `not` $A) = context(B, A)$

$context(B,\ A$ `and` $A') = context(B,\ A$ `or` $A') = context(B,\ (A$ `=` $A')) =$
$context(B, A) \vee context(B, A')$

$context(B, A$ `and then` $A') = context(B, A) \vee (A \wedge context(B, A'))$

$context(B, A$ `or else` $A') = context(B, A) \vee (\neg A \wedge context(B, A'))$

$context(B, A \rightarrow C \mid D) = context(B, A) \vee (A \wedge context(B, C) \vee (\neg A \wedge context(B, D))$

$context(B,$ `if` A `then` S_1 `else` S_n `end if;` $=$
$context(B, A) \vee (A \wedge context(B, S_1) \vee (\neg A \wedge context(B, S_2))$

$context(B,$ `case` A `is when` C_1 `=>` S_1`;` `...when` C_n `=>` S_n`;` `end case;`$) =$
$context(B, A) \vee (A \in C_1 \wedge context(B, S_1)) \ldots \vee (A \in C_n \wedge context(B, S_n))$.

$context(B,$ `case` A `is when` C_1 `=>` S_1`;` `...when` C_{n-1} `=>` S_{n-1}`;` `when others =>` S_n`;` `end case;`$) =$
$context(B, A) \vee (A \in C_1 \wedge context(B, S_1)) \ldots \vee (A \in C_n \wedge context(B, S_n))$.

Figure 5.1: The definition of *context*.

Context is important for expressing code in different forms. We regard it as permissible to rewrite code as long as the semantics remains the same in the absence of predefined exceptions: that is,

any predefined exception that can be raised by the rewritten code can be raised by the original given the same initial state. If we require that in the rewritten code the context of each expression is at least as strong as in the original code, then this property will be satisfied.

5.5 Decision Tables and Short Circuit Operations

Chilenski and Miller [5] point out that short circuit operations are sometimes necessary to ensure that certain conditions will not raise an exception when evaluated, as in the following example.

```
if x /= 0 and then y/x < 100 then ...
```

If short circuit operators may occur in code generated from decision tables, we need a convention to indicate short circuit operators in decision tables. The simplest convention seems to be to the following.

- Interpret each column as a short circuit conjunction of the interpretations its entries, ordered from top to bottom, omitting "don't cares".
- For partitioned decision tables, apply this convention recursively to the entries.

We did not say anything about the columns of an engagement criterion (or a complex entry of a partitioned decision table) forming a short circuit disjunction. Rather, we regard them as forming an ordinary disjunction. The reason is that we prefer to consider the columns of decision table as occurring in no particular order. Regarding columns as unordered means that we cannot simplify later columns by omitting conditions implied by the failure of previous columns. Accordingly, a decision table must explicitly represent all conditions relevant to a given column.

Our convention gives a natural way to define context of an input variable in a decision table.

- The context of an input variable in a column is either
  - *false*, if the variable's entry in that column is "don't care" (that is, the variable does not occur), or
  - the conjunction of the entries above it in that column.
- The context of an input variable in a decision table is the conjunction of:
  - the strict precondition of the table (the conjunction of the engagement criterion of *Always* and the negation of the engagement criterion of *Never*);
  - the disjunction of the variable's context in the columns of the table other than the strong preconditions.

- For partitioned decision tables, apply this definition recursively to complex entries in the obvious way.
- The context of a condition, or an instance of any expression, in a decision table is defined similarly.

Regarding each column as a short circuit conjunction from top to bottom implicitly imposes the restriction that the context of a condition in the table can only mention other conditions higher in the table, never lower. One can imagine, however, three conditions A, B, and C, such that when A is true, C may not be evaluated unless B is true, and when A is false, B may not be evaluated unless C is true. The way to work around this problem is to enter one of the input variables into the table twice and specify under *Always* that the two copies always have the same value. Figure 5.2 illustrates this solution. The solution is a bit awkward, but we expect that this problem will rarely come up.

		O_1		O_2	O_3		O_4	*Always*	
A	T, F	T	T	T	F	F	F	*	*
B	T, F	F	T	T	*	*	*	T	F
C	T, F	*	T	F	F	T	T	*	*
B	T, F	*	*	*	*	T	F	T	F

Figure 5.2: Varying order of evaluation in a decision table.

Under this convention, the engagement criterion of O_1 in Figure 5.2 would be the following.

(A and then not B) or (A and then B and then C)

The context of C in the table is

$$(A \wedge B) \vee \neg A.$$

(We have omitted the *Always* condition because it is vacuous.)

For a table manipulation to be valid, not only must the tables be logically equivalent (considering that input parameters take an arbitrary possible value in cases where they would actually raise an exception), but the context of each input variable in the modified table or in generated code must imply its context in the original table. Then, if evaluating any column in the original table will not raise an exception, neither will an exception be raised in evaluating columns of the modified table or in executing generated code.

Our algorithms for manipulating decision tables all assume that decision table as written will require each input parameter to be evaluated only in a proper context. In the results they produce, whether code or another decision table, each input variable always has a context at least as strong as its context in the original table.

Perhaps it would be worthwhile, in a system like that described in Chapter 3, to provide a means of specifying the minimum context required by each input variable. One could then check that a given table provides a sufficiently strong context for each of its input variables.

5.6 Generating Testable Code and Tests from Decision Tables

5.6.1 Testable Tree Code

We say that a formula or Boolean expression B is *logically independent* of another formula A if both B and $\neg B$ are both consistent with A.

We say that a program is MC/DC testable in the context of a logical formula (Boolean expression without side effects) A (a strict precondition) if the MC/DC testability requirements are satisfied by a suite of test cases that satisfy A.

Theorem 5.6.1 *Tree code is MC/DC testable in the context of A if every condition in it is independent of the conjunction of A and its context in the code.*

Proof. We prove by induction on tree code S that for any formula A, if every condition C in S is logically independent of $A \wedge context(C, S)$, then each such condition C can be shown to have independent effect by a pair of test cases that satisfy A.

The basis case, where S is straight line code, is vacuous, since there are no conditions at all in S.

Suppose that S satisfies the hypothesis and is of the form

`if` B `then` S_1 `else` S_2 `endif;`

To demonstrate independent effect of B, choose test cases τ_1 and τ_2 respectively satisfying $A \wedge B$ and $A \wedge \neg B$. These exist since B is logically independent of A. They demonstrate the independent effect of B in a vacuous sense because no conditions other than B are evaluated under both τ_1 and τ_2.

Now demonstrate independent effect of some other condition C. Suppose that C lies in S_1, the other case being symmetric. Apply the inductive hypothesis to $A \wedge B$, S_1, and C to obtain test cases τ_1 and τ_2 satisfying $A \wedge B$, demonstrating the independent effect of C in S_1. These test cases satisfy A and show independent effect of C in S because they agree on all conditions that both evaluate in S, namely B and the conditions that both evaluate in S_1. □

It is easy to transform any pure conditional code into equivalent tree code. One way to do the latter is to replace a statement of the form

`if` A `and` B `then` S_1 `else` S_2 `end if;`

by

```
if A then if B then S1 else S2 end if; else S2 end if;
```

Repeat this and similar translation steps for other Boolean operations until all decisions are conditions. This translation preserves context. All conditions but those in B have the same context in the translation. Those in B have their context strengthened by adding A to it.

Once tree code is obtained, examine each condition to see whether it is independent of its context. If not, replace the statement containing it by its sole accessible branch. This operation preserves context because either the eliminated condition or its negation is already implied by its context. The resulting pruned code is equivalent to the original code in the context of any given precondition A.

It is natural to generate tree code from simple decision tables in such a way that the context of each condition in the code is at least as strong as its context in the decision table. Indeed that sort of code is generated by Tablewise [13]. Partitioned decision tables suggest a more complex kind of nested tree structure, the decisions corresponding to the possibly complex entries in the decision table. We do not know how to generate MC/DC testable code while substantially preserving the structure of the entries of a partitioned decision table. A modification of the definition of MC/DC may be necessary to obtain such an algorithm. The problem of generating such code may be the best place to start thinking about what such a modification should be like, as we consider below in Section 5.8.

5.7 Testable Opproc by Opproc Code

Opproc-by-opproc code is of the form shown in Figure 5.3, the `elsif`s being regarded as the equivalent `else if`s. It is natural to generate opproc-by-opproc code from a decision table because the decisions in the code correspond to the engagement criteria of opprocs in the decision table.

```
if D1 then S1;
elsif D2 then S2;
  ...
elsif Dn then Sn;
[ else Sn+1; ]
```

Figure 5.3: Opproc-by-opproc code.

A segment of opproc-by-opproc code will be testable if each decision D_i in it is testable in the context of $A \wedge \neg D_1 \wedge \ldots \wedge \neg D_{i-1}$ where A is some overall context.

Making a decision D testable in a context A can be accomplished by methods similar to those used to make tree code testable.

We say that a decision D is in tree form if it and every subexpression of it has the form $p \rightarrow B \mid C$, where p is a condition.

The algorithm for making a decision D MC/DC testable in the context of A goes as follows.

- Translate D into tree form, preserving context. The simplest method is the following.
  - Choose a condition p in D such that $context(p, D) = true$.
  - Let D^{true} and D^{false} be the formulas respectively obtained by replacing p by *true* and *false*, then applying this algorithm recursively to the result.
  - Reduce $p \rightarrow D^{true} \mid D^{false}$ using equivalences such as the following.

$$(q \rightarrow true \mid C) = (q \texttt{ or else } C) \tag{5.1}$$
$$(true \rightarrow B \mid C) = B \tag{5.2}$$

  Return the result.

- Prune conditionals in which the condition is not independent of the conjunction of A and its context.

Programming languages like Ada do not have conditional expressions, though some other languages such as C do have them (A ? B : C). In order to make a testable decision into a testable Ada expression, eliminate conditional expressions using the following function f in Figure 5.4.

Theorem 5.7.1 *If a tree expression B is testable in the context of A, $f(B)$ is equivalent to B and testable in the context of A. Furthermore, for any decision variable q, $context(q, f(B)) \equiv context(q, B)$.*

Proof. The clauses about equivalence of B and $f(B)$ and their associated context functions are trivial.

For testability, it will suffice to treat one of the middle cases, to show how they all go, and the last case.

Suppose then that B is $p \rightarrow C \mid T$ and $f(B)$ is not p or else C. Any pair of test cases v, v' that respectively satisfy $A \wedge p \wedge \neg C$ and $A \wedge \neg p$ demonstrate the independent effect of p. Such valuations exist because they are required to demonstrate independent effect of p in B.

Suppose that B is $p \rightarrow C \mid D$, $C, D \notin \{T, F\}$, and that $f(B)$ is

$$(p \texttt{ and then } C) \texttt{ or } (\texttt{not } p \texttt{ and then } D).$$

Since B is strongly testable given A and neither C nor D is either T or F, there exist valuations u, u', v, v' such that

$f(b) = b,\ b \in \{T, F\}$

$f(p \rightarrow T \mid F) = p$

$f(p \rightarrow F \mid T) = \texttt{not } p$

$f(p \rightarrow C \mid T) = \texttt{not } p \texttt{ or else } f(C)$

$f(p \rightarrow T \mid D) = p \texttt{ or else } f(D)$

$f(p \rightarrow C \mid F) = p \texttt{ and then } f(C)$

$f(p \rightarrow F \mid D) = \texttt{not } p \texttt{ and then } f(D)$

$f(p \rightarrow C \mid D) = (p \texttt{ and then } f(C)) \texttt{ or else } (\texttt{not } p \texttt{ and then } f(D)),\ C, D \notin \{T, F\}.$

Figure 5.4: How to eliminate the conditional operator.

- u satisfies $A \wedge p \wedge C$,
- u' satisfies $A \wedge \neg p \wedge \neg D$,
- v satisfies $A \wedge \neg p \wedge D$,
- u' satisfies $A \wedge p \wedge \neg C$.

The pair u, u' demonstrate the independent effect of the first, positive occurrence of p in $f(B)$, while v, v' demonstrate the independent effect of the second, negative occurrence of p in $f(B)$. □

5.8 Testable Code Containing More General Decisions

We have been able to give algorithms only for generating code with a rather simple special structure—either tree structured code or opproc-by-opproc code in which the decisions are tree structured. We could use the same methods to generate more general testable code, in which decisions could have more complex structure, if the conditions for independent effect were relaxed a bit so that a conditional expression

$$A \rightarrow B \mid C$$

would be treated in the same way as the conditional statement

$$\texttt{if } D \texttt{ then } S_1 \texttt{ else } S_2 \texttt{ end if;}$$

In the conditional statement, to show independent effect of a condition p in S_1 only requires two test cases τ_1 and τ_2 that both make D true and that show independent effect of p in S_1.

In the conditional expression, however, to show independent effect of a condition p in B requires two test cases τ_1 and τ_2, both making A true, showing independent effect of p in B, *and* agreeing on all conditions in A that both τ_1 and τ_2 cause to be evaluated.

If conditional expressions were treated in the same way as conditional statements, and a similar change in the rules was made for the short circuit operations **and then** and **or else**, then to show independent effect of a condition C in a decision D built using only short circuit operations, one would only need two test cases τ_1 and τ_2, both causing D to be evaluated (i.e. satisfying its context) and giving D different values.

This notion of independent effect seems a bit weak; perhaps tightening it up a bit would yield a more satisfactory test coverage criterion.

5.9 Test Coverage for Decision Tables

To test a decision table means to indicate what behavior it prescribes in a particular test case; that behavior is easily computed from the table. The idea of prescribing a coverage criterion to be met by a suite of tests is simply to ensure that the tests give a reasonably complete picture of what the table says.

The aim of this section is to consider the idea of test coverage criteria for decision tables, and in particular an analog to MC/DC. This work constitutes a preliminary essay toward developing such a test coverage criterion. We believe that further work along the lines indicated in Section 5.8 would help delineate the idea that should underly such a testing criterion.

Let us start by returning to the definition of MC/DC and see which clauses of its definition can apply to decision tables, and which cannot.

> Modified Condition/Decision Coverage (MC/DC): every point of entry and exit in the program has been invoked at least once; every condition in a decision in the program has taken all possible outcomes at least once; and each condition has been shown to independently affect the decision's outcome (by varying it while keeping all other possible conditions fixed).

The notion of entries and exits does not make much sense for decision tables. In code generated from decision tables, the one entry and all the exits get exercised automatically if other conditions are satisfied. We will therefore ignore the first clause.

The second clause brings up the question: in a decision table what is a condition and what is a decision?

The way conditions are grouped into decisions in code is rather arbitrary, as is shown by the contrast between tree-form code (many decisions, one condition per decision) and opproc by opproc code

(only one decision per opproc, many conditions in each decision). Our considerations in Section 5.8 suggest that it might be possible to dispense with or at least reduce the importance of the rather arbitrary way in which conditions are grouped into decisions. For the time being, however, we find a reasonable way to apply the notion of a decision to decision tables.

For a decision table, there are a number of reasonable ways to define decisions.

- A decision is an entry.
- A decision is a column, regarded as a short-circuit conjunction of its nontrivial entries, ordered from top to bottom.
- A decision is the engagement criterion of an opproc.

Here, we choose the second definition because it permits us to apply the idea of independent effect to conditions in decision tables in an unproblematic but nontrivial way.

Analogy with the treatment of **case** statements above makes it clear that the conditions are statements $v = a$ where v is an input variable of the table and a is one of its possible values. (Of course, $v = \mathit{true}$ can be abbreviated by v and $v = \mathit{false}$ by $\neg v$.) In the following table, the entry a, b, e for variable v denotes the disjunction of three conditions, $v = a \vee v = b \vee v = e$.

...	...
v | a, b, c, d, e	... | a, b, e | ...
...	...

Our interpretation of independent effect for tables will be guided by the idea that the context of an entry E in a decision table is the conjunction of the strict precondition of the table and the entries above E in the same column.

We regard a test category as an assignment of values to input variables, rather than directly to conditions.

Suppose that the entry E belongs to the input variable v, has context A, and $E = \{c_1, \ldots, c_n\}$. To show independent effect of the condition $v = c_i$, $c_i \in E$, we require two test cases, σ and τ, both satisfying A, with $\sigma(v) = c_i$, $\tau(v) \notin E$, and σ satisfying all entries of the column of E.

Taking individual entries as conditions, but with the same context, would eliminate the need for σ to satisfy the rest of the column.

We do not require that all conditions implied by a decision table have independent effect, since eliminating all conditions that do not have independent effect might interfere with modification and reuse of specifications. For example, we might want to use a given column in tables with different preconditions. One precondition might preclude independent effect of some conditions in the column, while the other precondition might not. Any such conditions will be eliminated in code generation in any case.

Rather, we say that a test suite $\mathcal{T}$ for a decision table T satisfies Tabular MC/DC for every condition C of every entry of T that has independent effect, $\mathcal{T}$ contains a pair of tests demonstrating the independent effect of C.

As an example consider the following table.

Input Variables	*States*	*A*		*B*	*Never*
p	a, b, c	a	b, c	b	c
q	T, F	$*$	T	F	F

The following test suite satisfies TMC/DC for this table. We use $*$ as usual for "don't care"—any value will do.

Tests		τ_1	τ_2	τ_3	τ_4
p	a, b, c	a	b	c	b
q	T, F	$*$	T	T	F

For example, according to our definition either pair $\{\tau_2, \tau_3\}$ or $\{\tau_2, \tau_4\}$ demonstrates independent effect of the second entry in the second column of the decision table.

As a final remark, we note that there is no direct connection between test suites that satisfy TMC/DC for a decision table T and test suites that satisfy MC/DC for various kinds of code generated from T.

Chapter 6

Safety Analysis

Previous chapters have considered table-driven code and the question of whether the entries in the table determine an unambiguous output for every input. We want to turn toward the bigger picture of how this code fits into the system as a whole.

Essentially, the component defined by each table is a state machine that interacts with other software and hardware components. It seems reasonable to picture the entire system as a collection of communicating state machines. This chapter summarizes a number of papers on requirements capture and software safety, especially for systems of state machines. All the papers originate in the group headed by Nancy Leveson, now at the University of Washington.

Some basic terminology (from [14]):

- *hazard*: a condition that could potentially lead to a mishap (accident)
- *safety*: the avoidance of hazards
- *risk*: a measure of the "expected value" of a system in terms of the likelihood of a hazard arising, the likelihood the hazard will lead to a mishap, and the cost of the mishap.

Safety is defined as the avoidance of hazards rather than the avoidance of accidents themselves, since the occurrence of an accident often involves conditions in the environment over which the system designer has no control. (If the railroad crossing gate is up when a train comes roaring through, that is a hazard even if it does not lead to an accident.)

Some distinct, but related notions: A *failure* is an error of any kind. The notion of failure treats all errors as equal—a misspelled message equals a nuclear meltdown. *Reliability* is the avoidance of failures. Safety and reliability sometimes conflict: a completely safe bomb, one that cannot go off, is completely unreliable.

The goal is to find systematic ways to design an acceptable level of safety into a system before producing it, by reducing or eliminating the occurrences of hazards. This poses problems in science (such as proofs of correctness), engineering (such as fail-safe design), and management. The special

difficulty introduced by software is that errors in software components almost never result from fabrication errors or failures of materials—whose likelihood can be estimated by well-founded statistical techniques—but result, rather, from errors in design; and software designs can be enormously complex. In real life catastrophes typically result when several things go wrong at once.

There are well-established general principles for safe design (in order of decreasing desirability):

- Intrinsically safe systems incapable of generating or releasing enough energy to create hazards.
- Elimination of hazards by design, for example
  - Lockouts, lock-ins, interlocks
  - Safety kernels that isolate critical from non-critical functions, that limit the authority of code or users to execute certain functions, etc.
- Control of hazards when they occur, for example
  - Fail-safe designs
  - Isolation of hazards
  - Detection of hazards (and appropriate recovery strategies)
- Warning devices, emergency training, etc.

Having, in some way, identified the hazards one can proceed to a safety requirements analysis—that is, to discovering the safety-relevant requirements on all the system components that will eliminate or reduce the likelihood of these hazards. The papers considered here are principally concerned with techniques for discovering the safety requirements on software components, and with general techniques for specifying system requirements (not only safety requirements).

The discussion will try to make it clear when we are dealing with mathematical procedures and when we are dealing with aids to systematic brainstorming, which may be formal in the sociological sense of being highly organized, but are not mathematical. In this chapter "formal" always means "mathematical."

We will use a slight variation on Leveson's terminology: *Hazard analysis* attempts to identify the hazards of a system or, derivatively, the hazardous behaviors of system components that could result in hazardous behavior of the system. It is necessarily informal—though the product of a hazard analysis might be a mathematical definition of the hazardous behaviors. *Safety analysis* begins with some list of hazards and a formal or informal model of a system (or system component) and tries to determine whether hazardous behaviors could actually occur. Safety analysis can be either formal or informal, and is applicable to those parts of the overall system components under the designer's control. Strictly speaking, a safety analysis can only attempt to bound the probability that a hazardous state will occur.

The paper [3] considers complex software-controlled systems with very stringent safety requirements (i.e., a vanishingly small probability of mishap during the system's expected operating life). It

argues that there will be no way to provide scientifically-based validation of such a system unless the system is designed with validation in mind. For example, the analysis will always rely on knowing certain probabilities that can only be estimated empirically. While there is a solid scientific basis for bounding the probabilities of hardware failure there is none for bounding the probabilities of software failure (not, at least, within the tolerances required). Thus the system must either be designed so that software failures are not critical, or so that failures of critical software are ruled out (e.g., by mathematical proof). Notice that we can often simplify the verification of a software component by considering its safety specification alone, and ignoring other aspects of its functional correctness.

This chapter represents safety analysis more concretely as follows: A system (or component) is a non-deterministic state machine, a hazard analysis defines a set of hazardous states of that state machine, and a safety analysis—which can be formal or informal—attempts to determine whether any of the hazardous states are *reachable* by executing the state machine from one of its allowed initial states. The picture can be dressed up by modeling failures as transitions to particular failure states, by assigning probabilities to failures, etc. This may not be the most suitable picture for all systems. In some cases it may be preferable to understand a hazard not as a dangerous state (a snapshot in time) but as a dangerous sequence of events.

Note finally that, like any analysis technique, a technique for safety analysis might simply provide a yes/no answer or might also help guide the search for an acceptable design.

6.1 Hazard Analysis

A hazard analysis attempts to identify hazards and evaluate their likelihood and seriousness, taking into account normal operation, operation in maintenance modes, failures (e.g., hardware failures), unusual occurrences in the environment, and operator error. In some cases some hazards are defined by government standards or by law.

By its nature, the process of hazard analysis is informal. The question of whether we have the right list of hazards, or whether the formal model is adequate to capture all the safety-relevant information is extra-mathematical. If the system (or component) whose hazards we are describing has a formal model, it is possible for the product of the hazard analysis to be expressed formally. Ultimately, some of the products of some of the hazard analysis will have to be interpreted formally, if we ever want to state precise requirements on the hardware and software components of the system.

It is expected that hazard analysis, in various forms, will continue throughout the development cycle. The paper [14] describes typical kinds of hazard analysis, including:

- Preliminary—an initial assessment to identify safety critical areas and functions.
- Subsystem—identifies hazards associated with the design of the subsystems (component failures, operators' errors, etc.)

- System—identifies hazards created by interfaces between subsystems, such as occurrences of multiple hazards or failures.
- Operating and support—examines, in particular, hazards created by the man-machine interface.

Various methods can be used to perform these analyses, including design reviews, walk-throughs, checklists, etc.

6.2 Safety Analysis

Whether safety analysis is done formally or informally, a common strategy is to work backward from a hazard in an attempt to identify credible scenarios that will bring the hazard about. This section describes some strategies for doing so.

6.2.1 Fault-Tree Analysis

One way of proceeding backward, informally, from hazards to hazardous scenarios is fault-tree analysis. A sample fault-tree for a patient monitoring system, taken from [14], is shown in Figure 6.1.

Each box corresponds to some possible situation. An expert in the problem domain and the design of the system analyzes each situation into an "OR" (a list of scenarios, any one of which leads to it) or an "AND" (a list of scenarios all of which taken together lead to it). When the top box in the tree is a hazard the tree is called a fault-tree. One typically elaborates the tree until each situation can be attributed either to a single system component or to the system's environment. We can then use the hazards attributed to a system component as a guide to formulating a safety specification for the component.

There can be no guarantee that this procedure is complete. It is an aid to brainstorming. Nor is there any guarantee that the component safety specifications obtained in this way, even if they are interpreted formally, will combine coherently so as to avoid the hazard. Notice finally that there is some ambiguity about what decomposition means. It can be understood as purely logical—e.g., decomposing "the door is not shut" into the OR of "the door is fully open," "the door is partly open." It can be understood as temporal or causal—e.g., decomposing "the door is not shut" into the OR of "the last person out failed to shut it," "the latch failed."

6.2.2 Fault-Tree Analysis of Software

Brainstorming via fault-trees need not stop at the boundary of a component: [16] suggests using a fault-tree walk-through of the code in order to check that it meets those safety specifications, or to determine where to insert checking code that will detect hazards. That paper considers software written in Ada, though the method is applicable to any language.

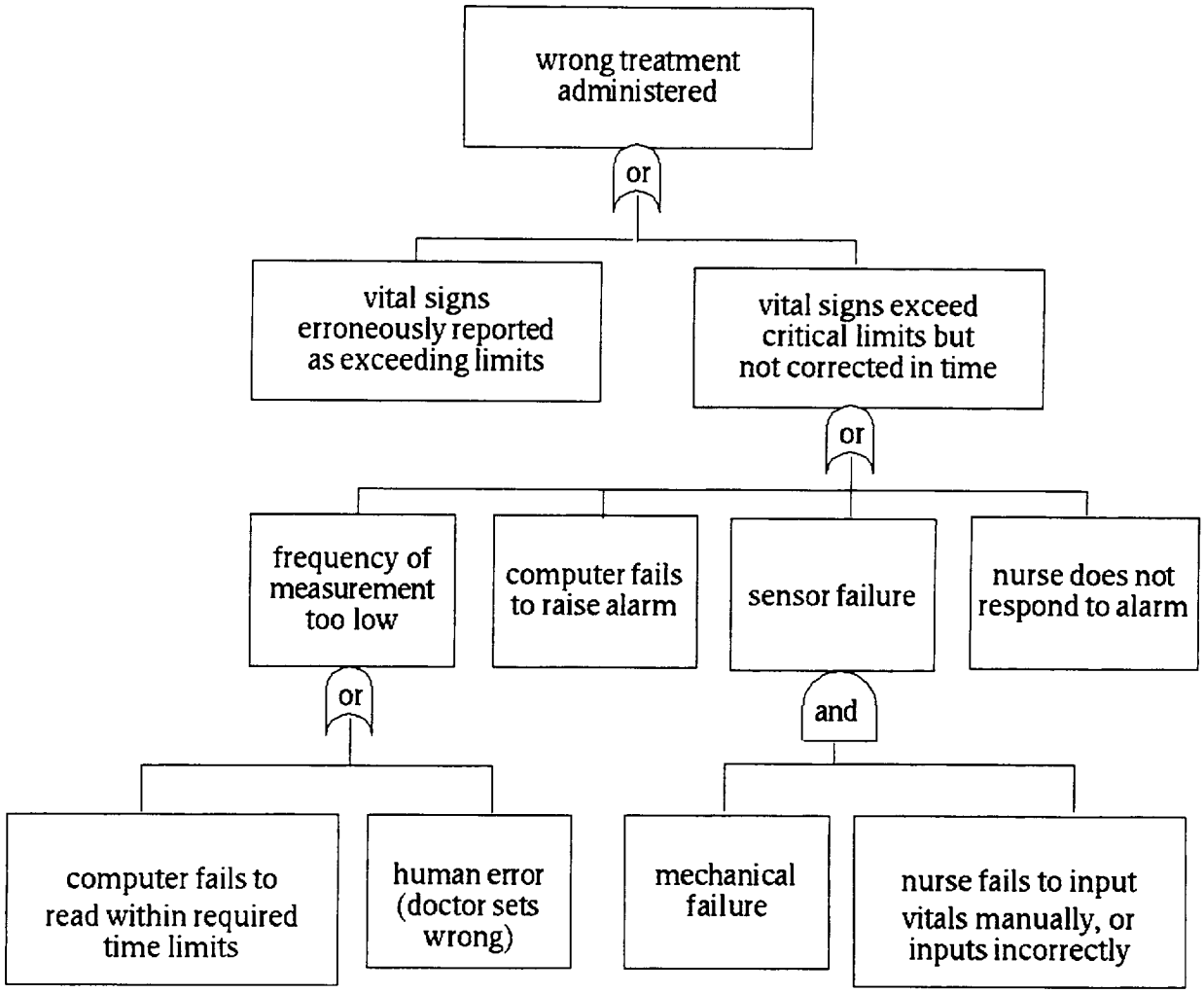


Figure 6.1: A fault-tree.

The software itself is associated with a well-defined model of execution, so it would seem that we could actually provide a formal safety analysis along lines suggested by the fault-tree strategy. This possibility will be discussed briefly in section 6.2.3.

The example given in the paper [16] considers a control system for a traffic light. Four sensors detect traffic approaching an intersection, and the controller uses reports from those sensors to change the settings of the lights. The sensors are labeled `north`, `south`, `east`, and `west`: the `north` sensor guards the north approach to the intersection, etc. The sensors are represented by an array `sensor` of Ada tasks indexed by these four directions.

The first step is to relate the hazards identified in the hazard analysis to events and states occurring

in the execution of the software. This is an informal procedure. The key part of each sensor task looks like this:

```
loop
   accept car_comes;  -- Accepting an interrupt from
                      -- the appropriate sensor device

   if (lights(dir) /= green) then
                      -- dir is the direction guarded by
                      -- this sensor

      controller.notify(dir); -- (+)
                      -- Rendezvous with the controller
                      -- while it resets the signals

   end if;            -- (*)
end loop;
```

The way hazard analysis works is to try to imagine each possible hazard and then check whether the code allows it to occur. Let us examine the process of checking for the hazard described as follows: "the north car enters the intersection as the east car enters." This symmetric-sounding situation is (implicitly) decomposed into subcases, and the particular case analyzed is "the `sensor(east)` task is in rendezvous with the controller task and the `sensor(north)` task bypassed the rendezvous":

- `sensor(east)` is in rendezvous if it is at `(+)`, waiting for the completion of the rendezvous `controller.notify(east)`.
- `sensor(north)` bypassed the rendezvous if it reaches `(*)` because the test `lights(north) /= green` was false.

So the authors have not only identified a particular hazardous subcase, but also interpreted it in the form of events in the code. This interpretation requires insight that goes beyond identifying the hazard in the first place.

The idea is that a fault-tree walk-through should identify execution sequences leading to the hazard. Each programming language construct corresponds to a "generic" template representing the fault-tree analysis for all instances of the construct. For example, Figure 6.2 gives the template for an if-then-else. Fault-tree analysis of the code proceeds by applying these templates systematically. In the case of tasks executing in parallel, the templates are applied separately in each task. There are also templates representing the entry calls and accept statements whereby tasks communicate. (This fault-tree procedure seems to omit the possibility of tasks communicating by shared variables.)

The authors apply this if-then-else template to the execution of the following conditional

```
if (lights(north) /= green) then
```

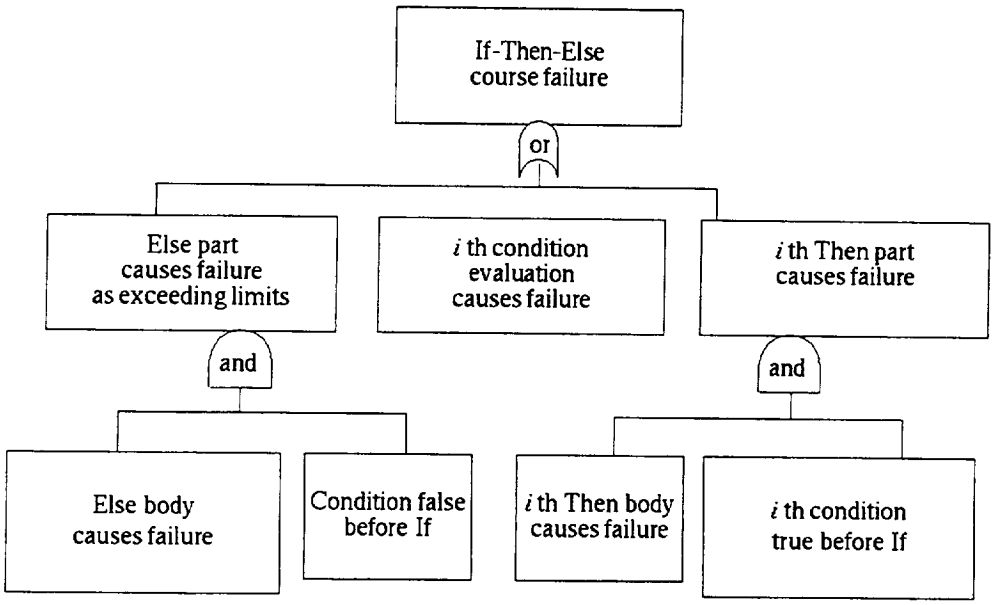


Figure 6.2: Fault-tree for a conditional.

```
      controller.notify(north);  -- (+)
-- else
--    null;
   end if;                       -- (*)
```

in the task `sensor(north)`. (To make the relation to the template clearer we have added, in comments, a vacuous "else" part.)

Consider how the analysis of this statement proceeds inside task `sensor(north)` where the failure condition is interpreted as "`sensor(north)` is at `(*)` after skipping the rendezvous"—presumably with the additional understanding that the `sensor(east)` task is simultaneously at `(+)`.

Since this failure condition assumes that the "then" part is not taken, and since there is no real "else" part, the only possibility left to consider is that the condition evaluated to true and its evaluation caused a failure, which the authors expand as in Figure 6.3. The "lights(north) is green ..." condition restates one of the assumptions about the case we are in. Where did the condition "evaluation occurs as controller changes light" come from? It could result simply from puzzling about what could go wrong; it could also be regarded as the place, at last, to record explicitly the

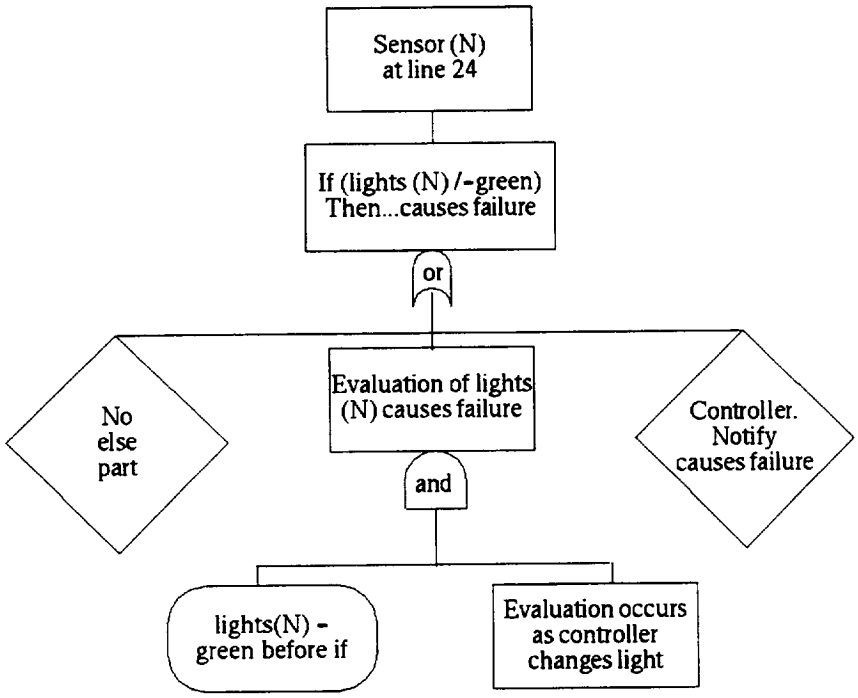


Figure 6.3: Applying the template.

assumption about the other task that makes this a hazard.

6.2.3 Making the Analysis Mathematical

Software fault-tree analysis provides a systematic framework for walking through code with safety requirements in mind. It would take some hard work (especially, work on the semantics of tasking) to make the procedure fully formal.

We should first of all note that the not-fully-mathematical nature of the procedure described here gives the authors no heartburn at all. They say that "industrial users of the technique have commented that the actual process of having people examine the code thoroughly was crucial to its effectiveness."[1]

[1]The authors mention tools for automating generation of the fault-trees, but it is not clear whether that simply means some kind of structure editor that would pop up appropriate templates or whether it means something with

The authors note that their procedure is essentially predicate transformation, a well-known technique for analyzing the consistency of software with some given specification. In a certain sense they are merely doing predicate transformation with respect to the predetermined specification "a hazardous state does not arise."

There are two problems with making this procedure precise. One is the highly technical one of understanding how to do predicate transformation at all for a complex language, in particular for code that involves tasking. The other is to define the language of "software events" into which the assertions about hazards are translated.

For the example presented, the language of software events must be able to assert that executions in two tasks simultaneously reside at certain control points, and that execution in one of those tasks got there by proceeding along a certain path (because the if-test was false). One question is whether, in general, we know what this language of software events will be.

Predicate transformation is not the only way to formalize safety analysis of software. We could instead translate the code directly into a state machine and then use one of the techniques for analyzing state machines. To make this tractable it might be necessary to reduce the number of states by abstracting away from unnecessary information. If, for example, only the sequence in which tasks rendezvous is important (not the values passed at each rendezvous) we can consider a model that ignores the values passed. This may well lead to a model that contains execution sequences impossible in the original, but such a conservative approximation is legitimate.

6.2.4 Formal Safety Analysis via Petri Nets

To repeat: a formal safety analysis (of the kind we are considering in this chapter) starts from a formal model of a system as a state machine and a formal definition of all its hazardous states. The paper [18] considers the case in which the system is formally modeled as a *Petri net.* The analysis attempts to determine whether the hazardous states are reachable—and, if so, to provide guidance on how to redesign the system.

Intuitively, we can think of a Petri net model of a system as follows: the state of the net at some moment represents a *property* of the system state at that moment. If the net can make a transition from state s_1 to state s_2 that means that the system can make a transition from a state in which "the property represented by s_1 is true" to a state in which "the property represented by s_2 is true."

Untimed Petri Nets

Somewhat more precisely, a Petri net consists of a collection of *places* and a collection of *transitions.* Graphically, we represent each place by a *circle* and each transition by a *bar.* In Figure 6.4, taken from [18], the places are $P_1, \ldots, P_{12}$ and the transitions are $t_1, \ldots, t_7$. Each transition is associated with one or more input places (graphically, arrows go from the input places to the transition bar)

more semantic content.

and one or more output places (arrows go from the transition bar to its output places). So, for example, in Figure 6.4 the only input place of transition t_1 is P_1 and its output places are P_2 and P_3.

Each place may contain zero or more *tokens*, and the distribution of tokens on places is called the *marking* of the net. Figure 6.4 has tokens in places P_1, P_6, and P_{11}, while Figure 6.5—a different marking of the same net—has tokens in places P_2, P_5, P_6, and P_{11}. The state of a net is its marking. We can think of each place as representing a simple proposition about the system, which is true when the place contains a token and false otherwise.[2]

The nets shown in Figures 6.4 and 6.5 model different states of a system that consists of a train, a crossing gate that guards an intersection, and a controller for the gate. Place P_{11} represents the proposition that the crossing gate is up and P_{12} the assertion that it is down. Thus, the markings in both Figure 6.4 and Figure 6.5 "say" that the gate is up (because P_{11} contains a token). Similarly, the places $P_1, \ldots, P_4$ represent propositions about the train: P_1, which is marked in Figure 6.4, says that the train is approaching the crossing, while P_2, which is marked in Figure 6.5, says that it is about to enter the crossing.

The transition rules tell us how to model the evolution of the system by moving tokens around the net. A transition t is *enabled* if all its input places have tokens in them. In Figure 6.4 transition t_1, which represents the detection of the approaching train, is enabled. The transitions enabled in Figure 6.5 are t_2, representing the act of entering the crossing, and t_4, an internal transition of the controlling computer. When an enabled transition *fires* the tokens are removed from its input places and one token is put on each of its output places, and the resulting marking "says" something about the system in its new state.[3] Figure 6.5 is the result of firing transition t_1 in Figure 6.4. A Petri net is executed by repeatedly firing enabled transitions.

In terms of this model we can formally describe what the product of a hazard analysis is supposed to be—a list of all the markings of the Petri net that represent hazards. (The way in which we decide what constitutes a hazard is still non-mathematical.) In this case, presumably, the hazards are all those markings in which P_3 and P_{11} have tokens; that is, all those situations in which the gate is up when the train is in the crossing.

From this formal definition of hazards and formal model of the system we can attempt a purely mathematical safety analysis. There are two obvious strategies: one is to determine whether, starting from some given initial state, a hazard is *reachable* by executing the net; the other is to start from a hazard state and run the Petri net backward to see whether the given initial state could have been a precursor of the hazard (and, if not, the hazard is avoided).

For a complex system brute force application of either method might be computationally intractable. The authors of [18] suggest a strategy that is computationally cheaper. One first does a relatively cheap search for certain "critical" states that represent key decision points: from a

[2]Strictly speaking, this interpretation makes sense only when there can never be more than one token on any place.

[3]Again, this discussion makes strict sense only if there is never more than one token on each place. The rules concerning multiple tokens per place and multiple arrows between a place and a transition are straightforward.

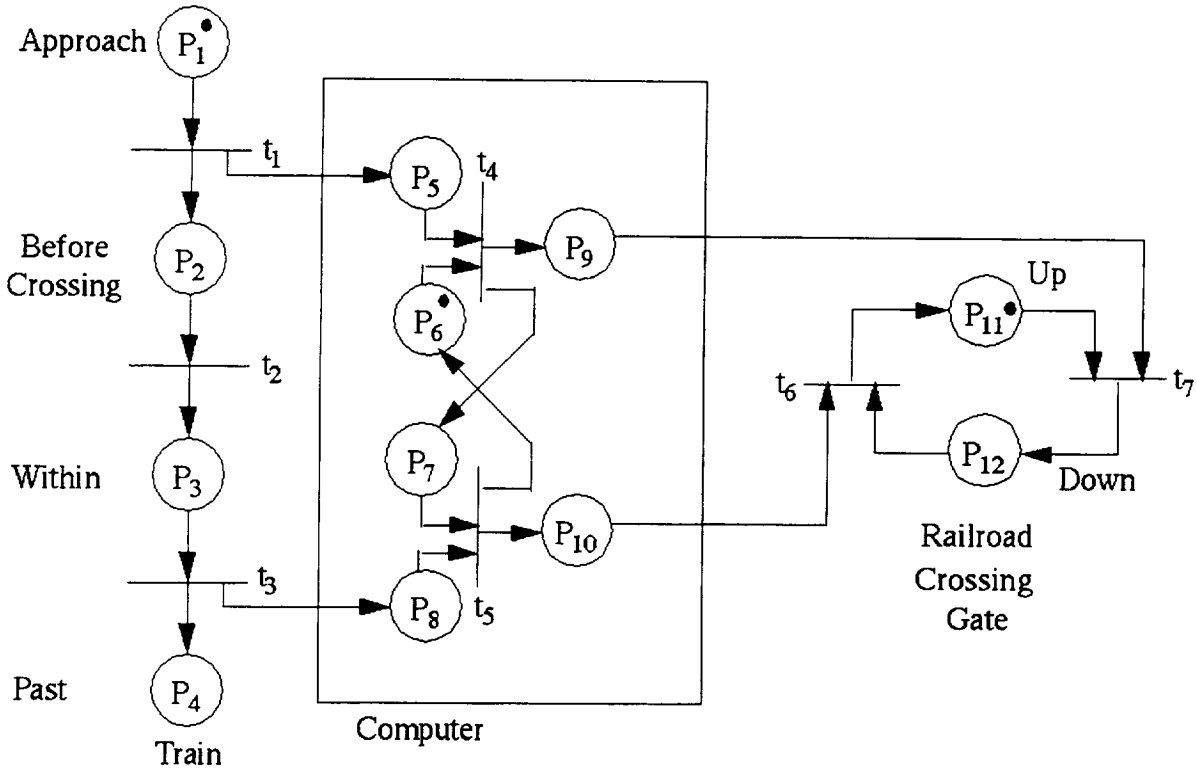


Figure 6.4: A Petri net model of a rail-crossing.

critical state it is possible for the system either to make a transition to either a safe or an unsafe state. For example, if the system is in the state $s = P_2P_7P_9P_{11}$ it can take two transitions:

- Transition t_2 results in the hazardous state $P_3P_7P_9P_{11}$—hazardous because P_3 means that the train is in the crossing and P_{11} means that the gate is up.
- Transition t_7 results in the state $P_2P_7P_{12}$, which is completely safe—it is non-hazardous and cannot lead to a hazardous state.

This makes s a critical state. To remove the hazard from our design, we can add to the design an interlock giving preference to t_7 over t_2.

This strategy has strengths and weaknesses. The strengths:

- It is (relatively) computationally cheap.
- It can be used to help *develop* a safe design, and not merely to check a completed design for safety.

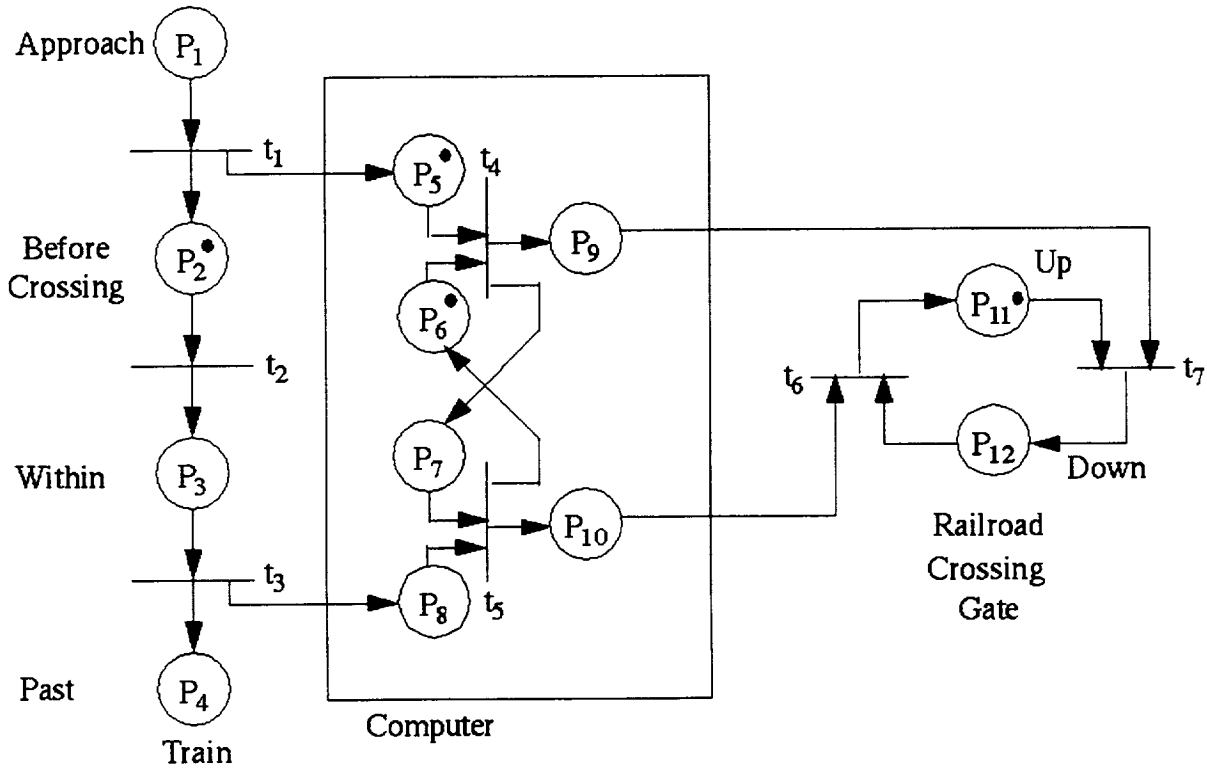


Figure 6.5: A different marking of the Petri net.

The weaknesses:

- It is conservative. The strategy may identify critical states that are not actually reachable and, as a result, add features to the design for the sake of a nonexistent danger.
- The revised design, with its interlocks, is purely abstract and may not be straightforwardly implementable. (How do you build the interlock that keeps the train from entering the crossing until the gate is down? One cannot do so directly. One would have to build a more complex model incorporating a mechanism to trigger an emergency brake if a train threatened to enter the crossing before the gate was lowered.)

Timed Petri Nets

The Petri net formalism just described models the sequencing of states and transitions, but not their timing. Various real-time extensions have been proposed. The authors of [18] adopt one in which each transition t is associated with two numbers Min_t and Max_t, between 0 and ∞, representing the minimum and maximum transition times for t. Intuitively, t cannot fire unless

it has been continuously enabled for at least Min_t and it must fire once it has been continuously enabled for time Max_t. (To be sensible, one requires that $Min_t \leq Max_t$.)

For example, safety of the railway crossing control system modeled by the Petri nets in Figures 6.4 and 6.5 may rely on a timing property: it always takes longer for a train to cross the approach sector than it does for the crossing gate to be lowered. The untimed Petri net model of the system is unsafe because it does not capture this property, which would naturally be captured in a timed model.

General analysis of these timed models seems difficult. Instead, the authors incorporate them into the strategy already described: One first ignores the timing information and identifies the critical states just as before; but now a greater repertoire of options is available for designing around the hazards. In particular, one can add to the design (or find already present in the design) timing constraints that will force the system down a safe path. For example, it may suffice to add a watchdog timer that will put the system into a safe state if it is taking too long for some transition to fire.

Petri Nets with Failures

One can also generalize Petri nets in order to model failures, by adding to the model a special set of failure transitions (distinct from the normal ones) and a special set of failure places (distinct from the normal ones). The output place of a failure transition is a failure place. A failure place might represent, for example, the failure of a processor.

In this formalism it is possible to model a number of important ideas about faults and fault-tolerance. For example, a system is *fail-safe* (for some particular failure) if no execution that proceeds from that failure transition arrives at a hazardous state. Based on engineering judgements about the worst hazards and the most likely failures, this model can be used to design for safety in the presence of various kinds of failures.

6.2.5 Other Means of Formal Safety Analysis

Two questions arise for formal safety analysis of the kind we are considering: Are some representations of state machines more helpful than others? What other analysis techniques might be helpful?

Representing State Machines

Consider again the crossing guard system represented by the Petri net of Figure 6.4. Note one way in which the Petri net representation is economical. Even if we restrict attention to markings that put at most one token on each place, the 12 places can have 2^{12} different markings.[4] Thus this small diagram represents a state machine that could have as many as 2^{12} states, which is much

[4] Not all of these states are reachable from any given initial state.

more compact than the "garden variety" representation of a state machine in which each of these states would be represented by a circle of its own. Here is a case in which the exponential function is working in our favor.

Looking more closely, the system "really" consists of the interaction of three components: the train, the controller, and the gate. There is always precisely one token on the places $P_1, \ldots P_4$ (which represent states of the train) and precisely one token on the places P_{11} and P_{12} (which represent states of the gate). The markings of the controller are more complex. One weakness of the Petri net formalism as a specification notation is the lack of a built-in mechanism for this kind of decomposition. (That weakness is irrelevant if the Petri net is merely the "compiled form" of a model initially described in a notation that is more highly structured.)

Thus the economy of this Petri net representation can largely be achieved by any formalism that explicitly decomposes the problem into three communicating components, each of which is a state machine. The one seeming advantage for Petri nets is representation of the controller, which does have fewer places than the controller has reachable states. On the other hand, this "more economical" representation can be obscure—the rules for firing transitions amount to a rather opaque notation for writing programs.

Analyzing State Machines

Reachability We may regard the Petri net as simply one implementation technique for permitting analysis of the *reachability graph* of the state machine it represents. The reachability graph can be thought of as the directed graph that results from "unrolling" execution of the state machine. Figure 6.6, taken from Figure 3 of [18] shows the reachability graph of the Petri net model from Figure 6.4. Each node in the reachability graph is a state of the system (in the Petri net model, that means a marking of the graph) and each arrow is a transition. Execution paths are paths through this graph. For pedagogical purposes the nodes corresponding to hazardous states have been drawn as rectangles.

In general, the reachability graph may be enormous. The critical-states trick described in section 6.2.4 is a way of generating useful information by looking only at small pieces of the graph. This trick can be implemented on top of any formal state-machine model—in particular, for RSML or StateCharts (see section 6.3). We only need to be able to generate the set of successor states and the set of predecessor states of any given state. This computation is easy for Petri net models and more complex for the RSML or StateCharts models described in section 6.3.

Model-Checking

Model checking consists of clever ways to make brute force techniques for analyzing reachability graphs more efficient and tractable. Symbolic model checking varies these techniques by incorporating symbolic manipulations roughly analogous to those used in symbolic algebra packages. When successful, model checkers permit analysis of a richer collection of properties than those expressible in the form "state s is never reached." For example, model checking can analyze prop-

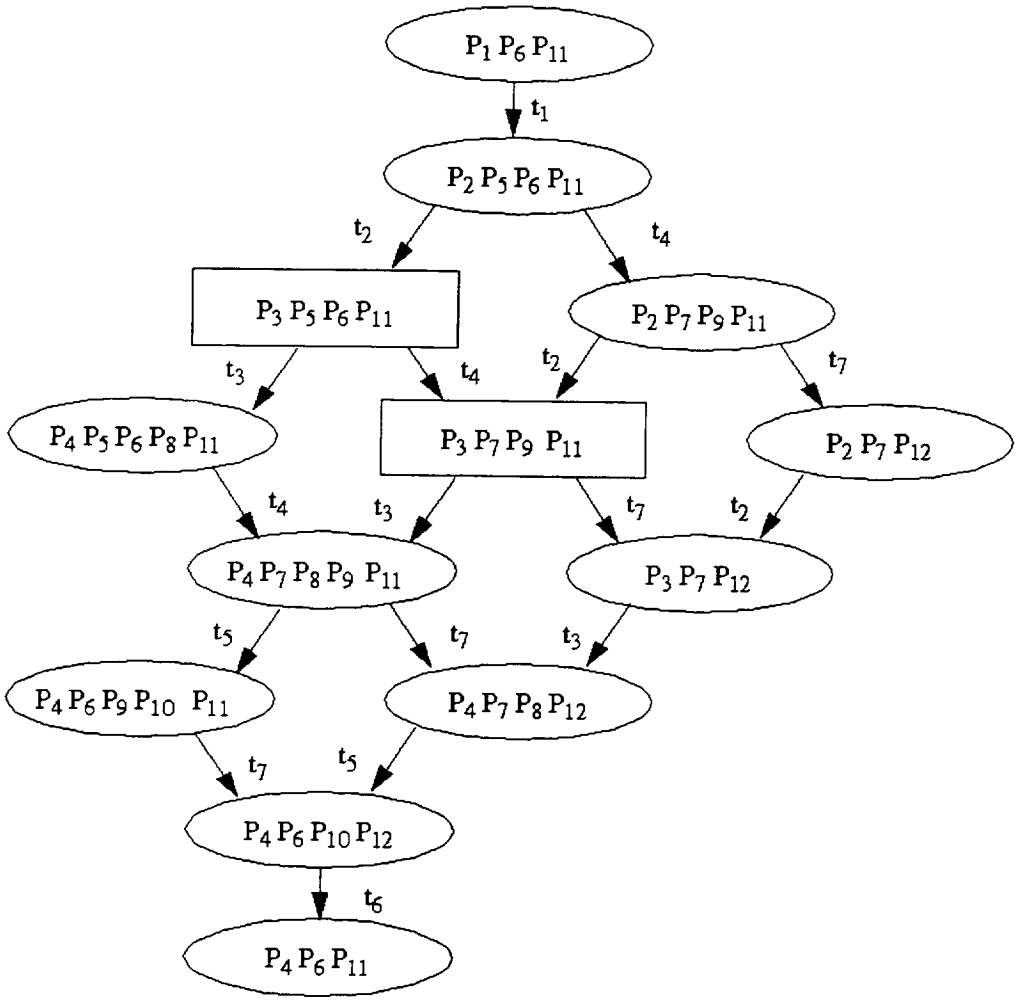


Figure 6.6: A reachability graph.

erties of the sequencing of events, such as "between every request and receipt there must be an acknowledgement." The literature on model checking and symbolic model checking is large [2, 1].

The symbolic model checking method of [1] can represent the timed Petri nets described above and analyze such timing properties as "the system will never spend more than t seconds continuously in state s." It is an open question how well this method will scale up.

6.3 Requirements Specification

The Requirements State Machine Language (RSML) is a specification notation that allows a user to describe a system as a collection of communicating state machines. The basic problem it addresses

is the problem of scale: how to write a large, complex, realistic specification in an intelligible way. The paper [17] describes applying RSML to specifying TCAS II, the airborne collision alert system for civilian aircraft; and [9] describes some techniques for automatically checking RSML specifications for certain completeness and consistency properties.

The TCAS specification gives RSML a great deal of credibility: starting as a shadow project, the RSML specification work outperformed the "ordinary" process so well that, part way through the experiment, it became the official standard. In particular, the designers of RSML claim a good degree of success in making the specifications accessible to pilots, external reviewers, programmers, etc. This is the most serious example we know of applying formal descriptive techniques to a complex reactive system.

The underlying mathematical model of RSML is state machine that can perform basic actions of the following kind: receive an *event* that triggers a transition, resulting in a new state and (possibly) the generation of more events. In particular, the environment in which TCAS operates can itself be modeled by such machines, which provides a way to record all the assumptions about the operating environment of TCAS.

Different kinds of specification information are written in different formats: graphs, tables, and text. The graphical notation, depicting states and transitions between them, is based largely on the StateCharts notation of Harel [7]. Once again the problem is scale: the authors estimate that, if written out as a single conventional state machine, a model of TCAS II would contain 10^{40} states. The notation makes this manageable by organizing the graphical representation of a state machine in certain hierarchical ways—in particular, using the notions of *superstate* and *parallel states*. As a result, the specification uses only about 100 states.

Figure 6.7 (taken from Figure 5 of [17]) shows the representation of a superstate. In this example, the superstate Q contains two substates, R and S. When a machine is in state Q it is also in (precisely) one of the substates R and S. (This is sometimes expressed by saying that Q is OR-decomposed into R and S.) This representation allows certain economies in describing the transitions of the machine. For example, transition C, represented by an outgoing arrow from the *boundary* of the superstate Q, is a transition *from* Q—that is, a transition that takes place regardless of which of its substates Q is in. Without the grouping of R and S into Q, we would have to represent the meaning of C by two arrows, one from R and one from S. Transition D, by contrast, is a transition *from* S, and cannot be taken from substate R.

Figure 6.8 (taken from Figure 6 of [17]) shows the representation of parallel states. In this case, state S is decomposed into the four states A, B, C, and D (which, in this picture, are themselves superstates). Being in state S means being simultaneously in each of these states, which is sometimes expressed by saying that S is the AND-decomposition of A, B, C, and D. Since the four parallel states are themselves superstates, being in S means being in exactly one of the (unnamed) circles from each of the four states. Now a transition *from* S can be described with one arrow instead of $3 \times 3 \times 2 \times 4$ (i.e., one for each possible combination of circles).

Each transition arrow is associated with three things: a triggering event, a condition (which must

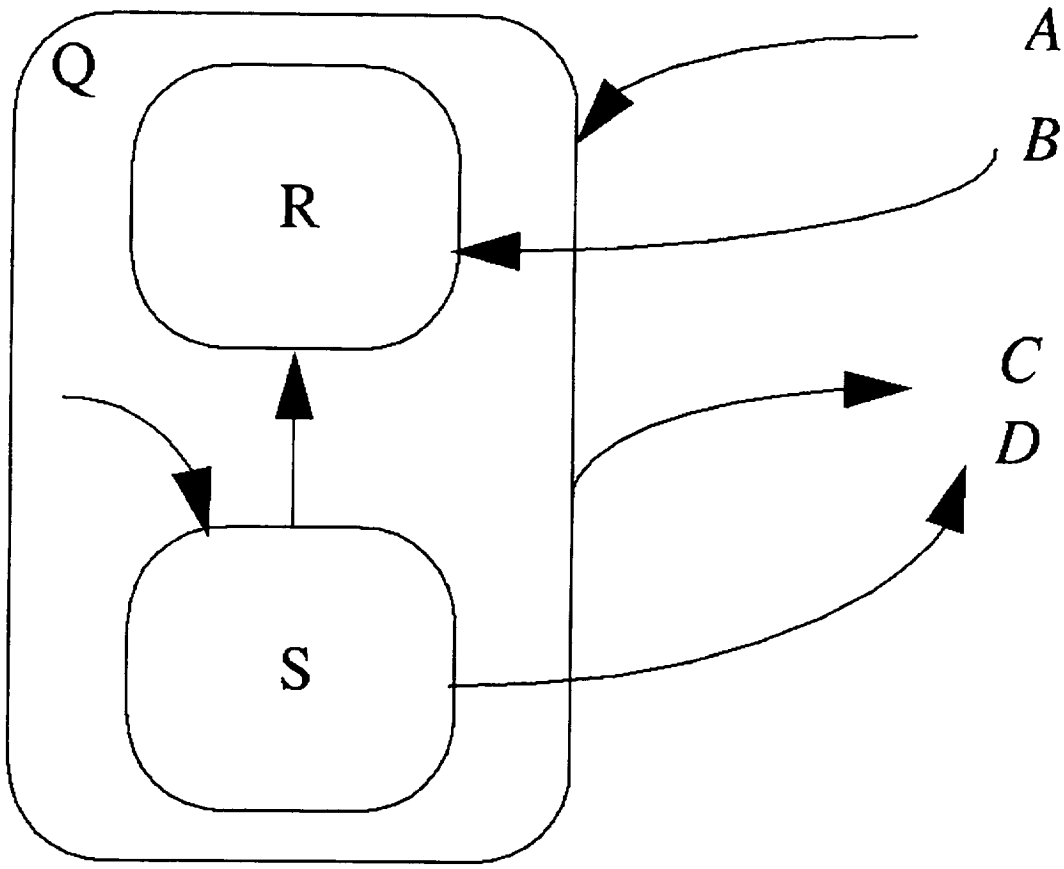


Figure 6.7: A superstate.

hold if a triggered transition is actually to occur), and the events (if any) it generates.[5] In the StateCharts notation this information is written directly beside each arrow. For the TCAS specification that proved impractical, so in RSML the events and conditions associated with transitions are separately specified in a format that is partly textual (name of transition, source and destination states, triggering event, etc.) and partly tabular (the condition).

The top level of an RSML description is a set of components. Each component is a state machine described in the hierarchical notation of superstates and parallel states. The events generated by components are of two kinds, internal and external. An internal event generated by some transition of component C is broadcast within C, and can trigger transitions of C, but is not visible outside C. (Internal events are used to regulate the ordering of actions within a component.) External events represent the sending and receiving of messages between components. The SEND(msg) event triggers a corresponding RECEIVE(msg) event in the destination component, but SEND and

[5] In the language of tables that we have been using in other parts of the project, the condition for a transition corresponds to the engagement criterion for a particular operational procedure.

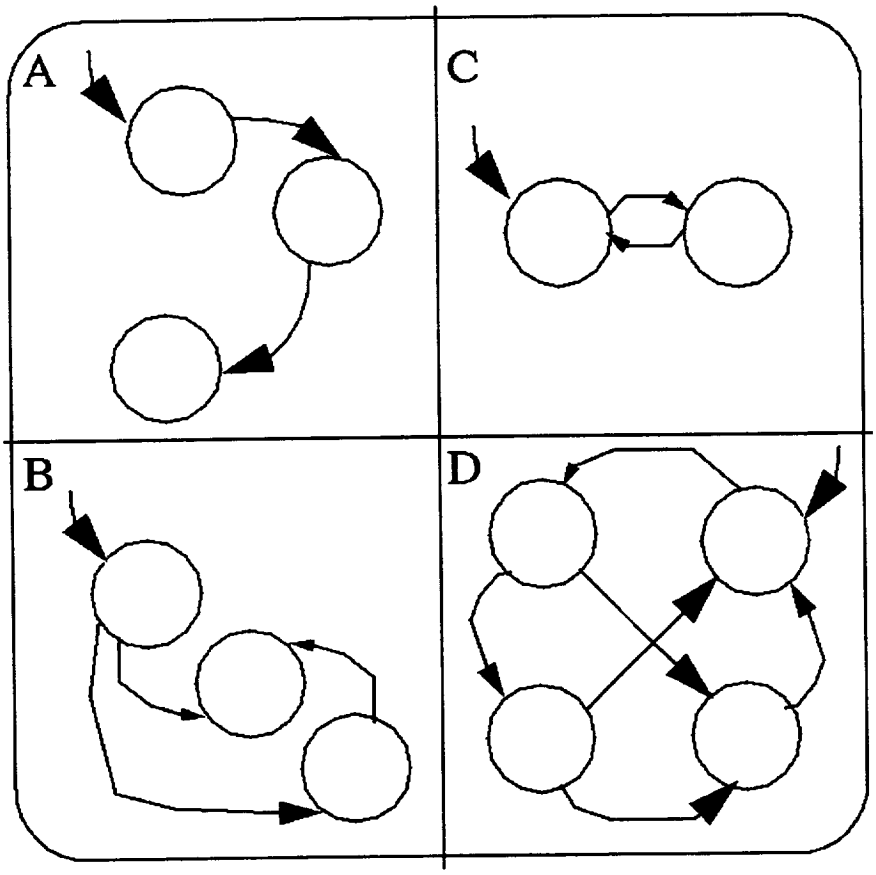


Figure 6.8: A parallel state.

RECEIVE are not assumed to be simultaneous. Typically, a RECEIVE event triggers a transition that causes a state change: certain input variables of the receiving component are updated with information contained in the message and the transition may generate internal events as well. Interfaces between components, which document the effects of these send and receive transitions, are economically described by a special-purpose RSML construct.

Experience with the TCAS project led the authors to diverge from the StateCharts notation in various ways. We have already noted that certain information-labels that StateCharts attaches to the graphical arrows are represented separately in RSML. In addition, the RSML notation makes heavy use of redundant information that supplies convenient cross-referencing. For example, an identifier I that denotes a variable defined on page 32 is always written as I_{v-32}, the subscript v serving as a reminder that I is a variable and the 32 as a reminder of where the definition can be found. Surprisingly, some of the more sophisticated notations of StateCharts that support information hiding have been removed because their use was found to cause more problems than they solved.

There is one substantial semantic difference between RSML and StateCharts, which lies in the definition of a single "execution step" for a single component. The two definitions of execution step have general similarities: some external events arrive at the boundary of a component; these external events may trigger internal transitions, generating events that may trigger other internal transitions, etc.; the resulting "step" of the component—which is atomic from the external point of view—is completed when all such possibilities for internal transitions are exhausted. The difference lies in the definitions of how the internal transitions cascade. The authors of [17] claim that RSML's definition of an execution step, though complex, is somewhat more intuitive than the definition in StateCharts. However, the definition of step used in StateCharts does have one virtue that the RSML definition lacks: it is guaranteed to terminate.

6.3.1 Tools for RSML

Leveson et al. have, or have under development, a number of prototype tools for RSML, including a simulator, tie-ins with fault-tree analysis and other kinds of safety analysis, and tools for doing consistency checking.

The consistency-checking tools, described in [9], have much in common with Tablewise. An RSML specification describes, potentially, a very big state machine, which could in principle be defined by a table. The consistency-checking problem amounts to checking whether this very big table is exhaustive and exclusive.

The strategy is to define the state machine in a highly structured way so that its checking can be factored into the checking of lots of smaller pieces. The natural pieces to consider are the definitions of individual transitions. The idea is to restrict the definitions of individual transitions in such a way as to guarantee that they will be well-behaved when put together.

Here are two illustrations of possibilities that complicate the composition of transitions, and must be ruled out (if this particular kind of local analysis is desired).[6]

Example 1 Suppose that the event e can trigger both a transition from state A to state B (enabling condition $x > 0$) and a transition from state A to state C (enabling condition $x < 2$). If e occurs when $x = 1$ both transitions are enabled, but they cannot both be taken. In the RSML semantics this is regarded as a nondeterministic choice. The nondeterminism complicates the analysis, so the analysis tool must forbid it. The user must change the conditions so that an event enables at most one transition from each state.

Example 2 Suppose that A and B are parallel states. We consider an example that introduces undesired complexity into the definition of taking simultaneous transitions out of both states. We have

[6] The underlying mathematics is not well explained in [9]. The interested reader should instead consult the Ph.D. thesis [8].

- event e can trigger a transition $A \rightarrow A'$ subject to the condition that "the system is also in state B"
- event e can trigger a transition $B \rightarrow B'$ subject to the condition "true"—that is, the transition is always enabled and e can always trigger it

The analysis would represent simultaneous transitions via interleaving. That is, it would represent the "simultaneous" transition from both of the parallel states as a result *either* of evaluating the transition $A \rightarrow A'$ followed by transition $B \rightarrow B'$, *or* of evaluating transition $B \rightarrow B'$ followed by transition $A \rightarrow A'$. For this to work the two choices must lead to the same result (and must lead to the same result as a simultaneous transition). If we choose the first order, then after evaluating the two transitions we are jointly in states A' and B'. If we choose the second order, the effect of transition $B \rightarrow B'$ is to make the condition on the transition from A false (because the system is no longer in state B). So the second transition is a no-op and we wind up jointly in the states A and B', a different result. The analysis tool must forbid this pair of transitions.

Chapter 7

Conclusions and Further Work

Our work has introduced four significant advances in decision table methodology:

- structural analysis of decision tables;
- partitioned decision tables;
- assertion tables and generation of decision tables from assertion tables;
- preconditions for decision tables.

We believe that the idea of decision tables with behaviors as objects, if properly developed, and our work on generating testable code, if properly finished, have the potential to be important advances as well.

The following list describes a number of directions in which the work described in this report could be pursued.

- Work out a form of structural analysis that applies to partitioned decision tables and decision tables with preconditions. (In fact, we have worked out a preliminary algorithm for such a form of structural analysis.)
- Follow up the idea that a decision table with behavior can represent an object. Use this idea to forge stronger connections with state machine or statechart based specification approaches.
- Deepen the connection with formal program development methods begun by our methods of generating decision tables from assertions.
- Sherry [25] associates with decision tables a specification development methodology with the following non-formal refinement path.

mission (entire table) → opprocs → scenarios
→ behaviors → dependent missions (tables)

Can we provide further formal support for this methodology?

- Strengthen the Tablewise logic module so as to support (at least) linear arithmetic. Doing so would allow it to support the following.
  - Automatic detection of mathematical relations between table entries.
  - Automatic extraction of test cases from test categories in fairly general situations.
  - Analysis of Parnas-style tables, where table entries are predicates.
- Explore methods for automatically simplifying decision tables and code generated from decision tables in the context of a precondition.
- Decision tables can be regarded as a means of describing transitions of state machines using logical conditions expressed in tabular form. Petri nets can be looked at as a way of defining states and transitions of state machines using logic in graphical form (a node of a Petri net = a logical proposition). Can the two be usefully connected? Petri nets are also closely related to the programming notation of UNITY [4], which is associated with a simple and elegant temporal logic. Both Petri nets and UNITY specifications can be analyzed using techniques such as model checking. Can a fruitful connection be made with decision table methods?
- The underlying idea of decision tables, and more so of partitioned decision tables, is to provide a well-structured way of stating logical propositions. The structure assists both understanding by humans and analysis by automated means. Continue to exploit the idea of using logical analysis to assist both understanding and analysis. Although humans and computers have different abilities, we believe that on the whole something that is easier for humans to understand is easier to analyze efficiently using a computer.
- Implement generation of testable code and tests, as well as behaviors of decision tables, as described in Chapter 3 and Chapter 5.
- The work for this task has led to several major innovations in decision table methodology: partitioned decision tables, assertion tables, preconditions for tables. Because these are innovations, we need to promote their acceptance by using them for significant work on examples and by refining them as necessary to make them more understandable and usable by ordinary users.

Bibliography

[1] R. Alur, T.A. Henzinger, and P.-H. Ho. Automatic symbolic verification of embedded systems. In *Proceedings of the 14th Annual Real-time Systems Symposium*, pages 2–11. IEEE Computer Society Press, 1993.

[2] J.R. Burch, E.M. Clarke, K.L. McMillan, D.L. Dill, and L.J. Hwang. Symbolic model checking: 10^{20} states and beyond. *Information and Computation*, 98(2):142–170, 1992.

[3] R.W. Butler and G.B. Finelli. The infeasibility of quantifying the reliability life-critical real-time software. *IEEE Transactions on Software Engineering*, 19(1):3–12, January 1993.

[4] K. Mani Chandy and Jayadev Misra. *Parallel Program Design, A Foundation.* Addison-Wesley, New York, 1988.

[5] John Joseph Chilenski and Stephen P. Miller. Applicability of modified condition/decision coverage to software testing. Distributed by FAA.

[6] E. W. Dijkstra. *A Discipline of Programming.* Prentice-Hall, Englewood Cliffs, N.J., 1976.

[7] D. Harel. Statecharts: A visual formalism for complex systems. In *The Science of Computer Programming 8*, pages 231–274, 1987.

[8] Mats Per Erik Heimdahl. *Static analysis of state-based requirements.* PhD thesis, UC Irvine, Irvine, California, 1994.

[9] Mats Per Erik Heimdahl and Nancy G. Leveson. Completeness and consistency analysis of state-based requirements. In *Proceedings of the 17th international conference on software engineering*, April 1995.

[10] K. L. Heninger. Specifying software requirements for complex systems: New techiques and their applications. *IEEE Transactions on Software Engineering*, SE-6(1):2–18, 1980.

[11] D. N. Hoover. Three extensions of decision table syntax and semantics. Technical report, Odyssey Research Associates, Inc., Ithaca NY 14850-1326, March 1995.

[12] D. N. Hoover and Zewei Chen. Tbell: A mathematical tool for analyzing decision tables. NASA Contractor Report 195027, November 1994.

[13] D. N. Hoover and Zewei Chen. Tablewise: A decision table tool. In John Rushby, editor, *COMPASS '95*, 1995.

[14] Nancy G. Leveson. Software safety: Why, What, and How. *Computing Surveys*, 18(2):125–163, June 1986.

[15] Nancy G. Leveson. Software safety in embedded computer systems. *CACM*, 34(2):34–46, February 1991.

[16] Nancy G. Leveson, Stephen Cha, and Timothy Shimeall. Safety verification of Ada programs using software fault trees. *IEEE Software*, pages 48–59, July 1991.

[17] Nancy G. Leveson, Mats Per Erik Heimdahl, Holly Hildreth, and Jon Damon Reese. Requirements specification for process-control systems. *IEEE Transactions on Software Engineering*, 20(9):684–707, September 1994.

[18] Nancy G. Leveson and Janice Stolzy. Safety analysis using petri nets. *IEEE Transactions on Software Engineering*, SE-13(3):386–397, March 1987.

[19] John R. Metzner. *Decision Table Languages and Systems.* Academic Press, New York, 1977.

[20] Greg Nelson and Derek C. Oppen. Simplification by cooperating decision procedures. *ACM Transactions on Programming Languages and Systems*, 1(2):245–257, October 1979.

[21] John K. Ousterhout. *Tcl and the Tk Toolkit.* Addison-Wesley, Reading, Mass., 1994.

[22] David L. Parnas. Tabular representation of relations. Technical Report 260, McMaster University Communications Research Laboratory, Hamilton, Ontario, Canada L8S 4K1, October 1992.

[23] RCTA, Inc., 1140 Connecticut Ave. NW, Suite 1020, Washington, D.C. 20036. *Software Considerations in Airborne Systems and and Equipment Certification*, December 1990. Document No. RCTA/DO-178-B.

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

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

[26] Robert E. Shostak. A practical decision procedure for arithmetic with function symbols. *Journal of the Association for Computing Machinery*, 26(2):351–360, April 1979.

[27] J. M. Spivey. *The Z Notation: A Reference Manual.* Prentice-Hall, Englewood Cliffs, N.J., 2nd edition, 1992.

[28] A. John van Schouwen. The a-7 requirements model: Re-examination for real-time systems and an application to monitoring systems. Technical Report CRL Report No. 242, Communications Research Laboratory, Faculty of Engineer, McMaster University, Hamilton, Ontario, Canada L8S 4K1, February 1992.

[29] N. Wirth. Program development by stepwise refinement. *Comm. ACM*, 14:221–227, 1971.

Appendix A

Notes on Tablewise 2

During work on the predecessor to this task, we developed a decision table tool, eventually named Tablewise, and described in [12, 13]. Tablewise is composed of a window-based decision table editor (selection part only) and a logic module that performs basic completeness and consistency analysis of decision tables as well as structural analysis and naive generation of tree code. Analysis results are displayed in table form by the table editor module.

During this task, we have been implementing a revision of Tablewise, called Tablewise 2, which incorporates a number of the decision table extensions described in this report as well as making a number of other improvements in both algorithms and "system" aspects. In this chapter we summarize the changes made and the features to be added or not.

A.1 Decision Table Extensions in Tablewise 2

We intend that eventually Tablewise 2 will support *all* the decision table extensions described in this report. As a deeply revered writer has observed, however, with man not all things are possible; therefore the initial release of Tablewise 2 will support only the extensions that we consider the most significant, innovative, and necessary to demonstrate by use of a decision table tool that supports them.

Extensions to be Supported by Tablewise 2.

- Partitioned decision tables. The table editor will support only one-level as described in [12, 13].
- Table preconditions (*Precond, Illegal, Always, Never*).
- Assertion tables, testing of decision tables against assertions, generation of decision tables from assertion tables.

- Structural analysis for partitioned decision tables and tables with preconditions.

Extensions Not Included in the Initial Release of Tablewise 2.

- Behaviors, systems of related tables. (Single tables with behaviors will be included in a later release. Systems of tables need another design cycle, taking full account of the idea of object tables.)
- Generation of MC/DC testable code, test generation. We will probably include code generation in an early release, perhaps even in the initial release. For test generation, we will probably wait until we strengthen the logic module to extract test cases from test categories, which is a nontrivial task.
- Parnas-style decision tables permitting arbitrary predicates as entries.

A.2 System and Algorithm Considerations in Tablewise 2

- The logic module has been ported from SML to CAML Light, another ML dialect. CAML Light produces far smaller executables that will load faster and will run conveniently on PCs that have a normal amount of power by current standards.

  It also produces a form of intermediate code that can be compiled into an executable for any platform that has a CAML Light compiler (many Unix platforms plus PCs with DOS or Windows 3.1). That means we can support the logic module on many platforms by simply distributing the intermediate code.

- The table editor is written in Tcl/Tk [21], as it was for the original Tablewise. It should run on most Unix platforms, though we may have to either distribute source or support it on those platforms ourselves. We believe that porting the table editor to PCs with Windows will be straightforward but will require a significant amount of labor.
- Because we have had a number of requests from people interested in using the Tablewise logic module with a different interface, in Tablewise 2 we have coupled the table editor and the logic module less closely than in Tablewise 1. The logic module will be run in batch mode accepting input with the same syntax as the table editor uses to store tables. The table storage grammar will be included in the Tablewise 2 release, probably in a Lex/YACC form.
- Analysis algorithms will be much more flexible and adaptable than in Tablewise 1. Tablewise 2 will, in fact, economically perform several forms of analysis simultaneously, so the logic module will not need to be called so often.
- Because the analysis algorithms will be more flexible, others may want to adapt them to uses we have not thought of. To this end, source for the top level of the logic module will be made public.

REPORT DOCUMENTATION PAGE

Form Approved
OMB No. 0704-0188

1. AGENCY USE ONLY (Leave blank)	2. REPORT DATE	3. REPORT TYPE AND DATES COVERED
	April 1996	Contractor Report

4. TITLE AND SUBTITLE
Applications of Formal Methods to Specification and Safety of Avionics Software

5. FUNDING NUMBERS
C NAS1-20335
TA02
WU 505-64-50-03

6. AUTHOR(S)
D.N. Hoover, David Guaspari, and Polar Humenn

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)
Odyssey Research Associates, Inc.
301 Dates Drive
Ithaca, NY 14850-1326

8. PERFORMING ORGANIZATION REPORT NUMBER
TM-95-0091

9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)
National Aeronautics and Space Administration
Langley Research Center
Hampton, VA 23665-5225

10. SPONSORING/MONITORING AGENCY REPORT NUMBER
NASA CR-4723

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

12a. DISTRIBUTION/AVAILABILITY STATEMENT
Unclassified - Unlimited
Subject Category 62

12b. DISTRIBUTION CODE

13. ABSTRACT (Maximum 200 words)

This report treats several topics in applications of formal methods to avionics software development. Most of these topics concern decision tables, an orderly, easy-to-understand format for formally specifying complex choices among alternative courses of action.

The topics relating to decision tables include: generalizations of decision tables that are more concise and support the use of decision tables in a refinement-based formal software development process; a formalism for systems of decision tables with behaviors; an exposition of Parnas tables for users of decision tables; and test coverage criteria and decision tables. We outline features of a revised version of ORA's decision table tool, Tablewise, which will support many of the new ideas described in this report.

We also survey formal safety analysis of specifications and software.

14. SUBJECT TERMS	15. NUMBER OF PAGES
Formal specification, decision tables, Tablewise, assertion tables, Parnas tables, A-7 Specification method, code generation, testing, MC/DC test coverage, formal safety analysis, RSML	95
	16. PRICE CODE A05

17. SECURITY CLASSIFICATION OF REPORT	18. SECURITY CLASSIFICATION OF THIS PAGE	19. SECURITY CLASSIFICATION OF ABSTRACT	20. LIMITATION OF ABSTRACT
Unclassified	Unclassified	Unclassified	