Analyzing Tabular and State-Transition Requirements Specifications in PVS[1]

Sam Owre, John Rushby, and Natarajan Shankar
Computer Science Laboratory
SRI International
Menlo Park CA 94025 USA

Technical Report CSL-95-12
June 1995 (Revised April 1996)

[1]This work was sponsored by the Naval Research Laboratory under Contract N00014-92-C-2177, by NASA Langley Research Laboratory under Contract NAS1-20334, and by the Air Force Office of Scientific Research under contract F49620-95-C0044.

Abstract

We describe PVS's capabilities for representing tabular specifications of the kind advocated by Parnas and others, and show how PVS's Type Correctness Conditions (TCCs) are used to ensure certain well-formedness properties.

We then show how these and other capabilities of PVS can be used to represent the AND/OR tables of Leveson and the Decision Tables of Sherry, and we demonstrate how PVS's TCCs can expose and help isolate errors in the latter.

We extend this approach to represent the mode transition tables of the Software Cost Reduction (SCR) method in an attractive manner. We show how PVS can check these tables for well-formedness, and how PVS's model checking capabilities can be used to verify invariants and reachability properties of SCR requirements specifications, and inclusion relations between the behaviors of different specifications.

These examples demonstrate how several capabilities of the PVS language and verification system can be used in combination to provide customized support for specific methodologies for documenting and analyzing requirements. Because they use only the standard capabilities of PVS, users can adapt and extend these customizations to suit their own needs. Those developing dedicated tools for individual methodologies may find these constructions in PVS helpful for prototyping purposes, or as a useful adjunct to a dedicated tool when the capabilities of a full theorem prover are required.

The examples also illustrate the power and utility of an integrated general-purpose system such as PVS. For example, there was no need to adapt or extend the PVS model checker to make it work with SCR specifications described using the PVS `TABLE` construct: the model checker is applicable to ***any*** transition relation, independently of the PVS language constructs used in its definition.

PVS specification files for several of the examples used here can be downloaded from `http://www.csl.sri.com/pvs/examples/tables`; PVS itself is available at `http://www.csl.sri.com/pvs.html`.

Note: this revised edition of the report differs significantly from the draft issued in June 1995.

Contents

List of Figures

Chapter 1

Introduction

An obstacle to the transfer of formal methods technology, as embodied in tools such as PVS, is that there is rather little *method* in formal methods. Prospective users of PVS, say, are provided with a powerful tool for formal specification and analysis, but are given little guidance on how best to apply this capability to their individual problems.

On the other hand, substantial methodologies for system specification and refinement have developed in some application areas, but these have generally not been supported by mechanized formal analysis. Several of these methodologies derived from work at the U.S. Naval Research Laboratory (NRL) in the 1970s on software requirements for the A-7E aircraft [21,22]. Such methods include Parnas's "four variable method" [42,43], the Software Cost Reduction (SCR) method of NRL [11], the Consortium Requirements Engineering (CoRE) method of the Software Productivity Consortium [12] and, more distantly, Harel's Statecharts [14] and its derivatives such as Leveson's Requirements State Machine Language (RSML) [29]. These methods are intended for reactive systems—that is, systems that operate continuously and interact with their environment—and model system requirements and behaviors as the traces (i.e., sequences of system states, inputs, and outputs) of interacting state machines. Some of these methods (notably Parnas's and SCR and, in different forms, RSML and the decision tables of Sherry [39]) also stress the use of tables to specify functions and state transition relations.

These methods provide organizing principles, systematic checks for well-formedness of specifications and, in some cases, simulators. For example, Heitmeyer and others at NRL have developed a mechanized toolset that performs systematic checks for well-formedness of SCR specifications and also provides a simulator for these specifications [18, 20], while Heimdahl and Leveson have developed a checker for RSML [16,17] and Hoover and others at ORA have mechanized the decision tables used by Sherry [23]. As yet, however, these and other tools for reactive systems do not provide the kind of formal analysis that is feasible with a true verification system such as PVS; in particular, their well-formedness checks cannot decide conditions that require arbitrary theorem proving, and they cannot examine application requirements such as safety (invariant) and liveness properties.

In this report, we describe some modest enhancements recently implemented in PVS that allow it to represent various kinds of tables in a fairly natural manner, and to provide syntactic and semantic well-formedness checks for tabular specifications. We also show how PVS's model checking capabilities can be used decide certain properties of SCR-type state-transition specifications. We hope this description will serve three purposes.

- To provide some methodological guidance for those who are using PVS in application areas where tabular and state-transition specifications are appropriate.

- To demonstrate how the resources of a verification system with a rich specification language and a repertoire of automated proof procedures can be used in combination to provide automated assistance in novel domains. The capabilities of PVS that we exploit—namely, its powerful type system, higher-order functions, tables, decision procedures, and model checker—are all useful individually, while their combination provides effective automation for various kinds of tabular specification methods at negligible development cost. Because our treatment uses the standard capabilities of PVS, we hope that others will be able to modify and adapt it to suit their own purposes, or to use it to suggest ways of using PVS to automate other methodologies.

- To provide rapid prototyping and back-end support for those developing specialized tools such as NRL's SCR* toolset [18] and those for RSML [17]. For example, we hope that experimenting with SCR model checking in PVS will be useful to developers of the model checker planned for SCR*, and that the ability to call, when necessary, on the full theorem-proving capability of PVS will free them to provide really efficient and smooth support for the majority of well-formedness cases that do not require this capability. Some aspects of the TableWise tool [23] were prototyped in PVS in just this way.

The body of this report is contained in three chapters. Chapter 2 describes PVS's representation of Parnas-style tables, and its method for generating and discharging the proof obligations that ensure completeness and consistency of tabular specifications. Chapter 3 shows how Leveson's AND/OR tables and Sherry's decision tables can be represented in PVS. Chapter 4 combines the methods of the previous two chapters to provide a treatment for SCR-style specifications, and shows how PVS's model checker can be used to decide application properties of these specifications. Brief conclusions are provided in Chapter 5.

Chapter 2

Basic Tables

Tables can be a convenient way to specify certain kinds of functions. An example is the function $\mathit{sign}(x)$, which returns $-1, 0$, or 1 according to whether its integer argument is negative, zero, or positive. As a table, this can be specified as follows.

$$\mathit{sign}(x) = \begin{array}{|c|c|c|} \hline x < 0 & x = 0 & x > 0 \\ \hline -1 & 0 & +1 \\ \hline \end{array}$$

This is an example of a piecewise continuous function that requires definition by cases, and the tabular presentation provides two benefits.

- It makes the cases explicit, thereby allowing checks that none of them overlap and that all possibilities are considered.
- It provides a visually attractive presentation of the definition that eases comprehension.

The first of these benefits is a semantic issue that is handled in PVS by the `COND` construct; the second is a syntactic issue that is handled in PVS by the `TABLE` construct (which is a variation on `COND`).

2.1 The PVS `COND` Construct

The PVS `COND` construct provides for specification by cases. Its general form is

```
COND
 c₁  →  e₁,
 c₂  →  e₂,
 ...
 cₙ  →  eₙ
ENDCOND
```

where the c_i are Boolean expressions and the e_i are values of some single type. The keyword `ELSE` can be used in place of the final condition c_n. The construct can appear anywhere that a value of the type of e_i is expected. PVS requires that exactly one of the c_i is true and ensures this by generating two Type Correctness Conditions (TCCs) for each `COND`.

Disjointness requires that each distinct c_i, c_j pair is disjoint (i.e., $c_i \wedge c_j$ is false).

Coverage requires that the disjunction of all the c_i is true.

The coverage TCC is suppressed if the `ELSE` keyword is used; also the c_i, c_j component of the disjointness TCC is suppressed when e_i and e_j are syntactically identical. TCCs are proof obligations that must be discharged before the specification that generated them is considered fully typechecked. (PVS allows proof of these obligations to be postponed, but keeps track of all unsatisfied obligations.) Given that the TCCs are true, the `COND` is equivalent to, and is treated internally as, the following construction.

```
IF    c1 THEN e1
ELSIF c2 THEN e2
...
ELSE en
ENDIF
```

Notice that the c_n condition does not need to be checked in the `IF-THEN-ELSE` translation: if this was given as an explicit `ELSE` in the `COND`, then the "fall through" default is exactly what is required; otherwise, the coverage TCC ensures that c_n is the negation of the disjunction of the other c_i, and the "fall through" is again correct.

Using `COND`, we can specify the *sign* function as follows.

```
signs: TYPE = { x: int | x >= -1 & x <= 1}

x: VAR int

sign_cond(x): signs =
  COND
    x < 0  -> -1,
    x = 0  ->  0,
    x > 0  ->  1
  ENDCOND
```

This generates the following TCCs, both of which are discharged by PVS's default strategy for TCCs in less than a second. (In addition, subtype TCCs are generated to ensure that 0, for example, is a valid element of the type `signs`.)

```
% Disjointness TCC generated (line 10) for
% COND x < 0 -> -1, x = 0 -> 0, x > 0 -> 1 ENDCOND
  sign_cond_TCC2: OBLIGATION
      (FORALL (x: int):
         NOT (x < 0 AND x = 0)
             AND NOT (x < 0 AND x > 0) AND NOT (x = 0 AND x > 0));

% Coverage TCC generated (line 10) for
% COND x < 0 -> -1, x = 0 -> 0, x > 0 -> 1 ENDCOND
  sign_cond_TCC3: OBLIGATION (FORALL (x: int): x < 0 OR x = 0 OR x > 0);
```

The variant that uses the `ELSE` clause looks as follows.

```
  sign_cond2(x): signs =
    COND
      x < 0  -> -1,
      x = 0  ->  0,
      ELSE   ->  1
    ENDCOND
```

It generates a simpler disjointness TCC (since there is no third case to consider), and no coverage TCC.

```
% Disjointness TCC generated (line 12) for
% COND x < 0 -> -1, x = 0 -> 0, ELSE -> 1 ENDCOND
  sign_cond2_TCC2: OBLIGATION (FORALL (x: int): NOT (x < 0 AND x = 0));
```

Both of these `COND` are equivalent to the following `IF-THEN-ELSE` form.

```
  sign_traditional(x): signs =
    IF x < 0 THEN -1 ELSIF x > 0 THEN 1 ELSE 0 ENDIF
```

The equivalence is demonstrated by the following lemmas

```
  trad_cond_same:  LEMMA sign_traditional = sign_cond
  trad_cond2_same: LEMMA sign_traditional = sign_cond2
```

which can each be proved in less than a second by the PVS proof commands `(apply-extensionality :hide? t)(grind)`.[1]

Because `COND` is treated internally as an `IF-THEN-ELSE`, proofs involving `COND` are mechanized in exactly the same way as `IF-THEN-ELSE`—that is, by the commands `(lift-if)` and `(split)` or `(bddsimp)`, and the higher-level commands such as `(grind)` that use these.

[1]The `:hide? t` keyword argument is optional: it simply hides the original formula once extensionality has been applied, and thereby reduces visual clutter in the sequent.

2.2 The PVS `TABLE` Construct

PVS provides `TABLE` constructs that allow specification of one- and two-dimensional tables. These constructions provide a fairly attractive input syntax and are LaTeX-printed as true tables. Their semantic treatment derives directly from the `COND` construct.

2.2.1 One-Dimensional Vertical Tables

These are the simplest form of table in PVS. They simply replace the `->` and `,` of `COND` cases by `|` and `||`, respectively, and introduce each case with `|`; they also add a final `||` and change the keyword from `COND` to `TABLE`. The *sign* example is therefore transformed from a `COND` to the following `TABLE`.

```
sign_vtable(x): signs = TABLE
                  %-------------%
                  | x < 0 | -1 ||
                  %-------------%
                  | x = 0 |  0 ||
                  %-------------%
                  | x > 0 |  1 ||
                  %-------------%
ENDTABLE
```

Note that the horizontal lines are simply comments (comments are introduced by `%` in PVS). This specification is equivalent to that of `sign_cond` and generates exactly the same TCCs and is treated the same in proofs. Note that PVS remembers the syntactic form used in a specification and always prints it out the same way it was typed in; thus, the prover will print a table as a table, even though it is treated semantically as a `COND` (which is itself treated as an `IF-THEN-ELSE`). Of course, the special syntactic treatment is lost once a proof step (e.g., `(lift-if)`) has transformed the structures appearing in a sequent.

2.2.1.1 LaTeX-Printing Tables

The PVS LaTeX-printer understands tables and automatically generates the code necessary to print them as true tables.

sign_vtable(x) : signs = TABLE

$x < 0$	-1
$x = 0$	0
$x > 0$	1

ENDTABLE

2.2.1.2 Enumeration Tables

The tables we have seen so far involve general comparison operators in their conditions. A special case arises when the intent is simply to enumerate all values of some finite type. In such cases, equality is the only comparison operator used, as in the following example.

```
few_ints: TYPE = { x : int | x >= -2 & x <= 2}

sign_fewv(z:few_ints): signs = TABLE
               %--------------%
               | z = -2 | -1 ||
               %--------------%
               | z = -1 | -1 ||
               %--------------%
               | z =  0 |  0 ||
               %--------------%
               | ELSE   |  1 ||
               %--------------%
ENDTABLE
```

Here we are defining a specialized ***sign*** function by enumeration over a type consisting of just the integers from -2 to +2, The `z =` appearing in each case is repetitive, so PVS allows us to factor it out as follows.

```
sign_fewv_enum(z:few_ints): signs = TABLE
                 z
               %----------%
               | -2 | -1 ||
               %----------%
               | -1 | -1 ||
               %----------%
               |  0 |  0 ||
               %----------%
               |ELSE|  1 ||
               %----------%
ENDTABLE
```

When an identifier (here `z`) follows the `TABLE` keyword, the first column is implicitly a list of values for this identifier, and the individual entries are treated as `identifier = value`.

2.2.1.3 Data Type Tables

A special case of enumeration tables arises when the values are the constructors of an abstract data type (ADT); this most commonly arises with enumeration types (which are implemented as degenerate ADTs in PVS), such as the following.

```
modes: TYPE = { off, armed, engaged }

value(m:modes):bool = TABLE
          m
        %------------------%
        | off     | false ||
        %------------------%
        | armed   | true  ||
        %------------------%
        | engaged | true  ||
        %------------------%
ENDTABLE
```

PVS recognizes this case specially and treats the `TABLE` internally as an ADT `CASES` construct, rather than as a `COND`. This has no semantic significance, but it allows more automated theorem proving to be used, and it allows the check for disjointness and coverage to be performed at typecheck-time (so the TCCs are not generated). Thus, the example above is semantically equivalent to the following form, which does generates TCCs and translates into the `COND` form.

```
value_alt(m:modes):bool = TABLE
        %----------------------%
        | off?(m)     | false ||
        %----------------------%
        | armed?(m)   | true  ||
        %----------------------%
        | engaged?(m) | true  ||
        %----------------------%
ENDTABLE

same: LEMMA value = value_alt
```

The lemma can be proved by `(apply-extensionality :hide? t)` and `(grind)`.

2.2.2 One-Dimensional Horizontal Tables

Horizontal tables are semantically identical to vertical tables, but use a slightly different syntax to notify PVS that the information is being presented in a different order. The first delimiter after the `TABLE` keyword must be `|[` rather than the simple `|`, and the final delimiter on the first row is `]|` rather than `||`. For example, here is the ***sign*** function presented as a horizontal table.

```
sign_htable(x): signs = TABLE
               %-------------------%
               |[ x<0 | x=0 | x>0 ]|
               %-------------------%
               |  -1  |  0  |  1  ||
               %-------------------%
ENDTABLE
```

The `ELSE` keyword can be used just as with vertical tables.

```
sign_htable2(x): signs = TABLE
               %--------------------%
               |[ x<0 | x=0 | ELSE ]|
               %--------------------%
               |  -1  |  0  |   1  ||
               %--------------------%
ENDTABLE
```

The PVS LaTeX-printer deals with these tables properly.

sign_htable2(x) : signs = TABLE

$x < 0$	$x = 0$	ELSE
-1	0	1

ENDTABLE

Horizontal enumeration tables are treated similarly to vertical ones, except that the enumerated identifier must follow a comma (because horizontal tables are actually a species of two-dimensional table).

```
sign_fewh_enum(z:few_ints): signs = TABLE ,
       %----------------------------------%
     z |[   -2   |   -1   |   0   | ELSE ]|
       %----------------------------------%
       |    -1   |   -1   |   0   |   1  ||
       %-----------------------------------%
ENDTABLE
```

2.2.3 Two-Dimensional Tables

These are similar to one-dimensional horizontal tables, except that there can be more than two rows, and the first row has one less column than the rest. Semantically, two-dimensional tables are treated as nested `COND` (or `CASES`) constructs; more particularly, the columns are nested within the rows. Here is a trivial example.

```
 example(state,input): some_type = TABLE
     state,  input
           %--------%
           |[ x | y |]
     %--------------%
     |  a  | p | q ||
     %--------------%
     |  b  | q | q ||
     %--------------%
ENDTABLE
```

This translates to the following.

```
COND
  state = a -> COND input = x -> p,  input = y -> q ENDCOND,
  state = b -> COND input = x -> q,  input = y -> q ENDCOND
ENDCOND
```

Notice that this translation causes disjointness and coverage TCCs for the columns to be generated several times—once for each row. For example, the coverage TCCs generated for the two inner `COND`s above have the following form.

```
 coverage_a: OBLIGATION state = a IMPLIES input = x OR input = y
 coverage_a: OBLIGATION state = b IMPLIES input = x OR input = y
```

These appear redundant, so we might be tempted to use the following translation instead.

```
LET
  x1 = COND input = x -> p, input = y -> q ENDCOND,
  x2 = COND input = x -> q, input = y -> q ENDCOND
IN
  COND state = a -> x1, state = b -> x2 ENDCOND
```

This generates the following single, simple coverage TCC for the columns.

```
 coverage_TCC: OBLIGATION input = x OR input = y
```

The problem with this translation is that there may be subtype TCCs generated from the terms corresponding to `p` and `q` that must be conditioned on the expressions corresponding to `a` and `b` in order to be provable. Here is an example due to Parnas [32, Figure 1] that illustrates this.

```
sqrt: [nonneg_real -> nonneg_real]

Parnas_Fig1(y,x:real):real = TABLE
          %--------------------------------------------------%
          |[ 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 ||
   %---------------------------------------------------------------%
ENDTABLE
```

The subtype constraint on the argument to the *sqrt* function generates TCCs in the second and third rows that are provable only when the corresponding row constraints are taken into account. The `LET` form translation loses this information. Therefore, PVS uses the simple nested `COND` translation—this sometimes leads to redundancy, but it generates the provable TCCs shown in Figure 2.1 (e.g., the TCCs numbered 2, 8, 11, and those numbered 3, 9, 12 are duplicative). These TCCs are all discharged in seconds by PVS's standard strategy for TCCs. In addition to the disjointness and coverage TCCs, there are subtype TCCs from the functions *sqrt* and exponentiation (indicated by `^`).

The LaTeX-printed form of this specification is as follows.

Parnas_Fig1$((y, x:$ real$))$: real = TABLE

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

ENDTABLE

```
% Subtype TCC generated (line 60) for 2
  Parnas_Fig1_TCC1: OBLIGATION
      (FORALL (x: real, y: real):
         x = 3 AND NOT y = 27 AND NOT y > 27 AND y < 27
             IMPLIES y /= 0 OR 2 >= 0);

% Disjointness TCC generated for
    % COND
    % y = 27 -> 27 + sqrt(27),
    % y > 27 -> 54 + sqrt(27),
    % y < 27 -> y ^ 2 + 3
    % ENDCOND
  Parnas_Fig1_TCC2: OBLIGATION
      (FORALL (x: real, y: real):
         x = 3
             IMPLIES NOT (y = 27 AND y > 27)
               AND NOT (y = 27 AND y < 27) AND NOT (y > 27 AND y < 27));

% Coverage TCC generated for
    % COND
    % y = 27 -> 27 + sqrt(27),
    % y > 27 -> 54 + sqrt(27),
    % y < 27 -> y ^ 2 + 3
    % ENDCOND
  Parnas_Fig1_TCC3: OBLIGATION
      (FORALL (x: real, y: real): x = 3 IMPLIES y = 27 OR y > 27 OR y < 27);

% Subtype TCC generated (line 62) for -(x - 3)
  Parnas_Fig1_TCC4: OBLIGATION
      (FORALL (x: real, y: real):
         NOT x = 3 AND x < 3 AND y = 27 IMPLIES -(x - 3) >= 0);

% Subtype TCC generated (line 62) for -(x - 3)
  Parnas_Fig1_TCC5: OBLIGATION
      (FORALL (x: real, y: real):
         NOT x = 3 AND x < 3 AND NOT y = 27 AND y > 27 IMPLIES -(x - 3) >= 0);

% Subtype TCC generated (line 62) for 2
  Parnas_Fig1_TCC6: OBLIGATION
      (FORALL (x: real, y: real):
         NOT x = 3 AND x < 3 AND NOT y = 27 AND NOT y > 27 AND y < 27
             IMPLIES y /= 0 OR 2 >= 0);

% Subtype TCC generated (line 62) for 2
  Parnas_Fig1_TCC7: OBLIGATION
      (FORALL (x: real, y: real):
         NOT x = 3 AND x < 3 AND NOT y = 27 AND NOT y > 27 AND y < 27
             IMPLIES (x - 3) /= 0 OR 2 >= 0);
```

Figure 2.1: TCCs Generated from Example Two-Dimensional Table (continues)

```
% Disjointness TCC generated for
    % COND
    % y = 27 -> 27 + sqrt(-(x - 3)),
    % y > 27 -> y + sqrt(-(x - 3)),
    % y < 27 -> y ^ 2 + (x - 3) ^ 2
    % ENDCOND
  Parnas_Fig1_TCC8: OBLIGATION
      (FORALL (x: real, y: real):
         NOT x = 3 AND x < 3
             IMPLIES NOT (y = 27 AND y > 27)
               AND NOT (y = 27 AND y < 27) AND NOT (y > 27 AND y < 27));

% Coverage TCC generated for
    % COND
    % y = 27 -> 27 + sqrt(-(x - 3)),
    % y > 27 -> y + sqrt(-(x - 3)),
    % y < 27 -> y ^ 2 + (x - 3) ^ 2
    % ENDCOND
  Parnas_Fig1_TCC9: OBLIGATION
      (FORALL (x: real, y: real):
         NOT x = 3 AND x < 3 IMPLIES y = 27 OR y > 27 OR y < 27);

% Subtype TCC generated (line 63) for x - 3
  Parnas_Fig1_TCC10: OBLIGATION
      (FORALL (x: real, y: real):
         NOT x = 3 AND NOT x < 3 AND x > 3 AND y = 27 IMPLIES x - 3 >= 0);

% Disjointness TCC generated for
    % COND
    % y = 27 -> 27 + sqrt(x - 3),
    % y > 27 -> 2 * y + sqrt(x - 3),
    % y < 27 -> y ^ 2 + (3 - x) ^ 2
    % ENDCOND
  Parnas_Fig1_TCC11: OBLIGATION
      (FORALL (x: real, y: real):
         NOT x = 3 AND NOT x < 3 AND x > 3
             IMPLIES NOT (y = 27 AND y > 27)
               AND NOT (y = 27 AND y < 27) AND NOT (y > 27 AND y < 27));

% Coverage TCC generated for
    % COND
    % y = 27 -> 27 + sqrt(x - 3),
    % y > 27 -> 2 * y + sqrt(x - 3),
    % y < 27 -> y ^ 2 + (3 - x) ^ 2
    % ENDCOND
  Parnas_Fig1_TCC12: OBLIGATION
      (FORALL (x: real, y: real):
         NOT x = 3 AND NOT x < 3 AND x > 3 IMPLIES y = 27 OR y > 27 OR y < 27);
```

Figure 2.1: TCCs Generated from Example Two-Dimensional Table (continues)

```
% Disjointness TCC generated (line 58) for
    % TABLE
    %   |[ 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
    %     ||
    %   ENDTABLE
  Parnas_Fig1_TCC13: OBLIGATION
      (FORALL (x: real):
         NOT (x = 3 AND x < 3)
             AND NOT (x = 3 AND x > 3) AND NOT (x < 3 AND x > 3));

% Coverage TCC generated (line 58) for
    % TABLE
    %   |[ 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
    %     ||
    %   ENDTABLE
  Parnas_Fig1_TCC14: OBLIGATION (FORALL (x: real): x = 3 OR x < 3 OR x > 3);
```

Figure 2.1: TCCs Generated from Example Two-Dimensional Table

2.2.4 Blank Entries

Some functions are not defined for all values of their arguments—for example, division is not defined when the divisor is zero. PVS is a logic of total functions, and does not admit such partial functions directly. However, because of the very precise typing provided by predicate and dependent types, functions that would be partial in simpler systems can be treated as total in PVS. For example, division in PVS is typed so that its second argument is a `nonzero_real`, and the function is total when its domain is accurately specified in this way. When specifying such a function by means of tables, however, it can be useful to explicitly (though redundantly) indicate "holes" in the domain by means of blank entries. This is particularly convenient for two-dimensional tables on dependent types, as will be illustrated later, but we will explain the idea with a one-dimensional example.

A standard "challenge" for specification languages is the partial function *subp* on the integers defined by

$$\mathit{subp}(i, j) = \textbf{if } i = j \textbf{ then } 0 \textbf{ else } \mathit{subp}(i, j+1) + 1 \textbf{ endif}.$$

This function is undefined if $i < j$ (when $i \geq j, \mathit{subp}(i, j) = i - j$) and it is argued that if a specification language is to admit this type of definition, then it must provide a treatment for partial functions [8]. PVS deals easily with this challenge by using dependent typing to specify that the second argument to the function must not exceed the value of its first argument.

```
    subp((i: int), (j: {x: int | x <= i})): nat
```

The function is total on this accurately specified domain, and can then be defined by means of a table as follows.

```
    subp((i: int), (j: {x: int | x <= i})): RECURSIVE nat =
     TABLE
           %-----------------------%
           | i=j |       0       ||
           %-----------------------%
           | i>j | subp(i, j+1)+1 ||
           %-----------------------%
     ENDTABLE
    MEASURE i - j
```

The coverage TCC generated from this specification is the following; it is proved trivially by the default strategy.

```
  subp_TCC5: OBLIGATION
      (FORALL (i: int, j: {x: int | x <= i}): i = j OR i > j);
```

This TCC shows that the "missing" case `i<j` does not need to be specified in the table because the types associated with `i` and `j` ensure that it can never arise. However, it may sometimes be desirable to make this fact visually explicit in the specification, and PVS allows blank entries to appear in tables for this purpose.

```
subp((i: int), (j: {x: int | x <= i})): RECURSIVE nat =
 TABLE
       %-----------------------%
       | i<j |                ||
       %-----------------------%
       | i=j |       0        ||
       %-----------------------%
       | i>j | subp(i, j+1)+1 ||
       %-----------------------%
 ENDTABLE
MEASURE i - j
```

Coverage TCCs are extended (if necessary) to ensure that blank entries are never encountered when evaluating such a specification. In this example, the TCC is identical to that of the previous specification without the blank entry.

Evaluation of tables (with or without blank entries) assumes that their TCCs have been discharged. For example, if we had incorrectly given the previous specification as

```
badsubp((i: int), (j: {x: int | x <= i})): RECURSIVE nat =
 TABLE
       %--------------------------%
       | i<j |        0          ||
       %--------------------------%
       | i=j |                   ||
       %--------------------------%
       | i>j | badsubp(i, j+1)+1 ||
       %--------------------------%
 ENDTABLE
MEASURE i - j
```

then we would obtain the following unprovable TCC.

```
badsubp_TCC4: OBLIGATION
    (FORALL (i: int, j: {x: int | x <= i}): i < j OR i > j);
```

If we ignore the TCC and try to prove the "theorem"

```
bang: CLAIM badsubp(3, 3) = 99
```

by expanding the definition of **`badsubp`**, we will obtain unpredictable behavior when we encounter the supposedly unreachable blank entry.

```
bang :
  |-------
{1}    badsubp(3, 3) = 99

Rule? (expand "badsubp")
Expanding the definition of badsubp, this simplifies to:

bang :
  |-------
{1}    (1 + badsubp(3, 4) = 99)

Rule?
```

In this case, PVS has applied the case for `i>j` in place of the missing case for `i = j`. This example reinforces the fact that PVS specifications are not guaranteed to be well-defined unless their TCCs have been discharged.

Blank entries may be used in conjunction with **`ELSE`** clauses. Recall that a coverage TCC is normally not required if an **`ELSE`** clause is given; this is not so when blank entries are present. For example, the specification

```
    subp((i: int), (j: {x: int | x <= i})): RECURSIVE nat =
     TABLE
           %------------------------%
           | i<j  |                ||
           %------------------------%
           | i=j  |       0        ||
           %------------------------%
           | ELSE | subp(i, j+1)+1 ||
           %------------------------%
     ENDTABLE
    MEASURE i - j
```

generates the following TCC

```
 subp_TCC4: OBLIGATION
   (FORALL (i: int, j: {x: int | x <= i}): i = j OR NOT (i < j OR i = j));
```

to ensure that the blank entry is inaccessible.

Strictly, blank entries are unnecessary in one-dimensional tables, since the entire case can always be omitted; they are extremely valuable, however, in two-dimensional tables. For example, Figure 2.2 reproduces the quotient lookup table

```
q(D, (P: bvec[7] | estimation_bound?(valD(D),valP(P)))): subrange(-2, 2) =
   LET
        a = -(2 - P(1) * P(0)),
        b = -(2 - P(1)),
        c = 1 + P(1),
        d = -(1 - P(1)),
        e = P(1),
        Dp      = bv2pattern(D),
        Ptruncp = bv2pattern(P^(6,2))
    IN
   TABLE Ptruncp, Dp
         |[ 000| 001| 010| 011| 100| 101| 110| 111]|
    %---------------------------------------------%
    |01010|    |    |    |    |    |    |    |  2 ||
    |01001|    |    |    |    |    |  2 |  2 |  2 ||
    |01000|    |    |    |    |  2 |  2 |  2 |  2 ||
    |00111|    |    |  2 |  2 |  2 |  2 |  2 |  2 ||
    |00110|    |  2 |  2 |  2 |  2 |  2 |  2 |  2 ||
    |00101|  2 |  2 |  2 |  2 |  2 |  2 |  2 |  1 ||
    |00100|  2 |  2 |  2 |  2 |  c |  1 |  1 |  1 ||
    |00011|  2 |  c |  1 |  1 |  1 |  1 |  1 |  1 ||
    |00010|  1 |  1 |  1 |  1 |  1 |  1 |  1 |  1 ||
    |00001|  1 |  1 |  1 |  1 |  e |  0 |  0 |  0 ||
    |00000|  0 |  0 |  0 |  0 |  0 |  0 |  0 |  0 ||
    |11111|  0 |  0 |  0 |  0 |  0 |  0 |  0 |  0 ||
    |11110| -1 | -1 |  d |  d |  0 |  0 |  0 |  0 ||
    |11101| -1 | -1 | -1 | -1 | -1 | -1 | -1 | -1 ||
    |11100|  a |  b | -1 | -1 | -1 | -1 | -1 | -1 ||
    |11011| -2 | -2 | -2 |  b | -1 | -1 | -1 | -1 ||
    |11010| -2 | -2 | -2 | -2 | -2 | -2 |  b | -1 ||
    |11001| -2 | -2 | -2 | -2 | -2 | -2 | -2 | -2 ||
    |11000|    |    | -2 | -2 | -2 | -2 | -2 | -2 ||
    |10111|    |    |    | -2 | -2 | -2 | -2 | -2 ||
    |10110|    |    |    |    |    | -2 | -2 | -2 ||
    |10101|    |    |    |    |    |    | -2 | -2 ||
    %---------------------------------------------%
    ENDTABLE
```

Figure 2.2: Quotient Lookup Table for an SRT Division Algorithm

from the PVS specification of an SRT division algorithm [36,41]. This specification generates 23 coverage TCCs to ensure that the blank entries can never be encountered. It is worth noting that the notorious Pentium **`FDIV`** bug, which is estimated to have cost Intel $500 million, was due to an SRT quotient lookup table, very similar to that of Figure 2.2, that had bad entries in a portion of the table that was incorrectly believed to inaccessible [34]. The TCCs of the PVS specification ensure that entries (indicated by blanks) that are believed to be inaccessible, truly are so; verification of the algorithm (which can be done largely automatically in PVS) then ensures that all the nonblank table entries are correct [36].

2.2.5 Variations

Parnas [32] advocates tabular specifications and introduces several kinds of tables for defining functions and relations; these have been given a formal definition by Janicki [26]. The PVS **`TABLE`** construct corresponds only to what Parnas calls a "normal" function table. However, other attributes of the PVS specification language allow specification of certain alternative kinds of tables.

For example, Parnas speaks of "vector" tables when defining a function whose value is a tuple, such as the following.

	$x < 0$	$x = 0$	$x > 0$
y	$x + 2$	$x + 4.21$	$5.4 + \sqrt{x}$
z	$5 + \sqrt{-x}$	$x - 4$	x

In this example from [33, Figure 1], the interpretation is that the value of the function is a pair, whose first and second components are represented by y and z, respectively.

Tuple types are directly available in PVS, so this function can be specified by simple tables. Both horizontal and vertical table formats for this example are illustrated here.

```
Vector_1(x:real): [real, real] =
  TABLE
     %-----------------------------------------------------------------%
     |[       x<0       |         x=0         |         x>0           ]|
     %-----------------------------------------------------------------%
     |(x+2, 5+sqrt(-x)) | (x+4+(21/100), x-4) | (5+(4/10)+sqrt(x), x) ||
     %-----------------------------------------------------------------%
  ENDTABLE

Vector_2(x:real): [real, real] =
  TABLE
     %         y                   z
     %---------------------------------------------%
     | x<0 | (x+2,                5+sqrt(-x))     ||
     %---------------------------------------------%
     | x=0 | (x+4+(21/100),       x-4)            ||
     %---------------------------------------------%
     | x>0 | (5+(4/10)+sqrt(x),   x)              ||
     %---------------------------------------------%
  ENDTABLE
```

Decimal notation is not supported in PVS so we have expressed the values 4.21 and 5.4 as fractions.

Because it is a higher-order logic with a rich type system, PVS can also deal uniformly with certain other kinds of tables that Parnas treats specially [32]. "Relation" and "predicate expression" tables, for example, are simply tables with range type `bool`. Thus, the following PVS specification is an example of what Parnas calls a "relation" table (from [32, Figure 4]).

```
rel(x,y,z:real):bool =
  TABLE
            %---------------------------------------------------------%
           |[ y>=0 & sqrt(y)<27 | y>=0 & sqrt(y)>=27 |     y < 0     ]|
     %----------------------------------------------------------------%
     | x=3  |    x^2+y^2 = z^2  |      x^2 = y^2     |      true     ||
     %----------------------------------------------------------------%
     | x<3  |        y^2 = z^2  |      x^2 = z^2     |      false    ||
     %----------------------------------------------------------------%
     | x>3  |        x^2 = z^2  |      x-z > 3       | x^2+y^2 = z^2 ||
     %----------------------------------------------------------------%
  ENDTABLE
```

PVS can easily establish that `(4, -3, 5)`, for example, is in the relation by using the strategy `(grind)` to prove the conjecture `rel(4, -3, 5)`. Similarly, `rel(4, 9,` `4)` and `rel(4,728, 4)` can be proved by `(grind)` plus elementary properties of the `sqrt` function.

Although the PVS `TABLE` construct can represent directly many of the kinds of tables introduced by Parnas [32], we have not found a convenient way to represent what Parnas calls “inverted” tables—but neither have we found a need for these,

In the next chapter, we consider rather different kinds of tables from those used by Parnas.

Chapter 3

AND/OR Tables and Decision Tables

In this chapter, we first consider a tabular representation for Boolean expressions that is quite different to any of Parnas's tables and that does not lend itself to the PVS `TABLE` construct either. We show how PVS can provide an adequate presentation of this kind of table using ordinary function application in a careful way. Then we combine a generalization of this approach with the `TABLE` construct to provide a treatment for a type of decision table that has been used for specifying avionics requirements.

3.1 AND/OR Tables

Leveson and her colleagues use a tabular representation for Boolean expressions [29] that is quite different from any of Parnas's tables. These AND/OR tables are most easily explained by means of an example. The following table describes some conditions under which a TCAS II avionics collision avoidance system should transition from the `Threat` state to the `Other-Traffic` state [29, Figure 32].

		OR			
	`Alt-Reporting`$_{s-202}$ **in state** `Lost`	T	T	T	-
A	`Bearing-Valid`$_{m-298}$	F	-	T	-
N	`Other-Range-Valid`$_{v-218}$ `= True`	-	F	T	-
D	`Proximate-Traffic-Condition`$_{m-317}$	-	-	F	-
	`Potential-Threat-Condition`$_{m-314}$	-	-	F	-
	`Other-Air-Status`$_{s-202}$ **in state** `On-Ground`	-	-	-	T

The idea is that each of the OR columns specifies one of the conditions under which the transition should be taken: the condition represented by a column is *true* if each of the expressions represented by those rows having a `T` in that column are *true*, and those having an `F` in the column are *false* (dashes indicate "don't care"). Thus, the condition represented by the first column is *true* when $\texttt{Alt-Reporting}_{s-202}$ is in state `Lost` *and* $\texttt{Bearing-Valid}_{m-298}$ is *false*. The conditions represented by the individual columns are disjoined (ORed together) to give the full set of conditions under which the transition should occur. Since the individual entries in each column are conjoined (ANDed together), the full AND/OR table is a structured presentation of a Boolean expression in disjunctive normal form (a disjunction of conjunctions). Leveson's AND/OR tables are quite effective for Boolean expressions that are conveniently expressed in disjunctive normal form; they are less so for expressions that are most naturally expressed in terms of implication, equivalence, or exclusive-or.

The `TABLE` construct of PVS is not well matched to the representation of AND/OR tables. We show how other constructs of PVS can be used to give an adequate representation for these tables. To describe the approach, we begin with the following simplified example of an AND/OR table.

		OR		
A	Expr_1	T	-	F
N	Expr_2	-	F	T
D	Expr_3	-	-	F

We can transpose this table to obtain the following equivalent representation.

	AND		
	Expr_1	Expr_2	Expr_3
O	T	-	-
R	-	F	-
	F	T	F

Written in this form, we can think of each row as a list of values (e.g., `(F, T, F)` in the case the bottom row) to be checked against the list of expressions `(Expr_1, Expr_2, Expr_3)`. Now, an existing construction in PVS that uses a list of expressions is function application: the arguments to a function application are written as a list of expressions. So we could hypothesize a function `X` that takes such a list as its arguments (e.g., `X( F , T , F )`) and returns *true* if `Expr_1` is `F` and `Expr_2` is `T` and `Expr_3` is `F`. Then we could write the table something like the following, which does have a fairly acceptable tabular layout.[1]

[1]In the PVS versions, we use `~` instead of `-` to indicate "don't care." This is because there is an inefficiency in PVS name resolution that is exponential in the number of overloadings (and `-` has

```
                                                                    1
      X( T , ~ , ~ ) OR
      X( ~ , F , ~ ) OR
      X( F , T , F )
```

We now need to consider the specification of `X`, and of `T`, `F`, and `~`. The bottom row of the table suggests that we might think of `T` and `F` as synonyms for *true* and *false*, respectively, and then `X` could be given as follows.

```
  X(x, y, z: bool): bool = Expr_1 = x AND Expr_2 = y AND Expr_3 = z
```

The trouble with this idea is that it does not extend to the "don't care" case: what truth value can we assign to `~`? A more sophisticated idea is to treat `T`, `F`, and `~` as the members of an enumerated type called `Extended_Bool` and to provide a function `cmp` that compares an `Extended_Bool` against a Boolean.

```
  Extended_Bool: TYPE = { T, F, ~ }

  cmp(e: Extended_Bool, b:bool): bool =
   CASES e OF
     T: b,
     F: NOT b,
     ~: TRUE
   ENDCASES

  X(x, y, z: Extended_Bool): bool =
        cmp(Expr_1, x) AND cmp(Expr_2, y) AND cmp(Expr_3, z)
```

The question now is: how do we supply values for the `Expr_`*i*? They must surely be the arguments to the predicate (called `Test`, say) whose behavior is defined by the specification in specification box 1 on page 25. We can establish this association by moving the definition of the function `X` inside a `LET` clause in the following definition of the function `Test`.

```
  Test(Expr_1, Expr_2, Expr_3: bool):bool =
    LET
      X(x, y, z: Extended_Bool): bool =
          cmp(x, Expr_1) AND cmp(y, Expr_2) AND cmp(z, Expr_3)
    IN
        X( T , ~ , ~ ) OR
        X( ~ , F , ~ ) OR
        X( F , T , F )
```

14 overloadings, whereas `~` has only two). This inefficiency will be eliminated in a future release of PVS.

Unfortunately, PVS does not at present allow the applicative kind of function definition inside a `LET` clause,[2] so we must define `X` with a `LAMBDA` as follows.

```
Test(Expr_1, Expr_2, Expr_3: bool):bool =
  LET
     X = LAMBDA (x, y, z: Extended_Bool):
        cmp(x, Expr_1) AND cmp(y, Expr_2) AND cmp(z, Expr_3)
  IN
      X( T , ~ , ~ ) OR
      X( ~ , F , ~ ) OR
      X( F , T , F )
```

Given this specification, PVS can easily prove conjectures about `Test` (e.g., `Test(FALSE, FALSE, TRUE)` is *true*) using the single command `(grind)`.

Following this model, we can construct a PVS rendition of the AND/OR table that was used to introduce this section (recall page 23). Notice how this PVS specification (for the predicate called `Transition` shown in Figure 3.1) builds the expressions `Alt_Reporting = Lost` and `Other_Air_Status = On_Ground` into the definition of the function `X`. We will see different ways to do this in the next section. The specification uses comments and careful layout to provide a tabular appearance, and to suggest the connection between the expressions in the definition of `X` and the columns of the table. The example conjecture `test` (which probes the second row of the table) is easily proved by the single command `(grind)`.

In the requirements specification method developed by Leveson and her colleagues [29], AND/OR tables are used to indicate the conditions under which state transitions should occur. The states and the transitions are specified separately, using Statechart-like diagrams for the latter. For that context, Heimdahl has developed tools for checking completeness and consistency of transition conditions described in AND/OR tables [16,17]. We can reproduce these checks in PVS if the specification method is reformulated so that the transitions are specified by means of tables, rather than graphically. An existing method that has this character is due to Lance Sherry [39]. The next section describes a PVS treatment of Sherry's decision tables.

[2]It will in a future release.

```
status: TYPE+
Lost, On_Ground, Other: status

Alt_Reporting, Other_Air_Status: VAR status

Bearing_Valid, Other_Range_Valid, Proximate_Traffic_Condition,
                                   Potential_Threat_Condition: VAR bool

Transition(Alt_Reporting,Bearing_Valid, Other_Range_Valid,
           Proximate_Traffic_Condition, Potential_Threat_Condition,
           Other_Air_Status): bool =
LET
 X = LAMBDA (x1,x2,x3,x4,x5,x6: Extended_Bool):
   (cmp(Alt_Reporting = Lost, x1)                       &
    cmp(    Bearing_Valid, x2)                          &
    cmp(        Other_Range_Valid, x3)                  &
    cmp(            Proximate_Traffic_Condition, x4)    &
    cmp(                Potential_Threat_Condition, x5)  &
    cmp(                    Other_Air_Status = On_Ground, x6))
        % |   |   |   |   |   |
        % |   |   |   |   |   |
IN      % v   v   v   v   v   v
        %---|---|---|---|---|---%
       X( T , F , ~ , ~ , ~ , ~ ) OR
        %---|---|---|---|---|---%
       X( T , ~ , F , ~ , ~ , ~ ) OR
        %---|---|---|---|---|---%
       X( T , T , T , F , F , ~ ) OR
        %---|---|---|---|---|---%
       X( ~ , ~ , ~ , ~ , ~ , T )
        %---|---|---|---|---|---%

test: LEMMA Transition(Lost, TRUE, FALSE, TRUE, TRUE, Other)
```

Figure 3.1: PVS Rendition of the AND/OR Table from Page 23

3.2 Decision Tables

Whereas AND/OR tables represent Boolean expressions, decision tables represent a collection of such expressions, together with the "decision" or output to be generated when a particular expression is true. There are many kinds of decision tables; the ones considered here are from a requirements engineering methodology developed for avionics systems by Lance Sherry of Honeywell [39], and given mechanized support in TableWise developed by Doug Hoover and others at ORA [23].

Figure 3.2 shows a simple decision table (taken from [23, Table 2]).[3] This table describes the conditions under which each of the four "operational procedures" **Takeoff**, **Climb**, **Climb_Int_Level**, and **Cruise** should be selected. The subtable beneath the name of each operational procedure can be interpreted rather like an AND/OR table, except that the input variables can have types other than **Boolean** (and * instead of - is used for "don't care"). For example, the third and fourth columns in the body of the table indicate that the operational procedure **Climb** should be used if the **Flightphase** is **climb**, **AC_Alt** is either equal or greater than **Acc_Alt**, and either **Alt_Capt_Hold** is *false*, or it is *true* and **Alt_Target** is greater than **prev_Alt_Target**.

	Operational Procedure					
Input Variables	Takeoff		Climb		Climb_Int_level	Cruise
Flightphase	climb	climb	climb	climb	climb	cruise
AC_Alt > 400	true	true	*	*	*	*
compare(AC_Alt, Acc_Alt)	LT	LT	GE	GE	*	GT
Alt_Capt_Hold	false	true	false	true	true	true
compare(Alt_Target, prev_Alt_Target)	*	GT	*	GT	*	EQ

Figure 3.2: A Simple Decision Table

We can model this decision table by combining the PVS **TABLE** construct, with a generalization of the treatment provided for AND/OR tables in the previous section. That treatment used a function **X** to give an interpretation to a column (transposed

[3]This table is a simplified version of one appearing Sherry's US patent [38, Appendix B]. Sherry's original contains several inconsistencies and incompletenesses of the kind also present in this simple example.

to a row) of an AND/OR table, such as `X(T, ~, F)`; now we need to generalize this treatment to give an interpretation to a construct like

```
X(climb, true, LT, false, *)
```

(from the first column of Figure 3.2). The previous treatment considered the arguments to `X` as extended Boolean constants to be compared with the corresponding input value using a function `cmp`. This treatment is satisfactory when all the arguments to `X` are of this same type, but it becomes rather clumsy when, as here, they can all be of different types (we would need a separate `cmp` function for each type). A better solution is to treat the arguments to `X` as *predicates* rather than constants, as shown in Figure 3.3.

```
tablewise: THEORY
BEGIN

  b:VAR bool

   true(b): bool = b
  false(b): bool = NOT b ;
      *(b): bool = TRUE

  x,y:VAR nat

  GT(x, y): bool = x > y
  LT(x, y): bool = x < y
  EQ(x, y): bool = x = y
  GE(x, y): bool = x >= y
  LE(x, y): bool = x <= y ;
   *(x, y): bool  = TRUE

  operational_procedures: TYPE = {Takeoff, Climb, Climb_Int_Level, Cruise}

  flight_phases: TYPE = {climb, cruise}

  Flightphase: VAR flight_phases
  AC_Alt, Acc_Alt, Alt_Target, prev_Alt_Target: VAR nat
  Alt_Capt_Hold: VAR bool
```

Figure 3.3: Preliminary PVS Constructions for the Decision Table in Figure 3.2

Here, `true` and `false`, for example, are not constants to be *compared* against the value of an expression such as `AC_Alt` $>$ `400`, but predicates that, when *applied* to this expression, indicate whether it is *true* or *false*, respectively. The symbol `*`, which in this example represents "don't care," is a predicate that always returns *true*. Slightly more complex are the predicates such as **GT**, which takes a *pair* of arguments and returns *true* if the first is greater than the second. Similarly, **climb?** and **cruise?** are predicates that can be applied to **Flightphase** and return *true* just in case it has the value **climb** or **cruise**, respectively.[4] The PVS specification corresponding to Figure 3.2 continues in Figure 3.4, where this generalization of

```
  decision_table(Flightphase,
                 AC_Alt,
                 Acc_Alt,
                 Alt_Target,
                 prev_Alt_Target,
                 Alt_Capt_Hold): operational_procedures =
  LET X = (LAMBDA (a: pred[flight_phases]),
                  (b: pred[bool]),
                  (c: pred[[nat,nat]]),
                  (d: pred[bool]),
                  (e: pred[[nat,nat]]):
     a(Flightphase) &
             b(AC_Alt > 400) &
                     c(AC_Alt,Acc_Alt) &
                            d(Alt_Capt_Hold) &
                                    e(Alt_Target,prev_Alt_Target)) IN TABLE
%       |        |       |       |       |
%       |        |       |       |       |
%       |        |       |       |       |
%       v        v       v       v       v   Operational Procedure
  %----------|-------|-------|-------|-------|------------- ----%
  | X(climb? , true  ,   LT  , false ,   * ) | Takeoff         ||
  %----------|-------|-------|-------|-------|------------------%
  | X(climb? , true  ,   LT  , true  ,   GT) | Takeoff         ||
  %----------|-------|-------|-------|-------|------------------%
  | X(climb? ,   *   ,   GE  , false ,   * ) | Climb           ||
  %----------|-------|-------|-------|-------|------------------%
  | X(climb? ,   *   ,   GE  , true  ,   GT) | Climb           ||
  %----------|-------|-------|-------|-------|------------------%
  | X(climb? ,   *   ,   *   , true  ,   * ) | Climb_Int_Level ||
  %----------|-------|-------|-------|-------|------------------%
  | X(cruise?,   *   ,   GT  , true  ,   EQ) | Cruise          ||
  %----------|-------|-------|-------|-------|------------------%
ENDTABLE
END tablewise
```

Figure 3.4: PVS Rendition of the Decision Table in Figure 3.2

[4]Note that **flight_phases** is specified as the enumeration type {**climb, cruise**}, which automatically creates the predicates **climb?** and **cruise?**.

the technique previously used for AND/OR tables is combined with use of the PVS `TABLE` construct.

Because the specification of Figure 3.4 uses the `TABLE` construct, PVS generates disjointness and coverage TCCs. The disjointness TCC (reformatted to fit the page) is shown in Figure 3.5.

```
% Disjointness TCC generated (line 44) for
    %  TABLE
    %   | X(climb?,  TRUE, LT, FALSE, *) | Takeoff         ||
    %   | X(climb?,  TRUE, LT, TRUE, GT) | Takeoff         ||
    %   | X(climb?,  *,    GE, FALSE, *) | Climb           ||
    %   | X(climb?,  *,    GE, TRUE, GT) | Climb           ||
    %   | X(climb?,  *,    *,  TRUE,  *) | Climb_Int_Level ||
    %   | X(cruise?, *,    GT, TRUE, EQ) | Cruise          ||
    %   ENDTABLE
  % unfinished

decision_table_TCC1: OBLIGATION
      (FORALL (X, AC_Alt, Acc_Alt, Alt_Capt_Hold,
               Alt_Target, Flightphase, prev_Alt_Target):
         X = (LAMBDA (a: pred[flight_phases]),
            (b: pred[bool]),
            (c: pred[[nat, nat]]), (d: pred[bool]), (e: pred[[nat, nat]]):
              a(Flightphase)
                  & b(AC_Alt > 400)
                    & c(AC_Alt, Acc_Alt)
                      & d(Alt_Capt_Hold) & e(Alt_Target, prev_Alt_Target))
 IMPLIES NOT (X(climb?, TRUE, LT, FALSE, *) AND X(climb?,  *, GE, FALSE, *))
 AND NOT     (X(climb?, TRUE, LT, FALSE, *) AND X(climb?,  *, GE, TRUE, GT))
 AND NOT     (X(climb?, TRUE, LT, FALSE, *) AND X(climb?,  *, *,  TRUE,  *))
 AND NOT     (X(climb?, TRUE, LT, FALSE, *) AND X(cruise?, *, GT, TRUE, EQ))
 AND NOT     (X(climb?, TRUE, LT, TRUE, GT) AND X(climb?,  *, GE, FALSE, *))
 AND NOT     (X(climb?, TRUE, LT, TRUE, GT) AND X(climb?,  *, GE, TRUE, GT))
 AND NOT     (X(climb?, TRUE, LT, TRUE, GT) AND X(climb?,  *, *,  TRUE,  *))
 AND NOT     (X(climb?, TRUE, LT, TRUE, GT) AND X(cruise?, *, GT, TRUE, EQ))
 AND NOT     (X(climb?, *,    GE, FALSE, *) AND X(climb?,  *, *,  TRUE,  *))
 AND NOT     (X(climb?, *,    GE, FALSE, *) AND X(cruise?, *, GT, TRUE, EQ))
 AND NOT     (X(climb?, *,    GE, TRUE, GT) AND X(climb?,  *, *,  TRUE,  *))
 AND NOT     (X(climb?, *,    GE, TRUE, GT) AND X(cruise?, *, GT, TRUE, EQ))
 AND NOT     (X(climb?, *,    *,  TRUE,  *) AND X(cruise?, *, GT, TRUE, EQ)));
```

Figure 3.5: Disjointness TCC for the Specification of Figure 3.4

The PVS proof command `(GRIND :EXCLUDE ("<" ">" "<=" ">="))` discharges 11 of the 13 cases in the TCC, but fails on two of them. After eliminating irrelevant definitions with the command `(HIDE -1 -2 -3 -4 -5)`, these reduce to the following subgoals.

```
decision_table_TCC1.1 :

[-1]    climb?(Flightphase!1)
[-2]    AC_Alt!1 > 400
[-3]    AC_Alt!1 < Acc_Alt!1
[-4]    Alt_Capt_Hold!1
[-5]    Alt_Target!1 > prev_Alt_Target!1
  |-------

Rule? (POSTPONE)

decision_table_TCC1.2 :

[-1]    climb?(Flightphase!1)
[-2]    Alt_Capt_Hold!1
[-3]    AC_Alt!1 >= Acc_Alt!1
[-4]    Alt_Target!1 > prev_Alt_Target!1
  |-------

Rule?
```

Since these sequents have nothing below the turnstile line, the only way they could be true is if the formulas above the line are mutually contradictory. PVS is unable to establish such contradictions, and thereby identifies flaws in the original table corresponding to the cases where all the formulas above the line in each sequent are true. The first sequent identifies a circumstance that satisfies both columns 2 and 5 of the original table in Figure 3.2 (corresponding to rows 2 and 5 of the PVS table in Figure 3.4), thereby leading to the conflicting selection of two different operational procedures (`Takeoff` and `Climb_Int_Level`). The second sequent identifies a similar conflict between columns 4 and 5. These flaws are identical to those identified by the special-purpose tool TableWise [23, Table 3].

The coverage TCC generated from the specification of Figure 3.4 is shown in Figure 3.6. The same proof commands as those used for the disjointness TCC produce the four unprovable subgoals shown in Figure 3.7. As before, PVS's inability to discharge these proof obligations identifies flaws in the specification. These sequents have nothing above the turnstile line, so for them to be true it is enough that just one of the formulas below the line should be true in each case. Since PVS cannot establish this, we must consider the case when all the formulas below the line in each sequent are false. The first sequent, for example, identifies the failure to select an operational procedure when `AC_Alt` is not greater than 400, `Alt_Capt_Hold` is

```
% Coverage TCC generated (line 44) for
    %  TABLE
    %   | X(climb?,  TRUE, LT, FALSE, *) | Takeoff         ||
    %   | X(climb?,  TRUE, LT, TRUE, GT) | Takeoff         ||
    %   | X(climb?,  *,    GE, FALSE, *) | Climb           ||
    %   | X(climb?,  *,    GE, TRUE, GT) | Climb           ||
    %   | X(climb?,  *,    *,  TRUE,  *) | Climb_Int_Level ||
    %   | X(cruise?, *,    GT, TRUE, EQ) | Cruise          ||
    %   ENDTABLE
  % unfinished

decision_table_TCC2: OBLIGATION
      (FORALL (X, AC_Alt, Acc_Alt, Alt_Capt_Hold,
               Alt_Target, Flightphase, prev_Alt_Target):
         X = (LAMBDA (a: pred[flight_phases]),
            (b: pred[bool]),
            (c: pred[[nat, nat]]), (d: pred[bool]), (e: pred[[nat, nat]]):
              a(Flightphase)
                  & b(AC_Alt > 400)
                    & c(AC_Alt, Acc_Alt)
                      & d(Alt_Capt_Hold) & e(Alt_Target, prev_Alt_Target))
             IMPLIES X(climb?,  TRUE, LT, FALSE, *)
                  OR X(climb?,  TRUE, LT, TRUE, GT)
                  OR X(climb?,  *,    GE, FALSE, *)
                  OR X(climb?,  *,    GE, TRUE, GT)
                  OR X(climb?,  *,    *,  TRUE,  *)
                  OR X(cruise?, *,    GT, TRUE, EQ));
```

Figure 3.6: Coverage TCC for the Specification of Figure 3.4

```
decision_table_TCC2.1 :

  |-------
[1]    AC_Alt!1 > 400
[2]    Alt_Capt_Hold!1
[3]    AC_Alt!1 >= Acc_Alt!1

Rule? (POSTPONE)
Postponing decision_table_TCC2.1.

decision_table_TCC2.2 :

  |-------
[1]    climb?(Flightphase!1)
[2]    Alt_Target!1 = prev_Alt_Target!1

Rule? (POSTPONE)
Postponing decision_table_TCC2.2.

decision_table_TCC2.3 :

  |-------
[1]    climb?(Flightphase!1)
[2]    AC_Alt!1 > Acc_Alt!1

Rule? (POSTPONE)
Postponing decision_table_TCC2.3.

decision_table_TCC2.4 :

  |-------
[1]    climb?(Flightphase!1)
[2]    Alt_Capt_Hold!1

Rule?
```

Figure 3.7: False Subgoals from the Coverage TCC of Figure 3.6

false, and `AC_Alt` is less than `Acc_Alt`. As before, the four flaws identified by these false subgoals are identical to those identified by the special-purpose tool TableWise [23, Table 4].

Unlike PVS, TableWise presents the anomalies that it discovers in a tabular form similar to that of the original decision table; TableWise can also generate executable Ada code and English language documentation from decision tables. These benefits are representative of those that can be achieved with a special-purpose tool. On the other hand, PVS's more powerful deductive capabilities also provide benefits. For example, PVS can settle disjointness and coverage TCCs that depend on properties more general than the simple Boolean and arithmetic relations built in to TableWise and similar tools. Heimdahl, who with Leveson developed a completeness and consistency checking tool for the AND/OR tables of RSML [17], describes spurious error reports when that tool was applied to TCAS II [16]. These were due to the presence of arithmetic and defined functions whose properties are beyond the reach of the BDD-based[5] tautology checker incorporated in the tool. As Heimdahl notes [16, page 81], a theorem prover is needed to settle such properties.

A theorem prover such as PVS can also examine questions beyond simple completeness and consistency. For example, Figure 3.8 presents a specification that corrects the incompleteness and inconsistencies detected in the specification of Figure 3.4. (The incompleteness is remedied by adding an `ELSE` clause, and the inconsistencies by replacing the "don't care" entries in the second and third columns of row 5 by `false` and `LT`, respectively.) Since the single TCC generated by this specification is provable (using `(grind)`), we may examine additional properties of the function `decision_table2`. To check that the specification matches our intent, we can use conjectures that we believe to be true as "challenges." For example, we may believe that when `AC_Alt = Acc_Alt`, the operational procedure selected should match the `Flightphase`. We can check this in the case that the `Flightphase` is `cruise` using the following challenge.

```
  test: THEOREM AC_Alt = Acc_Alt =>
    decision_table2(cruise, AC_Alt, Acc_Alt,
                    Alt_Target, prev_Alt_Target, Alt_Capt_Hold) = Cruise
```

This is easily proved using `(grind)`.

However, when we try the corresponding challenge for the case where `Flightphase` is `climb`,

```
  test2: THEOREM AC_Alt = Acc_Alt =>
    decision_table2(climb, AC_Alt, Acc_Alt,
                    Alt_Target, prev_Alt_Target, Alt_Capt_Hold) = Climb
```

[5]Ordered Binary Decision Diagrams (BDDs) are a very efficient representation for reasoning about Boolean functions and propositional calculus [5].

```
  decision_table2(Flightphase,
                 AC_Alt,
                 Acc_Alt,
                 Alt_Target,
                 prev_Alt_Target,
                 Alt_Capt_Hold): operational_procedures =
  LET X = (LAMBDA (a: pred[flight_phases]),
                  (b: pred[bool]),
                  (c: pred[[nat,nat]]),
                  (d: pred[bool]),
                  (e: pred[[nat,nat]]):
     a(Flightphase) &
             b(AC_Alt > 400) &
                     c(AC_Alt,Acc_Alt) &
                            d(Alt_Capt_Hold) &
                                    e(Alt_Target,prev_Alt_Target)) IN TABLE
%       |        |       |       |       |
%       |        |       |       |       |
%       |        |       |       |       |
%       v        v       v       v       v   Operational Procedure
  %----------|-------|-------|-------|-------|------------- ----%
  | X(climb? , true  ,   LT  , false ,   * ) | Takeoff         ||
  %----------|-------|-------|-------|-------|------------------%
  | X(climb? , true  ,   LT  , true  ,   GT) | Takeoff         ||
  %----------|-------|-------|-------|-------|------------------%
  | X(climb? ,   *   ,   GE  , false ,   * ) | Climb           ||
  %----------|-------|-------|-------|-------|------------------%
  | X(climb? ,   *   ,   GE  , true  ,   GT) | Climb           ||
  %----------|-------|-------|-------|-------|------------------%
  | X(climb? , false ,   LT  , true  ,   * ) | Climb_Int_Level ||
  %----------|-------|-------|-------|-------|------------------%
  | X(cruise?,   *   ,   GT  , true  ,   EQ) | Cruise          ||
  %----------|-------|-------|-------|-------|------------------%
  | ELSE                                     | Cruise          ||
  %------------------------------------------|------------------%
ENDTABLE
END tablewise
```

Figure 3.8: Corrected Version of the Decision Table in Figure 3.4

we discover that `(grind)` produces the following unproven goal.

```
test2 :

{-1}    Acc_Alt!1 >= 0
{-2}    Alt_Target!1 >= 0
{-3}    prev_Alt_Target!1 >= 0
{-4}    AC_Alt!1 = Acc_Alt!1
{-5}    Alt_Capt_Hold!1
  |-------
{1}    Alt_Target!1 > prev_Alt_Target!1

Rule?
```

The first three formulas are simply type predicates for the natural numbers concerned, and the next is the hypothesis to this challenge, but formulas `-5` and `1` reveal that we have overlooked the case where `Alt_Capt_Hold` is *true* and `Alt_Target <= prev_Alt_Target` (the latter condition is negated because it appears below the turnstile line). Further examination of the table (or another mechanically checked challenge) will disclose that the value of the function is not `Climb` but `Cruise` in this case, thereby exposing a flaw in either our expectations or our formalization of this function. Mechanically supported challenges of this kind illustrate the utility of undertaking the analysis of tabular specifications in a context that provides theorem proving. Special-purpose tools for tabular specifications generally provide only completeness and consistency checking, and perhaps some form of simulation. Such tools would help identify the flaw described only if we happened to choose to simulate a case where `Alt_Capt_Hold` is *true* and `Alt_Target <= prev_Alt_Target`.

Decision tables provide a way to specify the selection of operational procedures to be executed at each step. However, the model of computation that repeatedly performs these selection and execution steps is understood informally and is not explicit in the PVS specifications. Consequently, it is not possible to pose and examine overall system properties—such as whether a certain property is invariant, or another is reachable—without formalizing more of the underlying model of computation. In the following chapter, we will do this for the requirements specification methodology known as SCR.

Chapter 4

State Transition Systems and SCR Requirements Specifications

A common way to model distributed, concurrent, or reactive systems is by means of ***transition relations***. The instantaneous state of the system is represented by an assignment of values to its variables. As it executes, the system progresses from one state to another, and the transition relation specifies the possible successors to each state. The sequence of states visited in one run of the system is called a ***trace***; the set of all traces is called the ***behavior*** of the system.

Usually, the transition relation for a system is not specified monolithically, but as the interaction of several subsystems operating in parallel. Each subsystem will be characterized by its own transition relation and the composite, overall transition relation can then be defined as either the disjunction of the individual relations ("interleaving" concurrency) or their conjunction[1] ("true" concurrency).

Verification questions one might ask of transition relations include whether the behavior induced by one (regarded as an implementation) implies that of another (regarded as a specification), whether a certain property is true of all reachable states (i.e., an invariant), and whether a state having a certain property is reachable on some or all traces starting from some given state (these are examples of "liveness" properties). Many such properties of sets of traces can be specified compactly by means of ***temporal logic***. To ask whether the behavior specified by a certain transition relation satisfies a property specified by a certain formula of temporal logic can be viewed as asking whether the relation is a Kripke model of the formula. For certain temporal logics and for transition relations that induce a finite state space,

[1]The individual relations must usually allow "stuttering" (i.e., no change) steps in this case.

this *model checking* question can be decided very efficiently (i.e., in time linear in the length of the temporal logic formula and the number of states in the transition system) by a rather sophisticated form of brute force search. The invention and popularization of this approach is due to Edmund Clarke and his students [6,9,10]. For certain classes of systems and properties, model checking is an attractive alternative, or adjunct, to verification by theorem proving, because of its largely automatic character.

Using an efficient decision procedure[2] based on BDDs for a logic known as the Park's μ-calculus (this is basically quantified Boolean logic with least and greatest fixpoint operators [31]), PVS provides model checking for a temporal logic known as Computation Tree Logic (CTL) and transition relations defined on heriditarily finite types [35].[3]

Here, we consider the use of PVS to examine transition relations derived from the Naval Research Laboratory's SCR method for requirements specification [11]. We begin by considering how PVS can represent certain aspects of SCR specifications in a natural manner, and how it can check those specifications for well-formedness. This treatment builds directly on that developed in the previous chapter. We then consider use of PVS's model checker to examine application properties of SCR specifications. Finally, we consider specifications composed of more than one transition relation and use PVS's model checker to decide equivalence of the behaviors induced by different transition relations.

4.1 Representing SCR Specifications in PVS

In the SCR method [19], a system is described in terms of state machines that interact with their environment by periodically sampling the values of *monitored* (i.e., input) *variables* and calculating values to be assigned to *controlled* (i.e., output) *variables*. The states of an individual state machine are called *modes*. A *condition* is a predicate on the monitored variables; an *event* occurs when a monitored variable changes value. The *mode transitions* of an individual state machine are triggered by events, or by *conditioned events*—these are events that occur while certain conditions hold constant. Mode transitions are generally specified by a table such as the one shown in Figure 4.1. Complex systems are defined by several state

[2]This procedure, and also the BDD-based propositional simplifier invoked by PVS's `(bddsimp)` command, were provided by Geert Janssen of the Electrical Engineering Department of Eindhoven University of Technology in the Netherlands [27].

[3]This capability is similar to that of the SMV model checker [30]. Note that the μ-calculus is strictly more expressive than CTL, and is also used to define "fair" versions of the CTL operators within PVS. We are currently investigating the extension of PVS's μ-calulus-based model checking to linear-time temporal logic (CTL is a "branching-time" logic [28]), and to language containment [15].

Current Mode	Conditions							Next Mode
	Ignited	Running	Toofast	Brake	Activate	Deactivate	Resume	
Off	@T	-	-	-	-	-	-	Inactive
Inactive	@F	-	-	-	-	-	-	Off
	T	T	-	F	@T	-	-	Cruise
Cruise	@F	-	-	-	-	-	-	Off
	-	@F	-	-	-	-	-	Inactive
	-	-	@T	-	-	-	-	Inactive
	-	-	-	@T	-	-	-	Override
	-	-	-	-	-	@T	-	Override
Override	@F	-	-	-	-	-	-	Off
	-	@F	-	-	-	-	-	Inactive
	T	T	-	F	@T	-	-	Cruise
	T	T	-	F	-	-	@T	Cruise

Figure 4.1: Original Mode Transition Table for Cruise Control

machines operating in parallel and will have several such mode transition tables. Also, in such cases, the conditions in one table may refer to the modes of another, and events may include mode transitions of other state machines. These extensions require elaborations of the treatment given here, and we ignore them for brevity. However, we do consider interacting state machines in Section 4.3.

The mode transition table of Figure 4.1, taken from Atlee and Gannon [3, Table 2],[4] describes an automobile cruise control system.[5] This system has four modes: `off`, `inactive`, `cruise`, and `override`. The system is in exactly one of these four modes at all times. The system starts in the `off` mode, which represents the case where the car's ignition is off. The `inactive` mode stands for the case where the car's ignition is on, but the cruise control is off. The `cruise` mode is the case where both ignition and cruise control are on, and the cruise control is actually controlling the vehicle's speed. Finally, the `override` mode applies when both the ignition and cruise control are on, but the cruise control is not controlling the vehicle's speed.

The table of Figure 4.1 uses the following conditions on the system's monitored variables.

`Ignited:` The ignition is on.

`Running:` The engine is running.

`Toofast:` The vehicle speed is above that which the system can control.

[4]The same example is used in two papers by Atlee and Gannon [3, Tables 2 and 3], [4, Tables IV and V], and one by Atlee and Buckley [2, Figure 4]; however, the SCR tables are slightly different in each paper.

[5]This description does not resemble any real cruise control; we use it because it has been studied by others and thereby facilitates comparison between our methods and theirs.

`Brake`: The brake is being applied.

`Activate`: The cruise control lever is set to the “activate” position.

`Deactivate`: The cruise control level is set to the “deactivate” position.

`Resume`: The cruise control level is set to the “resume” position.

An `@T` entry in the mode transition table indicates an event where the condition in that column goes from *false* to *true*. For example, the `@T` in the first column of the first row of the table signifies an event that `ignited` goes from *false* to *true*. An `@F` entry similarly indicates an event that the condition corresponding to that column goes from *true* to *false*. The simple entries `T` and `F` indicate that the condition in their column remains *true* or *false*, respectively. The rows of the mode transition table specify conditioned events that trigger their associated mode transitions. For example, the third row specifies that a transition from the `inactive` to the `cruise` mode takes place when an `activate` event occurs while `ignited` and `running` remain *true* and `brake` remains *false* (the dashes indicate “don’t care” conditions). The system remains in its current mode until an event causes it to transition to another mode.

To specify this system in PVS, we first need to model the basic constructs of the SCR method, such as the notions of events and conditions, and the meaning of notations such as `@T`. The notions of the SCR method are defined relative to the input (monitored) and output (controlled) variables, and the system modes: a *condition*, for example, is formally a predicate on the inputs. We therefore specify the SCR constructs in a theory that is parameterized by the `input`, `mode`, and `output` types. This theory begins as follows.

```
scr[ input, mode, output: TYPE ]: THEORY
BEGIN
  condition: TYPE = pred[input]
  event: TYPE = pred[[input, input]]
  state: TYPE = [# mode: mode, vars: input #]
  ...
```

It specifies that a `condition` is a predicate on `input`s, while an `event` is a predicate on pairs of `input`s (or, equivalently, a relation on `input`s).[6] The instantaneous system `state` is a record composed of the current `mode` and current values of the `input`s.

It turns out to be very convenient to be able to apply a `condition` to a `state`, with the interpretation that the condition is actually to be applied to the inputs of

[6]In the declaration for the `event` type, the outer pair of brackets encloses the parameter to the `pred` type-constructor; the inner pair is the tuple-type constructor.

that state. The higher-order function `liftc` makes it possible to do this in a uniform fashion: if `cnd` is a `condition`, then `liftc(cnd)` is a predicate on `state`s that has the desired behavior (i.e., it applies `cnd` to the `vars` component of the state). By further declaring `liftc` to be a `CONVERSION`, we tell the PVS typechecker that it may insert an application of `liftc` wherever it will turn a type-incorrect application into a type-correct one. Thus, we can write `cnd(s)`, where `cnd` is a `condition` and `s` is a `state`, and PVS will automatically convert this to `liftc(cnd)(s)`.

```
liftc(cnd:condition): pred[state] = LAMBDA (s:state): cnd(vars(s))
CONVERSION liftc

liftm(mde: pred[mode]): pred[state] = LAMBDA (s:state): mde(mode(s))
CONVERSION liftm
```

The conversion `liftm` is defined similarly for predicates on `mode`s.

A trace is a sequence of states whose adjacent members are related by some transition relation. The important item to capture here is the notion of *transition relation*.

```
transition_relation: TYPE = pred[[state, state]]
```

This says that a `transition_relation` is a predicate on pairs of `state`s (i.e., a relation on `state`s).

The instantaneous values of the input variables are determined by the environment and are not under our control. When we specify the behavior of a particular state machine, we must be careful, therefore, to do so in a way that does not constrain how input variables may change from one state to another.[7] Hence, we do not specify the transition relation directly (since it would then be hard to check that we were not constraining the way in which inputs could change from one state to another), but do so implicitly by means of a *mode table* that allows us to specify only the part of the system that is under our control (i.e., in this case, just the modes).

```
mode_table: TYPE = [mode, input, input -> mode]
```

A `mode_table` specifies a new `mode` for the system as a function of its previous `mode`, and previous and current `input`s. We can then specify a function `trans` that constructs a `transition_relation` from a `mode_table` by specifying that two states `s` and `t` are in the relation whenever the mode of `t` equals that required by the mode table when given the mode of `s` and the inputs of `s` and `t`.

[7]It is acceptable to constrain the way input variables change if this is an explicit property or assumption about the environment (we will see an example of this shortly in the axiom `engine_prop`); it is not acceptable to do it accidentally while specifying the part of the system that we intend to implement.

```
trans(mt: mode_table): transition_relation =
  (LAMBDA (s,t: state): mode(t) = mt(mode(s), vars(s), vars(t)))
```

An `output_table` specifies the current `output` in a manner similar to that of a `mode_table`. For brevity, we will not specify particular `output_table`s for our examples.

```
output_table: TYPE = [mode, input, input -> output]
```

To specify a particular mode table such as Figure 4.1, we need to define the operators such as `@T` that appear within it. The operator `@T` in the column for `ignited` really stands for `@T(ignited)`, and represents the event where the condition `ignited` goes from *false* to *true*. Thus, `@T` is a function from `condition`s to `event`s. We say that such functions have type `event_constructor`, or `EC` for short.

```
event_constructor: TYPE = [condition -> event]
EC: TYPE = event_constructor
```

Now if `P` is a `condition`, `@T(P)` is the `event` that is *true* of two sets of input values `p` and `q` if `P` goes from *false* to *true* between them: that is, if `P(p)` is *false* and `P(q)` is *true*. We can define this in PVS as follows (because `@` cannot be used in a PVS identifier we use `atT` instead of `@T`). Observe the explicitly higher-order character of this definition.

```
p,q: VAR input
P: VAR condition

atT(P)(p,q): bool = NOT P(p) & P(q)      % @T(P)
```

We can define `atF` (i.e., `@F`) dually, and similarly the transition constructors `T` and `F`, which are *true* if their argument condition `P` remains *true* (resp. *false*) in the transition from `p` to `q`. We will also need the "don't care" transition constructor `dc`, which is always *true*.

```
atF(P)(p,q): bool = P(p) & NOT P(q)      % @F(P)
T(P)(p,q):   bool = P(p) & P(q)
F(P)(p,q):   bool = NOT P(p) & NOT P(q)
dc(P)(p,q):  bool = true                 % don´t care
```

This gives us all the generic constructions we need for the time being, and we can proceed to specify the particular mode transition table given in Figure 4.1. To begin, we need to specify the inputs and the system modes for this example. Among the inputs, we know that there is a cruise control lever that can take on three positions: `activate`, `deactivate`, and `resume`. We can specify these values as the components of a PVS enumerated type, as follows.

```
cruise: THEORY
BEGIN
  lever_pos: TYPE = {activate, deactivate, resume}
```

Similarly, the engine can be in one of three states; `off`, `running`, and an intermediate state where the `ignition` is on but the engine is not running.

```
  engine_state: TYPE = { off, ignition, running }
```

The input to the system will be a record of several fields: as well as the cruise control lever position and engine state, we need Boolean-valued fields that record whether the vehicle is going `toofast`, and whether the `brake` is on.

```
  monitored_vars: TYPE = [#
                  engine:   engine_state
                  toofast:  bool,
                  brake:    bool,
                  lever:    lever_pos
                 #]
```

We also define the modes of this system as an enumerated type.

```
  modes: TYPE = { off, inactive, cruise, override }
```

Since we will not specify the output behavior of the system, we will use an uninterpreted type `null` for this purpose.

```
  null: TYPE
```

Now that we have defined all the components of the system state, we can import the appropriate instance of the `scr` theory.

```
  IMPORTING scr[ monitored_vars, modes, null ]
```

To formally specify the mode transition table of Figure 4.1 in PVS, we next need to specify the conditions that label its columns. The condition `activate`, for example, is *true* when the `lever` field of the `monitored_vars` has the value `activate` (note, `activate` will be overloaded here as both a condition and the value of an enumerated type). Because `lever_pos` is an enumerated type, `activate?` is the predicate that recognizes this value, and so the specification is written as follows.

```
activate:   condition = LAMBDA (m:monitored_vars): activate?(lever(m))
```

The conditions `deactivate`, `resume`, and `running` are defined similarly.

```
deactivate: condition = LAMBDA (m:monitored_vars): deactivate?(lever(m))
resume:     condition = LAMBDA (m:monitored_vars): resume?(lever(m))
running:    condition = LAMBDA (m:monitored_vars): running?(engine(m))
```

The condition `ignited` is a little more complicated. It is to be *true* whenever the ignition is on, and this is obviously so when the `engine_state` is `ignition`; however, it is also so when `engine_state` is `running`, because the ignition must surely be on for the engine to be running. Hence, we have the following specification.

```
ignited:    condition = LAMBDA (m:monitored_vars):
                           ignition?(engine(m)) OR running?(engine(m))
```

The condition `brake` is to be true whenever the `brake` field in the `monitored_vars` is *true*. The condition `toofast` is defined similarly.

```
brake  :    condition = LAMBDA (m:monitored_vars): brake(m)
toofast:    condition = LAMBDA (m:monitored_vars): toofast(m)
```

These two definitions may seem redundant, but they are not. In their absence, the term `brake(m)` may look like a condition (i.e., predicate) applied to a variable `m` of type `monitored_vars`, but it is not: this `brake` is a record field selector and cannot appear on its own (i.e., not applied to a record `m`). It is necessary to explicitly overload `brake` by the definition above to be able to use it as a condition.

We now need to define the mode table of Figure 4.1 in PVS. This kind of table is rather different than any we have seen before, but it can be recast in the following generic form.

Current Mode	Conditioned Event	New Mode
m_1	$e_{1,1}$	$m_{1,1}$
	$e_{1,2}$	$m_{1,2}$
	$\cdots$	$\cdots$
	e_{1,k_1}	m_{1,k_1}
m_2	$e_{2,1}$	$m_{2,1}$
	$e_{2,2}$	$m_{2,2}$
	$\cdots$	$\cdots$
	e_{2,k_2}	m_{2,k_2}
$\cdots$	$\cdots$	$\cdots$
m_p	$e_{p,1}$	$m_{p,1}$
	$e_{p,2}$	$m_{p,2}$
	$\cdots$	$\cdots$
	e_{p,k_p}	m_{p,k_p}

This is actually the way mode transition tables are presented in formal treatments of the SCR method [19] and is similar to that used in the SCR* toolset [20]. Tables of this form can be specified in PVS using a one-dimensional vertical table to enumerate the Current Mode, with the Conditioned Event/New Mode subtables (inside the doubled lines) specified in the manner used for decision tables in the previous chapter. Using this approach, we can represent the mode table of Figure 4.1 by the PVS specification shown in Figure 4.2. In this specification, the function `PC` is imported from the generic `scr` theory and is defined as follows.

```
A,B,C,D,E,FF,G,H,I,J: VAR EC
a,b,c,d,e,f,g,h,i,j: VAR condition

PC(A)(a)(p,q):bool = A(a)(p,q)
PC(A,B)(a,b)(p,q):bool = A(a)(p,q) & B(b)(p,q)
...
PC(A,B,C,D,E,FF,G)(a,b,c,d,e,f,g)(p,q):bool = A(a)(p,q) & B(b)(p,q) &
    C(c)(p,q) & D(d)(p,q) & E(e)(p,q) & FF(f)(p,q) & G(g)(p,q)[8]
...
```

That is, `PC` (the name is short for "pairwise conjunction") is defined as a collection of functions that each take a list of event constructors and a list of conditions and conjoins their pairwise applications. The collection contains versions of `PC` for

[8]The variable `FF` is used rather than `F` because the latter is already defined in this context as the event constructor that is *true* when both its arguments are *false*.

```
  original(s: modes, (p, q: monitored_vars)): modes =
  LET
    x: conds7 = (ignited, running, toofast, brake, activate, deactivate, resume),
    X = (LAMBDA (a,b,c,d,e,f,g:EC): PC(a,b,c,d,e,f,g)(x)(p,q))
  IN TABLE s
   |off| TABLE
      %----|----|----|----|----|----|----|----|-----------||
      |X(   atT , dc , dc , dc , dc , dc , dc)| inactive  ||
      %----|----|----|----|----|----|----|----|-----------||
      |    ELSE                               | off       ||
      %----|----------------------------------|-----------||
    ENDTABLE ||

   |inactive| TABLE
      %----|----|----|----|----|----|----|-----|----------||
      |X(   atF , dc , dc , dc , dc , dc , dc )| off      ||
      %----|----|----|----|----|----|----|-----|----------||
      |X(    T  , T  , dc , F  ,atT , dc , dc )| cruise   ||
      %----|----|----|----|----|----|----|-----|----------||
      |    ELSE                                | inactive ||
      %----|-----------------------------------|----------||
     ENDTABLE ||

   |cruise| TABLE
      %----|----|----|----|----|----|----|-----|----------||
      |X(   atF,  dc,  dc,  dc,  dc,  dc,  dc )| off      ||
      %----|----|----|----|----|----|----|-----|----------||
      |X(    dc ,atF , dc , dc , dc , dc , dc )| inactive ||
      %----|----|----|----|----|----|----|-----|----------||
      |X(    dc , dc ,atT , dc , dc , dc , dc )| inactive ||
      %----|----|----|----|----|----|----|-----|----------||
      |X(    dc , dc , dc ,atT , dc , dc , dc )| override ||
      %----|----|----|----|----|----|----|-----|----------||
      |X(    dc , dc , dc , dc , dc ,atT , dc )| override ||
      %----|----|----|----|----|----|----|-----|----------||
      |    ELSE                                | cruise   ||
      %----|-----------------------------------|----------||
     ENDTABLE ||

    |override| TABLE
      %----|----|----|----|----|----|----|-----|----------||
      |X(   atF , dc   dc , dc , dc , dc , dc )| off      ||
      %----|----|----|----|----|----|----|-----|----------||
      |X(    dc ,atF , dc , dc , dc , dc , dc )| inactive ||
      %----|----|----|----|----|----|----|-----|----------||
      |X(    T  , T  , dc , F  ,atT , dc , dc )| cruise   ||
      %----|----|----|----|----|----|----|-----|----------||
      |X(    T  , T  , dc , F  , dc , dc ,atT )| cruise   ||
      %----|----|----|----|----|----|----|-----|----------||
      |    ELSE                                | override ||
      % ---|-----------------------------------|----------||
     ENDTABLE ||
ENDTABLE
```

Figure 4.2: PVS Version of the Original Specification of Figure 4.1

different numbers of arguments; PVS resolves the overloading by the number of arguments provided in any particular application. The type `conds7` appearing in the `LET` clause of Figure 4.2 is also defined in the generic `scr` theory.

```
conds1:type = [condition]
conds2:type = [condition, condition]
...
conds7:type = [condition, condition, condition, condition,
                  condition, condition, condition]
...
```

4.1.1 Well-Formedness Checking for SCR Specifications in PVS

Typechecking the definition `original` of Figure 4.2 generates three TCCs; these are disjointness TCCs from the nested tables that specify the transitions from the modes `inactive`, `cruise`, and `override`. No disjointness TCC is generated for the transitions from mode `off`, since there is only a single non-`ELSE` case in its table. And no coverage TCCs are generated from any of these tables because they all have `ELSE` cases. No TCCs are generated from the outermost table, since PVS recognizes that it is simply enumerating the values of an enumerated type. The disjointness TCC from the table giving the transitions from mode `inactive` is proved automatically by PVS's default strategy for TCCs, but the other two are not. After applying the default `(cond-disjoint-tcc)` proof strategy to the disjointness TCC from the table giving the transitions from mode `cruise`, and eliminating some irrelevant formulas with `(hide -1 -2)`, we are presented with the following sequent.

```
Trying repeated skolemization, instantiation, and if-lifting,
this yields  8 subgoals:
original_TCC2.1 :

{-1}    cruise?(s!1)
{-2}    toofast(q!1)
{-3}    deactivate?(lever(q!1))
  |-------
{1}    toofast(p!1)
{2}    deactivate?(lever(p!1))

Rule?
```

This sequent is inviting us to contemplate the case where `toofast` and `deactivate` both go from *false* to *true* when in `cruise` mode. Referring back to the specification, we see that the first of these causes a transition to `inactive` mode, while the second causes a transition to `override` mode. The other subgoals of this failed proof reveal further similar ambiguities in the mode transitions from `cruise` mode; similar analysis of the third TCC reveals comparable problems in the mode transitions from `override` mode.

It seems clear that the specification should be modified so that the transitions from `cruise` mode to `override` mode are conditioned on `toofast` remaining *false*, and `running` remaining *true*. There are similar problems in the transitions from the `cruise` and `override` modes to the `off` and `inactive` modes: the transitions to `off` occur when `ignited` goes *false*, while those to `inactive` can occur when `running` goes *false*, and both of these events can occur at once. It seems that the transitions to `inactive` need to be conditioned on `ignited` staying *true*.

But what about the apparent ambiguity in the transitions from `cruise` to `off` when `ignited` goes `false`, and to `inactive` when `toofast` goes *true*? Atlee and Gannon [3,4] argue that there is no real ambiguity here, because these events cannot occur together—the engine surely has to be `running` (and therefore `ignited`) for the vehicle to go `toofast`. Atlee and Gannon add an assumption to this effect to their specification; we also add it to our specification as the axiom `engine_prop`.[9]

```
   engine_prop: AXIOM toofast(p) => running(p)
```

With this assumption, we can simplify the table by removing the condition that `ignited` stays *true* from any transitions where `running` stays or goes *true*. Notice that unlike Atlee and Gannon [3, 4], we do not need to add axioms to ensure disjointness of the conditions `activate`, `deactivate`, and `resume`, since these follow automatically by their derivation from an enumerated type. Also, we do not need to be concerned that (for example) the last two transitions from `cruise` mode have overlapping conditions—because the destination mode is `override` in both cases. PVS suppresses the disjointness TCC on `COND` (and hence `TABLE`) entries that have syntactically identical actions. The revised mode transition table incorporating these corrections and simplifications is shown in Figure 4.3, and the corresponding PVS specification is shown in Figure 4.4. The three disjointness TCCs generated by the revised specification are all proved by the following command.

```
   (then (grind)(lemma "engine_prop")(grind :if-match all))
```

In the next section we show how the model checking capabilities of PVS can be used to examine application-specific properties of this specification.

[9]Atlee and Gannon conjoin `running(p) => ignited(p)` to this axiom; we do not need to do so because this property is built in to the way we defined the condition `ignited`. We could have avoided the need for the axiom altogether by suitably modifying the definitions of `ignited` and `running`, but chose not to do so for variety.

Current Mode	Conditions							Next Mode
	Ignited	Running	Toofast	Brake	Activate	Deactivate	Resume	
Off	@T	-	-	-	-	-	-	Inactive
Inactive	@F	-	-	-	-	-	-	Off
	-	T	-	F	@T	-	-	Cruise
Cruise	@F	-	-	-	-	-	-	Off
	T	@F	-	-	-	-	-	Inactive
	-	-	@T	-	-	-	-	Inactive
	-	T	F	@T	-	-	-	Override
	-	T	F	-	-	@T	-	Override
Override	@F	-	-	-	-	-	-	Off
	T	@F	-	-	-	-	-	Inactive
	-	T	-	F	@T	-	-	Cruise
	-	T	-	F	-	-	@T	Cruise

Figure 4.3: Deterministic Mode Transition Table for Cruise Control

4.2 Model Checking SCR Specifications in PVS

TCCs generated by PVS's `COND` (and hence `TABLE`) construct provided useful well-formedness checks on our SCR requirements specification for the automobile cruise control example. The TCCs led us to discover flaws in the specification, and enabled us to demonstrate the absence of these flaws in the corrected specification. Deeper assurance that the specification captures our intent and intuitive understanding requires that we go beyond static attributes of the transition relation and examine properties of the behavior that it induces. A useful class of properties can be expressed in the branching time temporal logic called CTL, and their satisfaction by the behavior induced by a given transition relation can be determined very efficiently by model checking. Atlee and Gannon [3, 4] were the first to apply this idea to SCR specifications. Their approach used a rather indirect encoding of SCR specifications and the MCB model checker. Later, Atlee [1] developed a more direct encoding suitable for the SMV symbolic model checker [30] that has subsequently been applied to large examples [40]. Here, we apply PVS's model checker directly to the PVS specifications already developed.

CTL is a branching time temporal logic that extends propositional logic with modal operators: $AX(P)$ is true when the state predicate (i.e., SCR condition) P holds in every immediate successor to the current state; $EX(P)$ is true when P holds in some immediate successor to the current state; $AF(P)$ (resp. $EF(P)$) means that along every (resp. some) path (i.e., trace, or succession of states) from the current state there exists some future state in which P holds; finally, $AG(P)$ (resp. $EG(P)$) means that P holds in every state along every (resp. some) path from the current state.

```
  deterministic(s: modes, (p, q: monitored_vars)): modes =
  LET
    x: conds7 = (ignited, running, toofast, brake, activate, deactivate, resume),
    X = (LAMBDA (a,b,c,d,e,f,g:EC): PC(a,b,c,d,e,f,g)(x)(p,q))
  IN TABLE s
   |off| TABLE
      %----|----|----|----|----|----|----|----|-----------||
      |X(   atT , dc , dc , dc , dc , dc , dc)| inactive  ||
      %----|----|----|----|----|----|----|----|-----------||
      |    ELSE                              | off       ||
      %----|---------------------------------|-----------||
    ENDTABLE ||

   |inactive| TABLE
      %----|----|----|----|----|----|----|-----|----------||
      |X(   atF , dc , dc , dc , dc , dc , dc )| off      ||
      %----|----|----|----|----|----|----|-----|----------||
      |X(    dc , T  , dc , F  ,atT , dc , dc )| cruise   ||
      %----|----|----|----|----|----|----|-----|----------||
      |    ELSE                               | inactive ||
      %----|----------------------------------|----------||
     ENDTABLE ||

   |cruise| TABLE
      %----|----|----|----|----|----|----|-----|----------||
      |X(   atF,  dc,  dc,  dc,  dc,  dc,  dc )| off      ||
      %----|----|----|----|----|----|----|-----|----------||
      |X(    T  ,atF , dc , dc , dc , dc , dc )| inactive ||
      %----|----|----|----|----|----|----|-----|----------||
      |X(    dc , dc ,atT , dc , dc , dc , dc )| inactive ||
      %----|----|----|----|----|----|----|-----|----------||
      |X(    dc , T  , T  ,atT , dc , dc , dc )| override ||
      %----|----|----|----|----|----|----|-----|----------||
      |X(    dc , T  , T  , dc , dc ,atT , dc )| override ||
      %----|----|----|----|----|----|----|-----|----------||
      |    ELSE                               | cruise   ||
      %----|----------------------------------|----------||
     ENDTABLE ||

    |override| TABLE
      %----|----|----|----|----|----|----|-----|----------||
      |X(   atF , dc   dc , dc , dc , dc , dc )| off      ||
      %----|----|----|----|----|----|----|-----|----------||
      |X(    T  ,atF , dc , dc , dc , dc , dc )| inactive ||
      %----|----|----|----|----|----|----|-----|----------||
      |X(    dc , T  , dc , F  ,atT , dc , dc )| cruise   ||
      %----|----|----|----|----|----|----|-----|----------||
      |X(    dc , T  , dc , F  , dc , dc ,atT )| cruise   ||
      %----|----|----|----|----|----|----|-----|----------||
      |    ELSE                               | override ||
      % ---|----------------------------------|----------||
     ENDTABLE ||
ENDTABLE
```

Figure 4.4: PVS Version of the Revised Specification of Figure 4.3

Following Atlee and Gannon, we examine certain "mode invariants" of the SCR requirements specification of Figures 4.3 and 4.4. The properties examined by Atlee and Gannon are the following.[10]

1. When the mode is `off`, the ignition is off (i.e., `ignited` is *false*).
2. In modes other than `off`, the ignition is on (i.e., `ignited` is *true*).
3. In `inactive` mode, either the engine is not `running` or the cruise control is not `activated`.
4. In `cruise` mode, the engine is `running`, the vehicle is not going `toofast`, the `brake` is not on, and `deactivate` is not selected.
5. In `override` mode, the engine is `running`.

All of these can be expressed in CTL as AG properties as follows.

1. $AG((mode = \mathit{off}) \Rightarrow \neg ignited)$
2. $AG((mode \neq \mathit{off}) \Rightarrow ignited)$
3. $AG((mode = inactive) \Rightarrow (\neg running \vee \neg active))$
4. $AG((mode = cruise) \Rightarrow running \wedge \neg brake)$
5. $AG((mode = override) \Rightarrow running)$

In PVS, a CTL formula is specified by an expression of the following form.

```
AG( transition_relation,  predicate )( state )
```

This example uses the `AG` operator to assert that the ***predicate*** is true on all paths induced by the given ***transition_relation*** from the specified ***state***. (All the other CTL operators, as well as their fair variants, are available in PVS.) Usually, such expressions appear as the conclusion to an implication whose antecedent asserts properties of the specified initial ***state***. Usually, too, the ***predicate*** is defined in place by means of a `LAMBDA` abstraction. For example, if `init` characterizes the initial state, the first invariant above would be specified in PVS as follows.

[10]By virtue of the second of these properties, we have eliminated the clause "and the ignition is on (i.e., *ignited* is *true*)" from Atlee and Gannon's statements of the third, fourth, and fifth properties.

```
IMPORTING MU@ctlops, cruise_tab

p,q,r: var state

trans: transition_relation = trans(deterministic)

init(p): bool = off?(p) & NOT ignited(p)

safe1: THEOREM init(p)
         => AG(trans, (LAMBDA q: off?(q) => NOT ignited(q)))(p)
```

Here, `cruise_tab` is the PVS theory that defines the mode table concerned, and `ctlops` is the PVS library theory that defines the CTL operators. The `MU@ctlops` construction indicates that it can be found in the file `MU.pvs` in the directory containing the standard PVS libraries (these are distributed with PVS).

Next, we apply the function `trans` (from the `scr` theory) to the mode table `deterministic` to construct a transition relation (also called `trans`). Then we characterize the initial state as one whose mode is `off` and in which the engine is not `ignited`, and state the theorem corresponding to the formulas numbered 1 above.

When all the types involved are finite, formulas such as this can be proved using the PVS model checker by first setting up all the theories concerned as auto-rewrites, and then giving the `(model-check)` command. This will rewrite all the defined terms down to their primitive forms, reduce CTL operations to expressions in Park's μ-calculus, and then call an external BDD-based μ-calculus decision procedure. If the theorem is true, the decision procedure will so report it. If it is not, the proof will terminate unsuccessfully. PVS does not, at present, return a falsifying trace in the case of untrue CTL conjectures. The theorem `safe1` is indeed verified by the model checker using the following prover commands.

```
(auto-rewrite-theories
    ("scr" :defs t) "cruise" "cruise_tab" "cruise_test")
(model-check)
```

Here, `scr` is the PVS generic SCR theory, `cruise` is the theory that specifies the types used for the cruise control example, `cruise_tab` is the theory that specifies the mode table `deterministic`, and `cruise_test` is the theory containing the definitions we have given for `trans` and `init`. The `:defs t` qualifier for the `scr` theory instructs the prover to rewrite only definitions (as opposed to all conditional equations of the right form), and is important because PVS can figure out the correct theory instantiation on the fly in this case. Without this qualifier, it would be necessary to explicitly specify the required instance(s) of the `scr` theory in the `auto-rewrite-theories` command.

The other four CTL properties listed above are specified by the following PVS formulas.

```
safe2: THEOREM init(p)
         => AG(trans, (LAMBDA q: NOT off?(q) => ignited(q)))(p)

safe3: THEOREM init(p)
         => AG(trans,
                (LAMBDA q: inactive?(q) =>
                    NOT running(q) OR NOT activate?(q)))(p)

safe4: THEOREM init(p)
         => AG(trans,
                (LAMBDA q: cruise?(q) =>
                    running(q) & NOT toofast(q)
                       & NOT brake(q) & NOT deactivate?(q)))(p)

safe5: THEOREM init(p)
         => AG(trans, (LAMBDA q: override?(q) => running(q)))(p)
```

Theorems `safe2` and `safe5` are proved by model checking in the same way as `safe1`, but `safe3` and `safe4` fail. The failure to prove `safe4` motivates closer examination of the specification—this reveals that although `cruise` mode is exited when `toofast` goes *true*, the transitions into `cruise` mode neglect to check that `toofast` is *false* before making the transition. The correction is to add the condition `F(toofast)` to the three transitions into `cruise` mode. The corrected specification is shown in Figures 4.5 and 4.6.

The problem with conjecture `safe3` is of a different kind from that with the theorem `safe4`. Examination of the specification reveals that `safe3` is false because, for example, it is possible for `ignited`, `running`, and `activate` to become true simultaneously when the system is in `off` mode. This will cause a transition to `inactive` mode in a state that violates the invariant of `safe3`. Contemplation of the intent of the specification suggests that this is acceptable: it is not the transition relation that is wrong, but our formulation of the intended invariant for `inactive` mode. Atlee [1, page 9] suggests that a more appropriate invariant is one that states that if the current mode is `inactive` and the invariants for `cruise` mode apply when `activate` goes *true*, then the next mode will not be `inactive`. This can be expressed by the following formula, which is shown to be a theorem by the PVS model checker.

Current Mode	Conditions							Next Mode
	Ignited	Running	Toofast	Brake	Activate	Deactivate	Resume	
Off	@T	-	-	-	-	-	-	Inactive
Inactive	@F	-	-	-	-	-	-	Off
	-	T	F	F	@T	-	-	Cruise
Cruise	@F	-	-	-	-	-	-	Off
	T	@F	-	-	-	-	-	Inactive
	-	-	@T	-	-	-	-	Inactive
	-	T	F	@T	-	-	-	Override
	-	T	F	-	-	@T	-	Override
Override	@F	-	-	-	-	-	-	Off
	T	@F	-	-	-	-	-	Inactive
	-	T	F	F	@T	-	-	Cruise
	-	T	F	F	-	-	@T	Cruise

Figure 4.5: Corrected Mode Transition Table for Cruise Control

```
trans:transition_relation = trans(corrected)

safe6: THEOREM init(p)
     => AG(trans, (LAMBDA q:
             inactive?(q) & ignited(q) & running(q)
               & NOT toofast(q) & NOT brake(q) & NOT activate?(q))
           IMPLIES
               NOT EX(N, (LAMBDA r: inactive?(r) & ignited(r)
                        & running(r) & NOT toofast(r) & NOT brake(r)
                           & activate?(r))))(p)
```

Perhaps the most interesting feature here is not the utility of this particular formula, but its exemplification of nested CTL operators.

The most interesting feature of the overall exercise, however, is its integrated character: completeness and consistency checking, model checking of application properties, and (although we did not demonstrate these) direct evaluation of test cases and proof of general properties are all driven from the same specification. This is not only more convenient than, say, the translation to the language of the SMV model checker employed by Atlee, but it provides the assurance of working within a single semantics. Because the different methods of analysis are integrated and share a common semantics in PVS, they can be used in combination, so that arbitrary theorem proving (and not just propositional tautology checking) can be used to settle consistency checks, and theorem proving can be used to augment model checking in difficult cases. Furthermore, it is not just different methods of analysis that can brought to bear: the full resources of the PVS language are available within table entries, and other methods of specification can be combined with tabular forms. In the next section, we will exploit this capability to allow tables representing SCR

```
  corrected(s: modes, (p, q: monitored_vars)): modes =
  LET
    x: conds7 = (ignited, running, toofast, brake, activate, deactivate, resume),
    X = (LAMBDA (a,b,c,d,e,f,g:EC): PC(a,b,c,d,e,f,g)(x)(p,q))
  IN TABLE s
   |off| TABLE
      %----|----|----|----|----|----|----|----|-----------||
      |X(   atT , dc  , dc  , dc  , dc  , dc  , dc)| inactive  ||
      %----|----|----|----|----|----|----|----|-----------||
      |    ELSE                               | off       ||
      %----|----------------------------------|-----------||
    ENDTABLE ||

   |inactive| TABLE
      %----|----|----|----|----|----|----|-----|----------||
      |X(   atF , dc , dc , dc , dc , dc , dc )| off      ||
      %----|----|----|----|----|----|----|-----|----------||
      |X(    dc , T  , F  , F  ,atT , dc , dc )| cruise   ||
      %----|----|----|----|----|----|----|-----|----------||
      |    ELSE                                | inactive ||
      %----|-----------------------------------|----------||
     ENDTABLE ||

   |cruise| TABLE
      %----|----|----|----|----|----|----|-----|----------||
      |X(   atF,  dc,  dc,  dc,  dc,  dc,  dc )| off      ||
      %----|----|----|----|----|----|----|-----|----------||
      |X(    T  ,atF , dc , dc , dc , dc , dc )| inactive ||
      %----|----|----|----|----|----|----|-----|----------||
      |X(    dc , dc ,atT , dc , dc , dc , dc )| inactive ||
      %----|----|----|----|----|----|----|-----|----------||
      |X(    dc , T  , T  ,atT , dc , dc , dc )| override ||
      %----|----|----|----|----|----|----|-----|----------||
      |X(    dc , T  , T  , dc , dc ,atT , dc )| override ||
      %----|----|----|----|----|----|----|-----|----------||
      |    ELSE                                | cruise   ||
      %----|-----------------------------------|----------||
     ENDTABLE ||

    |override| TABLE
      %----|----|----|----|----|----|----|-----|----------||
      |X(   atF , dc   dc , dc , dc , dc , dc )| off      ||
      %----|----|----|----|----|----|----|-----|----------||
      |X(    T  ,atF , dc , dc , dc , dc , dc )| inactive ||
      %----|----|----|----|----|----|----|-----|----------||
      |X(    dc , T  , F  , F  ,atT , dc , dc )| cruise   ||
      %----|----|----|----|----|----|----|-----|----------||
      |X(    dc , T  , F  , F  , dc , dc ,atT )| cruise   ||
      %----|----|----|----|----|----|----|-----|----------||
      |    ELSE                                | override ||
      % ---|-----------------------------------|----------||
     ENDTABLE ||
ENDTABLE
```

Figure 4.6: PVS Version of the Corrected Specification of Figure 4.5

transition relations to be combined using ordinary relational composition and we will show how to establish that different transition relations induce identical behavior.

4.3 Interacting Transition Specifications

The cruise control example has only a single mode transition table. Matters can become much more complex when multiple, interacting transition systems are considered. In the case of Statecharts, for example, von der Beeck [44] identifies 21 different proposed semantics—most of these differ only in their treatment of interacting systems. One of the central difficulties is that of accounting for transitions due to internal events. In the cruise control example, it was understood that events such as `brake` becoming *true*, or `running` becoming *false*, happen in the external environment; with interacting systems, however, an event may be a mode transition in another system. Thus, a transition in one system may trigger one in another, leading to a potentially infinite cycle of activity without reference to the external environment. In the SCR method, such potential cycles are broken by requiring that events are ordered in some way. Here, we consider a simple example and show that the resources of PVS allow transition relations to be composed in a variety of ways. We argue that rather than build the treatment of interaction into the methodology, it may be best to allow this to be specified directly.

4.3.1 A Requirements Specification

Our example derives from an autopilot specification developed by Ricky Butler of NASA Langley Research Center [7]. Whereas an automobile's cruise control is concerned with only a single attribute—speed—an autopilot is responsible for many attributes and the functions controlling the different attributes interact with one another. For example, one function is responsible for acquiring and holding a particular altitude, while another is responsible for climbing at a particular rate. If a pilot dials in a desired altitude significantly higher than the present altitude, the altitude function does not become active immediately—a desired rate of climb must also be specified. Similarly, dialing in a desired rate of climb does not cause the plane immediately to start climbing—a target altitude must also be specified. Thus, these two functions cannot become active independently. When only one of them is selected, it is held in an intermediate "armed" state; when the other is selected, both jump to the "on" state. Conversely, if one of them is subsequently deselected, the other must drop back from its "on" to its "armed" state.

We might attempt to write a requirements specification for this behavior in which each component ("altitude level" and "climb angle") is specified as a separate transition system whose transitions are partly contingent on those of the other.

The complexity in this treatment would be compounded if we also desire that the individual specifications are those of components that could be developed independently. In this case, the individual specifications must serve a double duty: their composition must specify the behavior required of the overall system, and they must also serve as specifications of components. It seems to us that this approach conflates the issue of overall requirements specification with that of refinement to an implementation. We prefer to specify the overall requirement as the interaction of separate transition systems that are chosen for simplicity and clarity of expression, rather than because they correspond to components in an implementation. We can then develop a separate implementation specification and show that it induces the same behavior as the requirement specification.

We exemplify this approach with a drastically simplified version of the autopilot. We will have two attributes: `climb` and `level`, both of which may be either `off`, `armed`, or `on`. Each attribute is controlled by a separate button. If the `climb` attribute is `off` when its button is pressed, then it may change to either the `armed` or `on` states; otherwise, it stays `off`. If the button is pressed when the `climb` attribute is either `armed` or `on`, then the attribute is turned `off`; otherwise (if the button is not pressed) it may nondeterministically transition between either of these states (the nondeterminism will be resolved later). The `level` attribute is specified dually. Notice that these two specifications are completely independent: that for `climb` does not mention `level`, nor vice versa. If we specify the overall system as the disjunction of the separate `climb` and `level` specifications, then the overall system includes all the behaviors we require, but also many that we do not. We complete the specification by simply conjoining it with one that excludes the undesired behaviors: namely, one that says that `climb` is `on` if and only if `level` is also `on`, and that `climb` and `level` cannot both be `armed`.

A PVS rendition of this specification in SCR-style is shown in Figures 4.7 and 4.8. `Coff`, `Carm`, and `Con` indicate whether the `climb` attribute is `off`, `armed`, or `on`, respectively, and `Cbutton` indicates whether its button is pressed. `Loff`, `Larm`, `Lon`, and `Lbutton` perform dual roles for the `level` attribute.

The individual transition relations are specified directly as `Ctransition` and `Ltransition` rather than indirectly by means of mode tables. This is because the transitions are deliberately nondeterministic: the next mode for `Ctransition` in the `off` mode when the `Cbutton` is pressed is *either* the `arm` or the `on` mode. If we tried to specify this as a mode transition table, we would get unprovable TCCs, since such a table is intended to define the new mode as a *function* of the present one and the events. By using disjunctions in the "action" parts of the tables specifying the transition relations, we make the nondeterminism explicit and no TCCs are generated. Notice that, for variety, `Ctransition` explicitly enumerates over all modes, whereas `Ltransition` collapses the cases for `armed` and `on` into the `ELSE` case.

```
linkedmodes: THEORY
BEGIN

   modes: TYPE = {off, armed, on}

  combined_modes: TYPE = [# climbmode, levelmode:  modes #]

  m, n: VAR combined_modes

  Coff(m): bool =  off?(climbmode(m))
  Carm(m): bool =  armed?(climbmode(m))
  Con(m):  bool =  on?(climbmode(m))

  Loff(m): bool =  off?(levelmode(m))
  Larm(m): bool =  armed?(levelmode(m))
  Lon(m):  bool =  on?(levelmode(m))

  monitored_vars: TYPE = [# Cbutton, Lbutton: bool #]

  p, q: VAR monitored_vars

  null: TYPE

  IMPORTING MU@ctlops, scr[monitored_vars, combined_modes, null]

  r, s, t: VAR state

  Cbutton: condition = LAMBDA p: Cbutton(p)
  Lbutton: condition = LAMBDA p: Lbutton(p)
```

Figure 4.7: Preamble to PVS Requirements Specification for Interacting Autopilot Modes

```
Ctransition(s,t): bool =
  LET x: conds2 = (Cbutton, Lbutton),
      X = (LAMBDA (a,b:EC): PC(a,b)(x)(vars(s),vars(t)))
  IN TABLE climbmode(mode(s))
    |off| TABLE
      %---------------------------------%
      |X( atT, dc) | Carm(t) OR Con(t) ||
      |ELSE        | Coff(t)           ||
      %---------------------------------%
      ENDTABLE ||
    |armed| TABLE
      %---------------------------------%
      |X( atT, dc) | Coff(t)           ||
      |ELSE        | Carm(t) OR Con(t) ||
      %---------------------------------%
      ENDTABLE ||
    |on| TABLE
      %---------------------------------%
      |X( atT, dc) | Coff(t)           ||
      |ELSE        | Carm(t) OR Con(t) ||
      %---------------------------------%
      ENDTABLE ||
   ENDTABLE

Ltransition(s, t): bool =
  LET x: conds2 = (Cbutton, Lbutton),
      X = (LAMBDA (a,b:EC): PC(a,b)(x)(vars(s),vars(t)))
  IN TABLE levelmode(mode(s))
    |off| TABLE
      %---------------------------------%
      |X( dc, atT) | Larm(t) OR Lon(t) ||
      |ELSE        | Loff(t)           ||
      %---------------------------------%
      ENDTABLE ||
    |ELSE| TABLE
      %---------------------------------%
      |X( dc, atT) | Loff(t)           ||
      |ELSE        | Larm(t) OR Lon(t) ||
      %---------------------------------%
      ENDTABLE ||
   ENDTABLE

exclude(s): bool = (Con(s) IFF Lon(s)) AND NOT (Carm(s) AND Larm(s))

req(s,t): bool =
    (Ctransition(s,t) OR Ltransition(s,t)) AND exclude(s) AND exclude(t)

init(s): bool = Coff(s) AND Loff(s)
```

Figure 4.8: Transition Relations of PVS Requirements Specification for Interacting Autopilot Modes

The overall transition relation `req` is the disjunction of the `climb` and `level` mode transitions, conjoined with the predicate that excludes undesired states. This predicate, `exclude`, states that the `climb` and `level` attributes must both be `on` together, and cannot both be `armed`. The initial state `init` is specified as one where both `climb` and `level` are `off`.

We can determine that this specification preserves the safety properties we are interested in by checking the following "challenge" theorems

```
safe1: THEOREM init(s) => AG(req, (LAMBDA t: Con(t) => Lon(t)))(s)

safe2: THEOREM init(s) => AG(req, (LAMBDA t: NOT (Carm(t) & Larm(t))))(s)
```

Of course, these are trivially ensured by the specification and can be deduced by inspection, but they are also easily proved by the PVS `(model-check)` command.

More interesting here are liveness properties: we may wonder whether we have not excluded too many behaviors, so that the system can never get to states where both attributes are `on`, or one is `armed` and the other `off`. We can test these expectations by the following formulas (the CTL `EF` operator requires the property to be true at some point on some path).

```
live1: THEOREM init(s) => EF(req, (LAMBDA t: Carm(t) & Loff(t)))(s)

live2: THEOREM init(s) => EF(req, (LAMBDA t: Con(t) & Lon(t)))(s)
```

These properties are easily be shown to be true by the PVS `(model-check)` command.

We consider that our specification is a clear and direct specification of requirements for the autopilot. The `Ctransition` and `Ltransition` relations separately constrain the possible successors to any state in terms of the `climb` and `level` modes and buttons, and the `exclude` predicate completes the specification by disallowing certain combinations of modes. A more traditional specification would have made the `climb` and `level` transition specifications interdependent in order to exclude the disallowed combinations of modes. We regard such a specification more as a description of an implementation than as a statement of requirements. In the following section, we show how a deterministic implementation can be specified and shown to have the same behavior as our requirements specification.

4.3.2 An Implementation Specification, and Verification of Equivalence

An implementation of the requirements described in Figure 4.8 might have a much more monolithic and sequential character than suggested by the highly nondeterministic requirements specification. One approach might involve two phases. In the first, button presses for `climb` and `level` are processed independently, but deterministically (pressing the `climb` button in `off` mode sends it to `armed` mode; pressing it in `armed` or `on` mode sends it to `off` mode; the button for `level` works similarly). Then, in the second phase, the modes of the two attributes are brought into alignment: if the first phase has resulted in both attributes not being `off`, both are turned `on`; if one is `on` but the other `off`, then `on` changes to `armed`. We can think of button pushes being latched by some underlying operating system component; the state of these latches is examined in the first phase only. Consequently, the state of the buttons is held constant in the second phase. The two phases can be specified in PVS as shown in Figure 4.9. These transition relations are specified in `IF-THEN-ELSE` style, as befits their sequential interpretation. The overloadings in the `LET` clauses simply allow the `mode` field accessor to be omitted in references to the state variables `s` and `t`. The sequential composition of the two phases into the overall implementation transition relation `impl` is specified as ordinary relational composition.

We can check this "implementation" against the same reasonableness checks as the requirements.

```
safe1_i: THEOREM init(s) => AG(impl, (LAMBDA r: Con(r) => Lon(r)))(s)

safe2_i: THEOREM init(s) =>
                      AG(impl, (LAMBDA r: NOT (Carm(r) & Larm(r))))(s)

live1_i: THEOREM init(s) => EF(impl, (LAMBDA r: Carm(r) & Loff(r)))(s)

live2_i: THEOREM init(s) => EF(impl, (LAMBDA r: Con(r) & Lon(r)))(s)
```

The PVS model checker quickly verifies these.

What we really want to know, however, is whether the behavior specified by the implementation relation `impl` satisfies the requirements specification `req`. The relations `impl` and `req` need not be equal or similar *as relations*: what matters is whether they induce similar behaviors. A CTL formula that expresses the property that the behavior of a transition relation `A` is a superset of that of a transition relation `B` (i.e., `A` can do anything `B` can do) is the following.

```
 A, B: VAR transition_relation

 super(A, B)(s:state):bool =
          AG(B, (LAMBDA t: AX(B, (LAMBDA r: A(t,r)))(t)))(s)
```

What this formula says is that it is invariantly the case, at all states `t` reachable by `B` from the state `s`, that any state reachable in one step from `t` by `B`, is also reachable

```
P: VAR condition

Phase1(s, t): bool =
  LET atT = (LAMBDA P: atT(P)(vars(s),vars(t))),
      climbmode = (LAMBDA s: climbmode(mode(s))),
      levelmode = (LAMBDA s: levelmode(mode(s)))
  IN
      IF atT(Cbutton) THEN
         If Coff(s) THEN Carm(t) ELSE Coff(t) ENDIF
      ELSE climbmode(s) = climbmode(t)
      ENDIF
     OR
      IF atT(Lbutton) THEN
         If Loff(s) THEN Larm(t) ELSE Loff(t) ENDIF
      ELSE levelmode(t) = levelmode(s)
      ENDIF

Phase2(s, t): bool =
  LET climbmode = (LAMBDA s: climbmode(mode(s))),
      levelmode = (LAMBDA s: levelmode(mode(s)))
  IN
     IF NOT (Coff(s) OR Loff(s)) THEN Con(t) AND Lon(t)
     ELSIF Coff(s) AND Lon(s) THEN Coff(t) AND Larm(t)
     ELSIF Loff(s) AND Con(s) THEN Loff(t) AND Carm(t)
     ELSE  climbmode(t) = climbmode(s)
           AND levelmode(t) = levelmode(s)
     ENDIF
   AND Cbutton(t) = Cbutton(s)
   AND Lbutton(s) = Lbutton(t)

impl(s, t): bool = (EXISTS r: Phase1(s, r) AND Phase2(r, t))
```

Figure 4.9: PVS Implementation Specification for Autopilot

in one step by `A`. We can then assert that the behavior of `req` is a superset of that of `impl`, and vice versa (i.e., the behaviors induced by the two specifications are the same), by the following formulas.

```
req_impl: LEMMA init(s) => super(req, impl)(s)

impl_req: LEMMA init(s) => super(impl, req)(s)
```

The PVS model checker verifies both of these.

A subtle point in these specifications is the disjunction that appears in the definition of `req` and in the definition of `Phase1`. These indicate interleaving concurrency. It is possible to replace both `OR`s by `AND`s (indicating true concurrency) and still verify all the results in this section. If only one is changed, the safety and liveness results remain true, but only one of the superset properties connecting `req` and `impl` will hold (the behavior of the one using true concurrency will be a strict superset of the other). The ability to represent different models of concurrency is another of the benefits that follows from undertaking this development within a fully general specification and verification environment.

Overall, this "autopilot" specification demonstrates the flexibility provided by a general-purpose system such as PVS: the resources of the system allow us to reproduce methodologies such as SCR when these are appropriate, but also allow us to depart from them when necessary.

Chapter 5

Conclusion

We have shown previously [37] how PVS can be used to discharge the well-definedness proof obligations that arise in Parnas's tabular specification style [33]. Those proof obligations were generated by hand. In this report, we have described the `COND` construct, recently added to PVS, that generates the proof obligations automatically, and the `TABLE` construct that provides a visually appealing rendition of tabular specifications.

These new constructs required no change to the core of PVS; the `COND` construct required small extensions to the typechecker, but none to the prover, and `TABLE` is simply a syntactic variation on `COND`. In the future, we hope to make the implementation of PVS more "open," so that similar customizations can be made very easily.

We have also shown how standard notation for function application can be adapted to provide a tolerable representation for the AND/OR tables used in RSML [29], and then showed how this technique can be combined with the new `TABLE` construct to provide a treatment for the Decision Tables advocated by Sherry [39].

We then described how an independent enhancement to PVS—the incorporation of a decision procedure for Park's μ-calculus and its use to provide CTL model checking [35]—enables properties of finite-state transition systems to be examined automatically. These two developments—tables and model checking—come together to provide support for the Naval Research Laboratory's SCR method for requirements specification [11].

The generic support provided for tables and for model checking in PVS may be compared with the more specialized support provided in tools such as ORA's TableWise [23], NRL's SCR* [18,20], and Leveson and Heimdahl's consistency checker for RSML [17]. Dedicated, lightweight tools such as these are likely to be superior to a heavyweight, generic system such as PVS for their chosen purposes. Our goal in providing these capabilities in PVS is not to compete with specialized tools but to complement them. The generic capabilities of PVS can be used to prototype some of the capabilities of specialized tools (this was done in the development of TableWise), and can also be used to supplement their capabilities when comprehensive theorem proving and model checking power is needed. Heimdahl, for example, has noted that consistency analysis of the TCAS II requirements specification in RSML produced many spurious error reports because only simple propositional reasoning was available [16]. As well as being able to settle more demanding consistency and completeness checks, we have illustrated how the general theorem proving power of PVS can be used to probe tabular specifications by attempting to prove “challenge” theorems. We also showed how the PVS model checker can be used to test properties of the behaviors specified by SCR mode transition tables, and even to establish inclusion or equivalence between the behaviors of different specifications. All these capabilities are available within a common framework and can be used together.

In addition to being of interest to tool developers, we hope that these examples showing how PVS can represent the specification styles of some existing methodologies will encourage PVS users to incorporate these styles in their own specifications where appropriate, and will also help users to develop support for other methodologies within PVS.

The methodologies we have examined here are primarily concerned with requirements specifications for avionics applications. Hoover, Guaspari, and Humenn provide a general examination of the use of formal methods in these applications [24]. Requirements specifications are particularly challenging to formalize: because there is no “higher” specification against which to verify them, it is particularly important that they should be perspicuous and well suited to human review. Tabular forms of expression seem to serve these needs well. But because verification against a higher specification is impossible, we believe that it is also important that requirements specifications should be subjected to a great deal of mechanized analysis. Mechanization is needed for reliability and efficiency and, since requirements specifications evolve continuously, repeatability. Tabular specifications also serve these needs well: their completeness and consistency checks catch many errors very quickly. However, deterministic specifications are not always the most appropriate and it is important not to become overly committed to a single style of specification. In our deliberately nondeterministic “autopilot” example, we were able to retain the tabular style of an SCR specification, while consciously eschewing its normal consistency checks. We could do his because we had direct access to the representation and model of computation employed. We were able to resolve the nondeterminism using other resources of the PVS specification language (conjunction and disjunction of transition relations) and to explore the specification using model checking. This ability to depart from the "standard" SCR approach would be absent from tools dedicated to that standard approach.

In future work, we plan to examine use of nondeterministic state transition relations for top-level requirements specification of interacting systems (as in the "autopilot" example) in more detail. We are also considering a sublanguage to PVS based on state transition relations that would serve as a convenient intermediate form for a number of analyses (e.g., simulation, explicit state enumeration, model checking, synthesis) in a variety of application domains (e.g., requirements, hardware, protocols). We also plan to explore the rather different approach to requirements specifications used in synchronous dataflow languages, exemplified by Lustre [13].

Bibliography

Most papers by SRI authors can be retrieved from `http://www.csl.sri.com/fm.html`. PVS specification files for several of the examples used here can be downloaded from `http://www.csl.sri.com/pvs/examples/tables`; PVS itself is available at `http://www.csl.sri.com/pvs.html`.

[1] Joanne M. Atlee. Native model-checking of SCR requirements. In *Fourth International SCR Workshop*, Washington, DC, November 1994. Naval Research Laboratory.

[2] Joanne M. Atlee and Michael A. Buckley. A logic-model semantics for SCR software requirements. In Steven J. Zeil, editor, *International Symposium on Software Testing and Analysis (ISSTA)*, pages 280–292, San Diego, CA, January 1996. Association for Computing Machinery.

[3] Joanne M. Atlee and John Gannon. State-based model checking of event-driven system requirements. In *SIGSOFT '91: Software for Critical Systems*, pages 16–28, New Orleans, LA, December 1991. Published as ACM SIGSOFT Engineering Notes, Volume 16, Number 5.

[4] Joanne M. Atlee and John Gannon. State-based model checking of event-driven system requirements. *IEEE Transactions on Software Engineering*, 19(1):24–40, January 1993.

[5] Randal E. Bryant. Symbolic boolean manipulation with ordered binary-decision diagrams. *ACM Computing Surveys*, 24(3):293–318, September 1992.

[6] 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, June 1992.

[7] Ricky W. Butler. An introduction to requirements capture using PVS (specification of a simple autopilot). NASA Technical Memorandum 110255, NASA Langley Research Center, Hampton, VA, June 1996. Forthcoming.

[8] J. H. Cheng and C. B. Jones. On the usability of logics which handle partial functions. In Carroll Morgan and J. C. P. Woodcock, editors, *Proceedings of the Third Refinement Workshop*, pages 51–69. Springer-Verlag Workshops in Computing, 1990.

[9] E. M. Clarke, E. A. Emerson, and A. P. Sistla. Automatic verification of finite-state concurrent systems using temporal logic specifications. *ACM Transactions on Programming Languages and Systems*, 8(2):244–263, April 1986.

[10] Edmund M. Clarke, Orna Grumberg, and David E. Long. Model checking and abstraction. *ACM Transactions on Programming Languages and Systems*, 16(5):1512–1542, September 1994.

[11] S. Faulk and P. Clements. The NRL Software Cost Reduction (SCR) requirements specification methodology. In *Fourth International Workshop on Software Specification and Design*, Monterey, CA, April 1987. IEEE Computer Society.

[12] Stuart Faulk, John Brackett, Paul Ward, and James Kirby, Jr. The CoRE method for real-time requirements. *IEEE Software*, 9(5):22–33, September 1992.

[13] N. Halbwachs, P. Caspi, P. Raymond, and D. Pilaud. The synchronous dataflow programming language Lustre. *Proceedings of the IEEE*, 79(9):1305–1320, September 1991.

[14] D. Harel et al. STATEMATE: A working environment for the development of complex reactive systems. *IEEE Transactions on Software Engineering*, 16(4):403–414, April 1990.

[15] Zvi Har'El and Robert P. Kurshan. Software for analytical development of communications protocols. *AT&T Technical Journal*, 69(1):45–59, January/February 1990.

[16] Mats P. E. Heimdahl. Experiences and lessons from the analysis of TCAS II. In Steven J. Zeil, editor, *International Symposium on Software Testing and Analysis (ISSTA)*, pages 79–83, San Diego, CA, January 1996. Association for Computing Machinery.

[17] Mats P. E. Heimdahl and Nancy G. Leveson. Completeness and consistency analysis of state-based requirements. In *17th International Conference on Software Engineering*, pages 3–14, Seattle, WA, April 1995. IEEE Computer Society.

[18] Constance Heitmeyer, Alan Bull, Carolyn Gasarch, and Bruce Labaw. SCR*: A toolset for specifying and analyzing requirements. In COMP [25], pages 109–122.

[19] Constance Heitmeyer, Ralph Jeffords, and Bruce Labaw. Tools for analyzing SCR-style requirements specifications: A formal foundation. Technical Report 7499, Naval Research Laboratory, Washington DC, 1995. In press.

[20] Constance Heitmeyer, Bruce Labaw, and Daniel Kiskis. Consistency checking of SCR-style requirements specifications. In *International Symposium on Requirements Engineering*, York, England, March 1995. IEEE Computer Society.

[21] K. L. Heninger. Specifying software requirements for complex systems: New techniques and their application. *IEEE Transactions on Software Engineering*, SE-6(1):2–13, January 1980.

[22] K. L. Heninger et al. Software requirements for the A-7E aircraft. NRL Report 3876, Naval Research Laboratory, November 1978.

[23] D. N. Hoover and Zewei Chen. Tablewise, a decision table tool. In COMP [25], pages 97–108.

[24] D. N. Hoover, David Guaspari, and Polar Humenn. Applications of formal methods to specification and safety of avionics software. NASA Contractor Report 4723, NASA Langley Research Center, Hampton, VA, April 1996. (Work performed by Odyssey Research Associates).

[25] *COMPASS '95 (Proceedings of the Ninth Annual Conference on Computer Assurance)*, Gaithersburg, MD, June 1995. IEEE Washington Section.

[26] Ryszard Janicki. Towards a formal semantics of Parnas tables. In *17th International Conference on Software Engineering*, pages 231–240, Seattle, WA, April 1995. IEEE Computer Society.

[27] G. L. J. M. Janssen. *ROBDD Software*. Department of Electrical Engineering, Eindhoven University of Technology, October 1993.

[28] L. Lamport. Sometime is sometimes not never. In *10th ACM Symposium on Principles of Programming Languages*, pages 174–185, Austin, TX, January 1983.

[29] 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.

[30] Kenneth L. McMillan. *Symbolic Model Checking*. Kluwer Academic Publishers, Boston, MA, 1993.

[31] David Park. Finiteness is mu-ineffable. *Theoretical Computer Science*, 3:173–181, 19.

[32] David Lorge Parnas. Tabular representation of relations. Technical Report CRL Report 260, Telecommunications Research Institute of Ontario (TRIO), Faculty of Engineering, McMaster University, Hamilton, Ontario, Canada, October 1992.

[33] David Lorge Parnas. Some theorems we should prove. In Jeffrey J. Joyce and Carl-Johan H. Seger, editors, *Higher Order Logic Theorem Proving and its Applications (6th International Workshop, HUG '93)*, number 780 in Lecture Notes in Computer Science, pages 155–162, Vancouver, Canada, August 1993. Springer-Verlag.

[34] Vaughan Pratt. Anatomy of the Pentium bug. In *TAPSOFT '95: Theory and Practice of Software Development*, number 915 in Lecture Notes in Computer Science, pages 97–107, Aarhus, Denmark, May 1995. Springer-Verlag.

[35] S. Rajan, N. Shankar, and M.K. Srivas. An integration of model-checking with automated proof checking. In Pierre Wolper, editor, *Computer-Aided Verification, CAV '95*, volume 939 of *Lecture Notes in Computer Science*, pages 84–97, Liege, Belgium, June 1995. Springer-Verlag.

[36] H. Rueß, N. Shankar, and M.K. Srivas. Modular verification of SRT division. In R. Alur and T.A. Henzinger, editors, *Computer-Aided Verification, CAV '96*, Lecture Notes in Computer Science, New Brunswick, NJ, July 1996. Springer-Verlag. To appear.

[37] John Rushby and Mandayam Srivas. Using PVS to prove some theorems of David Parnas. In Jeffrey J. Joyce and Carl-Johan H. Seger, editors, *Higher Order Logic Theorem Proving and its Applications (6th International Workshop, HUG '93)*, number 780 in Lecture Notes in Computer Science, pages 163–173, Vancouver, Canada, August 1993. Springer-Verlag.

[38] Lance Sherry. Apparatus and method for controlling the vertical profile of an aircraft. United States Patent 5,337,982, August 16, 1994.

[39] Lance Sherry. A structured approach to requirements specification for software-based systems using operational procedures. In *13th AIAA/IEEE Digital Avionics Systems Conference*, pages 64–69, Phoenix, AZ, October 1994.

[40] Tirumale Sreemani and Joanne M. Atlee. Feasibility of model checking software requirements. In *COMPASS '96 (Proceedings of the Ninth Annual Conference on Computer Assurance)*, Gaithersburg, MD, June 1996. IEEE Washington Section. To appear.

[41] G. S. Taylor. Compatible hardware for division and square root. In *Proceedings of the 5th Symposium on Computer Arithmetic*, pages 127–134. IEEE Computer Society, 1981.

[42] A. John van Schouwen. The A-7 requirements model: Re-examination for real-time systems and an application to monitoring systems. Technical Report 90-276, Department of Computing and Information Science, Queen's University, Kingston, Ontario, Canada, May 1990.

[43] A. John van Schouwen, David Lorge Parnas, and Jan Madey. Documentation of requirements for computer systems. In *IEEE International Symposium on Requirements Engineering*, pages 198–207, San Diego, CA, January 1993.

[44] Michael von der Beeck. A comparison of statecharts variants. In H. Langmaack, W.-P. de Roever, and J. Vytopil, editors, *Formal Techniques in Real-Time and Fault-Tolerant Systems*, volume 863 of *Lecture Notes in Computer Science*, pages 128–148, Lübeck, Germany, September 1994. Springer-Verlag.