Slightly expanded version of a paper presented at the Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS '97), Enschede, The Netherlands, April 1997. Springer-Verlag Lecture Notes in Computer Science Vol. 1217, pp. 366–383.

Integration in PVS: Tables, Types, and Model Checking*

Sam Owre, John Rushby, Natarajan Shankar

Computer Science Laboratory, SRI International,
Menlo Park, CA 94025, USA

Abstract. We have argued previously that the effectiveness of a verification system derives not only from the power of its individual features for expression and deduction, but from the extent to which these capabilities are integrated: the whole is more than the sum of its parts [20, 21]. Here, we illustrate this thesis by describing a simple construct for tabular specifications that was recently added to PVS. Because this construct integrates with other capabilities of PVS, such as typechecker-generated proof obligations, dependent typing, higher-order functions, model checking, and general theorem proving, it can be used for a surprising variety of purposes. We demonstrate this with examples drawn from hardware division algorithms and requirements specifications.

1 Introduction

Persuaded by the advocacy of David Parnas and others [15], we recently added a construct for tabular specification to PVS [12]. The construct generates proof obligations to ensure that the conditions labeling the rows and columns are disjoint and exclusive. This simple capability has been found useful by colleagues at NASA and Lockheed-Martin, who applied it in requirements analysis for Space Shuttle flight software [2, 18]. The capability becomes rather richer in the presence of dependent typing, and in this form it has been used to verify the accessible region in a quotient lookup table for SRT division [19]. When combined with other features of the PVS specification language, the table construct provides some of the attractive attributes of the TableWise [8] and SCR [6] specification methods. Because these constructions are performed in the context of a full verification system, we are able to use its theorem prover and model checker to establish invariant and reachability properties of the specifications concerned, and are able also to compose specifications described by separate tables and to establish refinement and equivalence relations between state machines specified in this manner.

* This work was supported by the Air Force Office of Scientific Research, Air Force Materiel Command, USAF, under contract F49620-95-C0044 and by the National Science Foundation under contract CCR-9509931.

2 Basic Tables

Tables can be a convenient way to specify certain kinds of functions. An example is the function $sign(x)$, 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.

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

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 builds on `COND`.

Before we introduce these constructs, we should mention that the PVS specification language is a higher-order logic that supports both predicate subtypes and dependent types, and that the system provides strong assurances that definitional constructs (such as recursive function definitions) are conservative [13, 14]. Some of the checks necessary to ensure type-correctness and conservative extension are not algorithmically decidable; in these cases, PVS generates Type Correctness Conditions (TCCs), which are obligations that must be discharged by theorem proving. PVS provides a powerful interactive theorem prover that includes decision procedures for linear arithmetic and other theories, and its default strategies are often able to discharge TCCs automatically; in more difficult cases, the user must guide the theorem prover interactively. Specifications with false TCCs are considered malformed and no meaning is ascribed to them. PVS allows proof obligations to be postponed, but keeps track of all unsatisfied obligations; a specification is not considered fully typechecked, and its theorems are considered provisional, until all TCCs have been proved.

2.1 The PVS `COND` Construct

Standard PVS language constructions for specification by cases are the traditional `IF-THEN-ELSE`, and a pattern matching `CASES` expression for enumerating over the constructors of an abstract data type. A `COND` construct has recently been added to these. Its general form is shown in 1, where the c_i are Boolean expressions and the e_i are values of some type t. (PVS has subtypes and overloading, so the types of the individual e_i must be "unified" to yield the common supertype t.) 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 t is allowed.

COND c_1 → e_1, c_2 → e_2, ... c_n → e_n ENDCOND	1
IF c_1 THEN e_1 ELSIF c_2 THEN e_2 ... ELSE e_n ENDIF	2

Exactly one of the c_i is required to be true; because PVS already supports proof obligations in the form of TCCs, it is easy to enforce this requirement by causing each COND to generate two TCCs as follows.

- *Disjointness* requires that each distinct c_i, c_j pair is disjoint.
- *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.

A COND has meaning only if its TCCs are true, in which case the general COND expression of 1 is assigned the same meaning as (and is treated internally as) the IF-THEN-ELSE construction shown in 2. Notice that the condition c_n does not appear in the IF-THEN-ELSE translation: if this condition 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. Because COND is treated internally as an IF-THEN-ELSE, reasoning involving COND requires no extensions to the PVS theorem prover.

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 fractions of a second.

```
% 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 specification that uses an `ELSE` in place of the condition `x > 0` generates a simpler disjointness TCC (just the first of the three conjuncts in `sign_cond_TCC2`), and no coverage TCC.

2.2 The PVS `TABLE` Construct

PVS has `TABLE` constructs that provide a fairly attractive input syntax for one- and two-dimensional tables and that are LaTeX-printed as true tables (the example *Parnas_Fig1* that appears later illustrates this). Their semantic treatment derives directly from the `COND` construct.

2.2.1 One-Dimensional Tables. The simplest tables in PVS are one-dimensional. In their *vertical* format, 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 `TABLE` shown in 3. Note that the horizontal lines are simply comments (comments in PVS are introduced by `%`).

3
```
sign_vtable(x): signs = TABLE
            %-------------%
            | x < 0 | -1 ||
            %-------------%
            | x = 0 |  0 ||
            %-------------%
            | x > 0 |  1 ||
ENDTABLE    %-------------%
```

4
```
sign_htable(x): signs = TABLE
          %--------------------%
          |[ x<0 | x=0 | x>0 ]|
          %--------------------%
          |  -1  |  0  |  1  ||
ENDTABLE  %--------------------%
```

One-dimensional *horizontal* tables present the information in a different order, and use `|[...]|` to alert the parser to this fact, as illustrated in 4.

Both these tabular specifications are equivalent to `sign_cond`, generate exactly the same TCCs, and are treated the same in proofs. Notice that tables require no extensions to the PVS theorem prover, and the full repertoire of proof commands may be applied to constructions involving tables—for example, it is possible to rewrite with an expression whose right hand side is a table. Note, however, 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., one that "lifts" `IF-THEN-ELSE` constructs to the top level) has transformed the structures appearing in a sequent.

2.2.2 Blank Entries. Suppose we reformulated our *sign* example to take a natural number, rather than an integer, as its argument. The $x < 0$ case can no longer arise and can be omitted from the table. In some circumstances, however, we may wish to make it patently clear that this case should not occur and we can do this by including the case, but with a blank entry for the value of the expression.

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

The presence of blank entries changes the coverage TCC: this must now ensure that the disjunction of all the conditions with non-blank entries is true. Notice this requires a TCC to be generated even when an `ELSE` case is present.

In one-dimensional tables, blank entries can always be removed by simply deleting the entire case; this is not so with two-dimensional tables, however, where the accessibility of an entry may depend on the conditions labeling *both* its row and column. We describe an example later.

2.2.3 Enumeration Tables. These are a syntactic variation that provide more succinct representation when the conditions to a table are all of the form $x = expression$ for some single identifier x. In an enumeration table, the identifier concerned follows the `TABLE` keyword, and the conditions of the table simply list the *expressions*; a two-dimensional example appears below in 5.

Enumeration tables are an important special case because their TCCs are often easily decidable, and this allows some important optimizations. Observe that the number of conjuncts in a disjointness TCC grows as the square of the number of conditions; when enumerating over the values of an enumeration type, it is not uncommon to have tens or hundreds of conditions, and thus thousands of conjuncts in the disjointness TCC. It is unwieldy and slow to display such massive TCCs to the user. PVS therefore recognizes this case and treats it specially: when the expressions in an enumeration table are all constructors of a single datatype (and the values of an enumeration type are exactly these), the disjointness and coverage conditions are trivially decidable and are checked internally by the typechecker, which also translates such tables into a datatype `CASES` expression, rather than a `COND`.[2] Another special case arises when the expressions of an enumeration table are all literal values of some type (the usual case is values from some range of integers); again, the disjointness TCC is easily decidable and can be checked internally by the typechecker (the coverage TCC can require theorem proving and is generated normally). A table is immediately flagged as illegal if such internal checks reveal a false TCC.

2.2.4 Two-Dimensional Tables. 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 of a two-dimensional enumeration table in which the rows enumerate the values of a type `state` and the columns enumerate the values of a type `input`.

[2] The prover can provide greater automation for the `CASES` expression. The user could use a `CASES` construct directly in the one-dimensional case; the main benefit in providing the translation automatically is with two-dimensional tables.

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

This translates internally 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, apparently equivalent, translation.

```
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 [15, Figure 1] that illustrates this. We exhibit this example in the form output by the PVS LaTeX-printer.

Parnas_Fig1$((y, x : \text{ real})) : \text{ real } =$

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

The subtype constraint on the argument to the square root function (namely, that it be nonnegative) generates TCCs in the second and third rows that are true only when the corresponding row constraints are taken into account. The `LET` form translation loses this information. The advantage of the simple translation, which is the one used in PVS, is that it provides more precise (i.e., weaker but still adequate) TCCs, and therefore admits more specifications.

2.3 Applications

The PVS table constructs described above have been used in several applications performed by ourselves and others—indeed, some elements in the PVS treatment of tables (notably, blank entries, and the optimizations for enumeration tables) evolved in response to these applications.

In one application, PVS is being employed in analysis of new requirements documented in "Change Requests" (CRs) for the flight software of the Space Shuttle. This work is undertaken as part of a project involving staff from several NASA Centers (Langley, Johnson, and JPL) and Requirements Analysts (RAs) from the team at Lockheed Martin (formerly IBM) that develops this software. Running alongside what is generally considered an exemplary (though manual) process for requirements review, this experiment provides useful data on the effectiveness of automated formal analyses [2, 18].

One of the CRs focused on improving the display of flight information to Shuttle pilots guiding the critical initial bank onto the "Heading Alignment Cylinder" (HAC) during descent. The CR documented key portions of the required control logic in tabular form, and was readily formalized using PVS tables; a small representative example is reproduced in Appendix A. Attempts to discharge the TCCs generated by these tables immediately indicated the need to document implicit "domain knowledge," including constraints such as "`Major Mode = 305` or `603` implies `iphase` $\leq$ `3`," and "`wowlon` can be true only if `Major Mode = 305` or `603`." Such domain knowledge was incorporated into the specification using dependent predicate subtyping and was gradually extended and refined through an iterative process that relied on the automated strategies for proving TCCs that are built in to PVS.

Observe that proofs of the HAC TCCs could be automated because necessary domain knowledge was supplied through the type system, using predicate and dependent subtyping. For example, the constraints mentioned above were specified as follows (`iphase` and `wowlon` are record fields; notice that the latter has a type that is a subtype of `bool`!).

```
iphase: {p: iphase | (mode = mm602 => p >= 4) AND
                     ((mode = mm305 OR mode = mm603) => p <= 3)}
wowlon: {b: bool | b => (mode = mm305 OR mode = mm603)}
```

The PVS prover can make very effective and automated use of information supplied in this way; a system lacking such a rich type system would probably require an interactive proof to provide the domain knowledge in the form of axioms. (Of course, PVS's decision procedures for linear arithmetic also contributed to the automation of these proofs.)

After incorporating all constraints identified by the RAs, it was found that the conditions for several rows in one table still overlapped, and this led to identification of a missing conjunct in some of the conditions. In addition to

discovery of this error, the requirements analysts felt that explicit identification and documentation of the domain knowledge was a valuable product of the analysis [18].

Another application for PVS tables has been in verification of fast hardware division algorithms. The notorious Pentium `FDIV` bug, which is reported to have cost Intel $475 million, was due to bad entries in the quotient lookup table for an SRT divider. Triangular-shaped regions at top and bottom of these tables are never referenced by the algorithm; the Pentium error was that certain entries believed to be in this inaccessible region, and containing arbitrary data, were, in fact, sometimes referenced during execution [16].

An SRT division algorithm similar to that used in the Pentium has been specified and verified in PVS [19]. The quotient lookup table for this algorithm was specified as a PVS table (reproduced in Appendix B) which uses blank entries to indicate those regions of the table that are believed to be inaccessible. PVS generates 23 coverage TCCs to ensure that these entries will never be encountered; verification of the algorithm (which can be done largely automatically in PVS) then ensures that all the nonblank table entries are correct. Injection of an error similar to that in the Pentium leads to a failed TCC proof whose final sequent is a counterexample that highlights the error [19]. Miner and Leathrum have used this capability of PVS to develop several new SRT tables [11], each in less than three hours.

3 Decision Tables

Decision tables associate Boolean expressions 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 [22], and given mechanized support in TableWise, developed by Hoover and Chen at ORA [8]. The following is a simple decision table (taken from [8, Table 2]).

	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

This table describes the conditions under which each of the four "operational procedures" `Takeoff`, `Climb`, `Climb_Int_Level`, and `Cruise` should be selected. Each of the columns beneath the name of an operational procedure gives a conjunction of conditions under which that procedure should be selected

(where * indicates "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 greater than or equal to `Acc_Alt`, and either `Alt_Capt_Hold` is *false*, or it is *true* and `Alt_Target` is greater than `prev_Alt_Target`. The columns forming a subtable beneath each operational procedure are similar to the AND/OR tables used in the RSML notation of Leveson and colleagues [10].

The PVS `TABLE` construct cannot represent this type of decision table directly: we need some additional mechanism to represent a conjunction such as

$$(\texttt{Flightphase} = \texttt{climb}) \wedge (\texttt{AC_Alt} \geq \texttt{Acc_Alt}) \wedge \neg \texttt{Alt_Capt_Hold}$$

by the compact list given in the third column of the table.

Now the list `(climb, *, GE, false, *)` from that column can be interpreted as the argument list to a function `X` that treats the first element as a function to be applied to `Flightphase`, the second as a function to be applied to the expression `AC_Alt` > `400` and so on, as follows.

```
X(a,b,c,d,e): bool =
    a(Flightphase) & b(AC_Alt > 400) & c(AC_Alt,Acc_Alt)
       & d(Alt_Capt_Hold) & e(Alt_Target,prev_Alt_Target)
```

We can then use this construction to specify the third column of the decision table as the following row from a vertical one-dimensional PVS table; the complete table is shown in Appendix C (taken from [12], where full details may be found).

```
%----------|-------|-------|-------|-------|-----------------%
| X(climb? ,   *   ,  GE   , false ,   *  )| Climb           ||
%----------|-------|-------|-------|-------|-----------------%
```

The functions appearing in the argument list to `X` are defined as follows (note that * is overloaded and that `climb?` is a recognizer for an enumerated type).

```
q: VAR bool                        x, y: VAR nat
false(q): bool = NOT q             GE(x, y): bool = x >= y
    *(q): bool = TRUE               *(x, y): bool = TRUE
```

The disjointness TCC from this table immediately identifies two overlapping cases, while the coverage TCC identifies four that are missing. For example, one of the four unproved sequents[3] from the coverage TCC is the following.

[3] PVS uses a sequent calculus presentation whose interpretation is that the conjunction of formulas above the turnstile line (|------) should imply the disjunction of formulas below the line. The appearance of a formula on one side of the line is equivalent to its negation on the other, and this structural rule is used to eliminate top-level negations. Names with embedded ! characters are Skolem constants derived from variables with the same root name.

```
decision_table_TCC2.1 :                                              6

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

Unproven sequents such as this, with no formulas above the line, indicate the failure to select an operational procedure when all the formulas below the line are false. This one, for example, identifies the failure to consider the case when `AC_Alt` is not greater than `400`, `Alt_Capt_Hold` is *false*, and `AC_Alt` is less than `Acc_Alt`. The six flaws identified in this way are identical to those found in this example by the special-purpose tool TableWise [8].

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. The limitations of these tools are illustrated by Heimdahl [3], who describes spurious error reports when a completeness and consistency checking tool for the AND/OR tables of RSML (developed with Leveson [5]) was applied to TCAS II. These spurious reports were due to the presence of arithmetic and defined functions whose properties are beyond the reach of the BDD-based tautology checker incorporated in the tool. As Heimdahl notes [3, page 81], a theorem prover is needed to settle such properties; he and Czerny are now experimenting with PVS for this purpose [4].

A theorem prover such as PVS can also examine questions beyond simple completeness and consistency. For example, the incompleteness and inconsistencies detected in the example decision table can be remedied by adding an `ELSE` clause and by replacing the second and third "don't care" entries under `Climb_Int_level` by `false` and `LT`, respectively. The TCC generated by this modified specification is proved automatically by PVS, so we may proceed to examine general properties of the decision table. 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_table(cruise, AC_Alt, Acc_Alt,
                 Alt_Target, prev_Alt_Target, Alt_Capt_Hold) = Cruise
```

This is easily proved by PVS's standard `(grind)` strategy. However, when we try the corresponding challenge for the case where `Flightphase` is `climb`, we

discover that the conjecture is not proved, and actually is false in the case where `Alt_Capt_Hold` is *true* and `Alt_Target <= prev_Alt_Target`, thereby exposing a flaw in either our expectations or our formalization of the specification. 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 anomaly just described only if we happened to choose to simulate a case where `Alt_Capt_Hold` is *true* and `Alt_Target <= prev_Alt_Target`.

4 Transition Relations and Model Checking

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. *Transition relations* provide a way to do this, and the SCR method is a way to present such relations in a tabular manner [7].

The following is a typical SCR "mode transition table" (taken from Atlee and Gannon [1, Table 2]). This system, a simplified automobile cruise control, has four modes (`off`, `inactive`, `cruise`, and `override`) and the table describes the conditions under which it makes transitions from one mode to another.

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

An `@T` entry indicates the case where the condition labeling that column changes from *false* to *true*, while `@F` indicates the opposite transition; a `T` entry indicates the case where the condition labeling that column remains *true* through the transition, `F` indicates the case where it remains *false*, and a dash indicates

"don't care." Thus the third row indicates that the system transitions from the `Inactive` mode to the `Cruise` mode if `Activate` goes *true*, while `Ignited` and `Running` remain *true* and `Brake` remains *false*.

To model this type of specification in PVS, we specify a `condition` as a predicate on inputs to the system, then `atT` (which represents `@T`) is a higher order function that takes a condition and returns a relation on pairs of inputs (namely, one that is *true* when the condition is *false* when applied to the first and *true* when applied to the second). The constructions for `atF` (representing `@F`), `T`, `F`, and `dc` (representing "don't care") are specified similarly.

```
scr[ input, mode, output: TYPE ]: THEORY
BEGIN
  condition: TYPE = pred[input]
  p,q: VAR input
  P: VAR condition

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

With these constructions, the mode transition table shown earlier can be represented in PVS as follows (for brevity, we show only the transitions from the `Inactive` mode, corresponding to the second and third rows of the table; the complete table is shown in Appendix D, and full details are given in [12]).

```
event_constructor: TYPE = [condition -> event]
EC: TYPE = event_constructor
PC(A,B,C,D,E,F,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) & F(f)(p,q) & G(g)(p,q)
% Note: PC stands for "pairwise conjunction"

original(s: modes, (p, q: monitored_vars)): modes =
LET
 x = (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
  ...
  |inactive| TABLE %----|----|----|----|----|----|----|-----|----------||
                  |X(   atF , dc , dc , dc , dc , dc , dc )| off      ||
                  %----|----|----|----|----|----|----|-----|----------||
                  |X(    T  , T  , dc , F  ,atT , dc , dc )| cruise   ||
                  %----|----|----|----|----|----|----|-----|----------||
                  |    ELSE                                 | inactive ||
  ENDTABLE ||     %----|-----------------------------------|----------||
  ...
```

Typechecking this specification generates several TCCs; those for the transitions from mode `inactive` are proved automatically, but those from modes `cruise` and `override` are not. These unproved TCCs yield subgoals that pinpoint problems in the specification, rather in the way that [6] identified problems in the decision table. For example, the successor to `cruise` mode is ambiguous in the case where `toofast` and `deactivate` both go from *false* to *true*: the first of these causes a transition to `inactive` mode, while the second causes a transition to `override` mode. Repairing these flaws requires several changes to the table and—as with the Space Shuttle example—adding some "domain knowledge" (such as that `toofast` implies `running`).

Because a mode transition table specifies how the system proceeds from one mode to another, we can examine properties of the computations that this induces. To do this, we first need to derive the transition relation on states that is implicit in a mode table. We identify the instantaneous `state` of the system with its current mode and the current values of its input variables. We specify this as a record in PVS; a transition relation is a predicate on pairs of such states.

```
state: TYPE = [# mode: mode, vars: input #]
transition_relation: TYPE = pred[[state, state]]
```

Recall that a mode transition table has the following signature.

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

We can therefore define a function `trans` that takes a mode table and returns the corresponding state transition relation.

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

The branching time temporal logic CTL provides a convenient way to specify certain properties of the computations induced by a transition relation, and PVS can automatically verify CTL formulas for transition relations over finite types by using a decision procedure for Park's μ-calculus to provide CTL model checking [17]. An example of a property about this specification that can be specified in CTL is the following invariant.

> In `cruise` mode, the engine is `running`, the vehicle is not going `toofast`, the `brake` is not on, and `deactivate` is not selected.

We can examine this property with PVS in the following manner.

```
IMPORTING MU@ctlops, cruise_tab
p,q,r: var state
trans: transition_relation = trans(deterministic)
init(p): bool = off?(p) & NOT ignited(p)

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

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

Here, `cruise_tab` is the PVS theory that defines the mode table `deterministic` (formed by correcting the errors found in the table `original` discussed above), and `ctlops` is the PVS theory (from the library `MU`) that defines the CTL operators. The function `trans` introduced above is applied to the mode table `deterministic` to construct a transition relation (also called `trans`). We characterize the initial state as one whose mode is `off` and in which the engine is not `ignited`, and specify (as `safe4`) the invariant mentioned above (`AG` is the CTL operator meaning "in every reachable state"). Another plausible invariant property is specified by the formula `safe5`. The PVS `model-check` command verifies formula `safe5` but fails on `safe4`. This prompts closer examination of the specification and 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, and PVS is able to verify the formula `safe4` for the corrected specification.

Similar to the TableWise tool for decision tables, Heitmeyer and colleagues have developed the SCR* tool for checking consistency of SCR tabular specifications [6], while Atlee and colleagues have developed a translator that turns SCR tables into a form acceptable to the SMV model checker [23]. These special-purpose tools have the advantage of being closely tailored to their intended uses and are scalable to larger examples than is convenient for the PVS treatment. On the other hand, the PVS treatment required no customized development: it simply builds on capabilities such as tables, higher-order logic, theorem proving, and model checking that are already present in PVS.

Furthermore, the PVS treatment can draw on the full resources of the language and system to combine methods in novel ways, or to conduct customized analyses. For example, we have used a variant of PVS's treatment of SCR tables to specify the nondeterministic mode transitions of interacting "climb" and "level" components in the requirements for a simple "autopilot" [12, section 4.3]. The transitions of the components were specified as separate tables and combined by disjunction (representing interleaving concurrency). The combined specification was then tested against a number of challenge properties using model checking. A deterministic "implementation" specification of the autopilot was constructed from two "phases" using relational composition to specify

sequential execution. This specification was also tested against the challenge properties using model checking. Finally, model checking was used to show that the behaviors induced by the requirements and the implementation specifications are equivalent (this property can be expressed as a CTL formula).

5 Conclusion

We have described PVS's capabilities for representing tabular specifications, illustrated how these interact synergistically with other capabilities such as typechecker-generated proof obligations, dependent typing, higher-order functions, model checking, and general theorem proving, and described some applications. We demonstrated how these capabilities of the PVS language and verification system can be used in combination to provide customized support for existing 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.

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 [8], NRL's SCR* [6,7], and Leveson and Heimdahl's consistency checker for RSML [5]. 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 applying PVS to these problems 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.

Acknowledgments

Examples undertaken by Ricky Butler, Ben Di Vito, and Paul Miner of NASA Langley Research Center, Steve Miller of Collins Commercial Avionics and Harald Rueß of Universität Ulm, and suggestions by Connie Heitmeyer of the Naval Research Laboratory, were instrumental in shaping the PVS table constructs. Comments by the anonymous referees improved the presentation of this paper.

References

Papers by SRI authors are generally available from `http://www.csl.sri.com/fm.html`.

1. 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.

2. Judith Crow and Ben L. Di Vito. Formalizing space shuttle software requirements: Four case studies. Submitted for publication, 1997.
3. 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.
4. Mats P. E. Heimdahl and Barbara J. Czerny. Using PVS to analyze hierarchical state-based requirements for completeness and consistency. In *IEEE High-Assurance Systems Engineering Workshop (HASE '96)*, pages 252–262, Niagara on the Lake, Canada, October 1996.
5. 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.
6. Constance Heitmeyer, Alan Bull, Carolyn Gasarch, and Bruce Labaw. SCR*: A toolset for specifying and analyzing requirements. In COMPASS [9], pages 109–122.
7. 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.
8. D. N. Hoover and Zewei Chen. Tablewise, a decision table tool. In COMPASS [9], pages 97–108.
9. *COMPASS '95 (Proceedings of the Tenth Annual Conference on Computer Assurance)*, Gaithersburg, MD, June 1995. IEEE Washington Section.
10. 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.
11. Paul S. Miner and James F. Leathrum, Jr. Verification of IEEE compliant subtractive division algorithms. In Mandayam Srivas and Albert Camilleri, editors, *Formal Methods in Computer-Aided Design (FMCAD '96)*, volume 1166 of *Lecture Notes in Computer Science*, pages 64–78, Palo Alto, CA, November 1996. Springer-Verlag.
12. Sam Owre, John Rushby, and Natarajan Shankar. Analyzing tabular and state-transition specifications in PVS. Technical Report SRI-CSL-95-12, Computer Science Laboratory, SRI International, Menlo Park, CA, July 1995. Available, with specification files, at `http://www.csl.sri.com/csl-95-12.html`.
13. Sam Owre, John Rushby, Natarajan Shankar, and Friedrich von Henke. Formal verification for fault-tolerant architectures: Prolegomena to the design of PVS. *IEEE Transactions on Software Engineering*, 21(2):107–125, February 1995.
14. Sam Owre and Natarajan Shankar. The formal semantics of PVS. Technical Report SRI-CSL-97-2, Computer Science Laboratory, SRI International, Menlo Park, CA, August 1997.
15. 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.
16. Vaughan Pratt. Anatomy of the Pentium bug. In *TAPSOFT '95: Theory and Practice of Software Development*, volume 915 of *Lecture Notes in Computer Science*, pages 97–107, Aarhus, Denmark, May 1995. Springer-Verlag.
17. 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.

18. Larry W. Roberts and Mike Beims. Using formal methods to assist in the requirements analysis of the Space Shuttle HAC Change Request (CR 90960E). Technical Report JSC-27599, NASA Johnson Space Center, Houston, TX, September 1996.
19. H. Rueß, N. Shankar, and M. K. Srivas. Modular verification of SRT division. In Rajeev Alur and Thomas A. Henzinger, editors, *Computer-Aided Verification, CAV '96*, volume 1102 of *Lecture Notes in Computer Science*, pages 123–134, New Brunswick, NJ, July/August 1996. Springer-Verlag.
20. John Rushby. Mechanizing formal methods: Opportunities and challenges. In Jonathan P. Bowen and Michael G. Hinchey, editors, *ZUM '95: The Z Formal Specification Notation; 9th International Conference of Z Users*, volume 967 of *Lecture Notes in Computer Science*, pages 105–113, Limerick, Ireland, September 1995. Springer-Verlag.
21. Natarajan Shankar. Unifying verification paradigms. In Bengt Jonsson and Joachim Parrow, editors, *Formal Techniques in Real-Time and Fault-Tolerant Systems*, volume 1135 of *Lecture Notes in Computer Science*, pages 22–39, Uppsala, Sweden, September 1996. Springer-Verlag.
22. 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.
23. Tirumale Sreemani and Joanne M. Atlee. Feasibility of model checking software requirements. In *COMPASS '96 (Proceedings of the Eleventh Annual Conference on Computer Assurance)*, pages 77–88, Gaithersburg, MD, June 1996. IEEE Washington Section.

The views and conclusions contained herein are those of the authors and should not be interpreted as necessarily representing the official policies or endorsements, either expressed or implied, of the Air Force Office of Scientific Research or the U.S. Government.

Appendix

A HAC Requirements Table Expressed in PVS

```
switch_position: TYPE = {low, medium, high}
major_mode:      TYPE = {mm301, mm302, mm303, mm304, mm305, mm602, mm603}
iphase:          TYPE = {n: nat | n <= 6} CONTAINING 0

ADI_error_inputs: TYPE =
    [# mode: major_mode,
       switch_position: switch_position,
       iphase: {p: iphase | (mode = mm602 => p >= 4) AND
                            ((mode = mm305 OR mode = mm603) => p <= 3)},
       wowlon: {b: bool | b => (mode = mm305 OR mode = mm603)} #]

ADI_error_scale_deflection(A: ADI_error_inputs) : [real, real, real] =
 LET mode = mode(A), switch_position = switch_position(A),
    iphase = iphase(A), wowlon = wowlon(A) IN
 TABLE  %  Result is of form: [roll error, pitch error, yaw error]
                   ,                 switch_position
                   %-----------------------------------------------------%
                   |[     high      |     medium     |      low       ]|
 %-------------------------------------------------------------------------%
 | mode = mm301 OR
   mode = mm302 OR
   mode = mm303     | (10, 10, 10)   | (5, 5, 5)      | (1, 1, 1)      ||
 %-------------------------------------------------------------------------%
 | mode = mm304 OR
  (mode = mm602 AND
    (iphase = 4 OR
     iphase = 6))   | (25, 5, 5/2)   | (25, 2, 5/2)   | (10, 1, 5/2)   ||
 %-------------------------------------------------------------------------%
 | mode = mm602 AND
   iphase = 5       | (25, 5/4, 5/2) | (25, 5/4, 5/2) | (10, 1/2, 5/2) ||
 %-------------------------------------------------------------------------%
 | (mode = mm305 OR
   mode = mm603) AND
    NOT wowlon      | (25, 5/4, 5/2) | (25, 5/4, 5/2) | (10, 1/2, 5/2) ||
 %-------------------------------------------------------------------------%
 | wowlon           | (20, 10, 5/2)  | (5, 5, 5/2)    | (1, 1, 5/2)    ||
 %-------------------------------------------------------------------------%
ENDTABLE
```

B Quotient Lookup Table for SRT Divider

```
q(D: bvec[3], (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:nat = bv2pattern(D),
    Ptruncp:nat = 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
```

C Example Decision Table

```
 q:VAR bool

  true(q): bool = q
 false(q): bool = NOT q
     *(q): bool = TRUE

 x,y:VAR nat

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

 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
```

D Example SCR Table

```
  event_constructor: TYPE = [condition -> event]
  EC: TYPE = event_constructor
  PC(A,B,C,D,E,F,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) & F(f)(p,q) & G(g)(p,q)
% Note: PC stands for "pairwise conjunction"

  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 , 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
```

This article was processed using the LaTeX macro package with LLNCS style