TACAS: U Twente, Holland, 4 April 1997

Integration in PVS: Tables, Types, and Model Checking

Sam Owre, John Rushby, Natarajan Shankar

Computer Science Laboratory
SRI International
Menlo Park CA USA

Integration: The Key to Effective Automation for Formal Methods

The power and utility of modern verification systems emerge from the tight integration of several elements. For example:

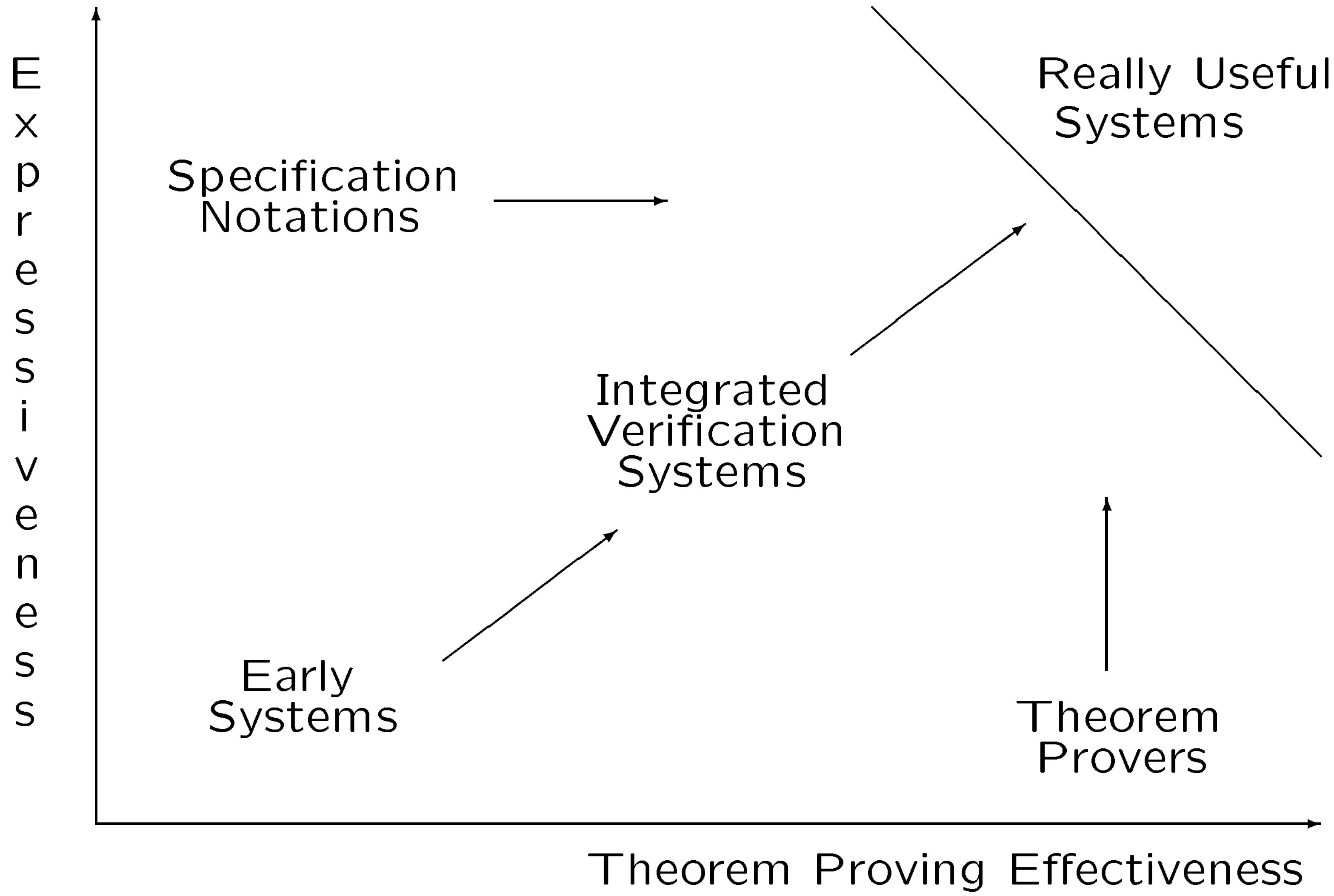
- Language and deduction
- Different kinds of deduction
  - Theorem proving and model checking
- Different theorem proving techniques
  - Integration of several decision procedures with each other
  - And with rewriting
  - And with higher-level strategies

The whole is more than the sum of its parts

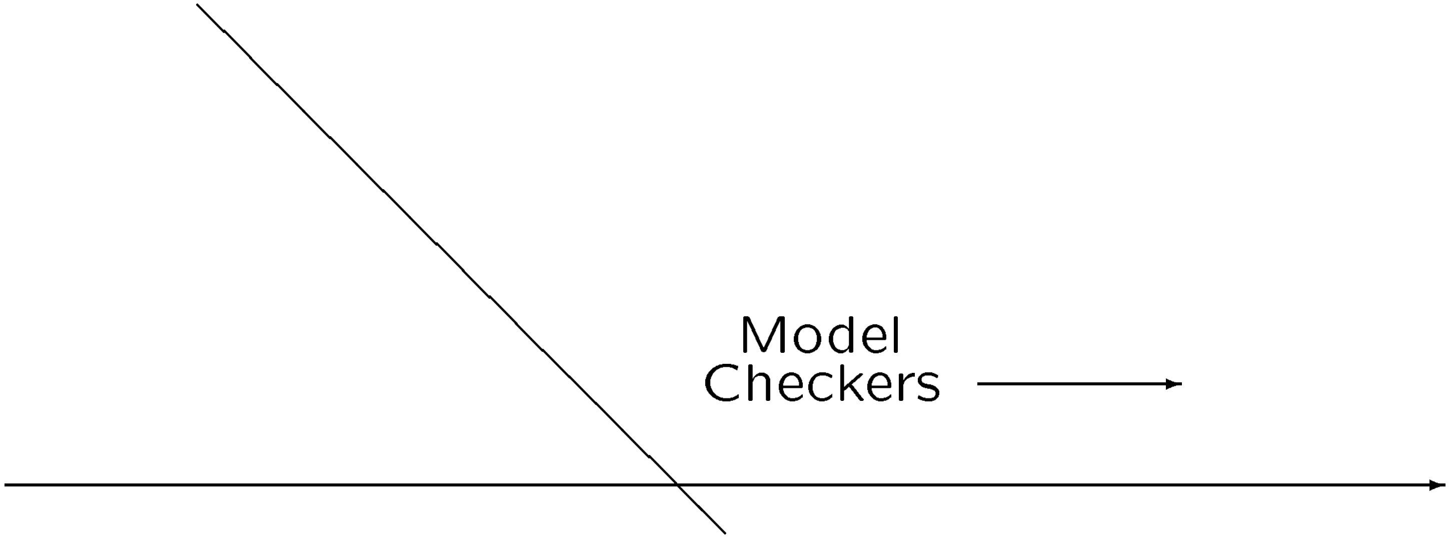
Integration of Language and Deduction

- Tools for formal methods need fast, powerful theorem provers
  - That also help in the discovery of errors
- And they have to do this while supporting an expressive specification language, not a raw logic
- Conventionally, theorem proving power is inversely related to the expressiveness of the underlying logical system
- Reconciling these disparate requirements is a fundamental challenge in building tools for formal methods

The Three Traditions



And A Fourth



Reconciling Language and Deduction

- You cannot design a specification language and then hope to add effective mechanization

- And you cannot build a theorem prover and hope to persuade very many people that its logic is a specification language

- The formal method, its specification language, and its tools have to be designed together
  - Tradeoffs and compromises are necessary. For example
    - ⋆ Effective theorem proving uses congruence closure for efficient treatment of equality:
      This requires that functions are total
    - ⋆ But many specification languages are based on set theory—where functions are inherently partial
  - But new opportunities arise, too

Type Systems for Formal Specification Languages

- Unlike programming languages, type systems for specification languages need not be restricted to those that are trivially decidable
- Can assume a theorem prover is available and allow typechecking to become undecidable
- Provides many of the conveniences of partial functions within a framework based on total function
- Allays many of the concerns of those who dislike traditional strong type systems
- Allows much of the specification to be expressed in the types
    - Contributes to clarity, modularity, and succinctness
    - Typechecking becomes a *very* strong check on consistency, and a potent error detector

Example: Predicate Subtypes

- These are subtypes that are associated with a predicate
  Example (in PVS):

```
even_nat: TYPE = {i: nat | ∃ (j: nat): i = 2×j}
half(e: even_nat): RECURSIVE nat =
  IF e=0 THEN 0 ELSE half(e-2) + 1 ENDIF
 MEASURE e
```

- Expressions of the supertype allowed where the subtype is expected, provided defining predicate can be discharged in that context. For example,

```
half_twice: LEMMA ∀ (i: nat): half(i+i) = i
```

- Generates the proof obligation (called a type-correctness condition, or TCC, in PVS)

```
TCC1: OBLIGATION ∀ (i: nat): ∃ (j:nat): i+i = 2×j
```

Example: Predicate Subtypes and Partial Functions

Many partial functions become total when their domains are specified more precisely

- For example, division can be typed as follows

```
nonzero_real: TYPE = { r: real | r ≠ 0 }

/: [real, nonzero_real → real]
```

Where `[A,...,B -> C]` are total functions from $\mathtt{A} \times \cdots \times \mathtt{B}$ to `C`

- Then

```
div_prop: LEMMA ∀ (x, y: real): x ≠ y ⊃ (x - y)/(y - x) = -1
```

- Generates the following proof obligation

```
TCC1: OBLIGATION ∀ (x, y: real): x ≠ y ⊃ (y - x) ≠ 0
```

Such TCCs are discharged automatically by decision procedures

The subp "Challenge"

- Consider the function *subp* on the integers defined by

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

- This function is undefined if $j > i$
  - When $j \leq i$, $\textit{subp}(i,j) = i - j$
- Often cited as demonstrating the need for partial functions
- But is easily handled using dependent predicate subtypes

```
subp((i:int), (j:int | j ≤ i)): RECURSIVE int =
  IF i = j THEN 0 ELSE subp(i, j+1) + 1 ENDIF
MEASURE i-j

subp_val: LEMMA ∀ (i,j: int): j≤i ⊃ subp(i,j) = i-j
```

The subp "Challenge" (continued)

Specification generates the following TCCs

```
% Subtype TCC generated (line 5) for  j + 1
  subp_TCC2: OBLIGATION
    FORALL (i: int, j: int | j <= i): NOT i = j IMPLIES j + 1 <= i

% Termination TCC generated (line 5) for  subp
  subp_TCC3: OBLIGATION
    FORALL (i: int, j: int | j <= i): NOT i = j IMPLIES i - (j + 1) < i - j

% Subtype TCC generated (line 6) for  i - j
  subp_TCC1: OBLIGATION FORALL (i: int, j: int | j <= i): i - j >= 0
```

All three are proved automatically

Example: Predicate Subtypes (Higher-Order)

- It often contributes clarity and precision to a specification if functions are identified as injections, surjections, etc.
- But how to ensure purported surjection really has the property?
- Predicate subtypes! The surjections are a subtype of the functions associated with the following predicate

```
function_props[dom, rng: TYPE]: THEORY
  surjective?(f): bool = ∀ (r: rng): ∃ (d: dom): f(d) = r
```

So that

```
half_alt: (surjective?[even_nat, nat]) = λ (e: even_nat): e/2
```

Generates the proof obligation

```
TCC2: OBLIGATION ∀ (n: nat): ∃ (e: even_nat): half_alt(e) = n
```

Example: Predicate Subtypes (Invariants)

- The specification of the transition function for a state machine indicates how it computes the next state from the current state and input

- Can characterize the "good" states of the state machine by a predicate (e.g., `not stack_overflow(s)`)

- If the transition function is specified to have type `[good_state, input -> good_state]`

- Then typechecking will automatically generate all the proof obligations necessary to ensure transition relation is closed on the `good_states`

Tables

- Once undecidable typechecking is accepted, it is natural to contemplate additional specification constructs that require theorem proving to check their acceptability—such as tables

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

- Tables are convenient for piecewise continuous functions that must be specified by cases
  - Makes the cases explicit, allowing checks that they do not overlap, and none are overlooked
  - Provides visually attractive presentation that eases comprehension
- First is a semantic issues handled in PVS by `COND` construct
- Second is a syntactic issues handled in PVS by `TABLE` construct that builds on `COND`

The PVS `COND` Construct

- The `COND` construct associates expressions e_i with conditions c_i
- Exactly one of the c_i must be true—two TCCs check this
  - *Disjointness* requires that each distinct c_i, c_j pair is disjoint
    Suppressed if e_i, e_j are syntactically identical
  - *Coverage* requires that the disjunction of all the c_i is true

  In which case the `COND` is equivalent to a nested `IF-THEN-ELSE`

COND $c_1 \rightarrow e_1$, $c_2 \rightarrow e_2$, $\ldots$ $c_n \rightarrow e_n$ ENDCOND	IF c_1 THEN e_1 ELSIF c_2 THEN e_2 $\ldots$ ELSE e_n ENDIF

- Can use `ELSE` for c_n, in which case no coverage TCC is needed

The PVS COND Construct: Example

```
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
```

Generates the following TCCs (proved automatically)

```
% Disjointness TCC (line 10)
  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 (line 10)
  sign_cond_TCC3: OBLIGATION FORALL (x: int): x < 0 OR x = 0 OR x > 0
```

Tables in PVS

- One-dimensional tables are just syntactic variations on COND
- Two forms: vertical and horizontal

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

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

Note: the %----------% lines are just comments

- Both forms generate same TCCs as the COND version and are treated internally the same as COND
- But prettyprinter remembers form used; prints tables as tables
- No extensions required to prover, so can easily establish

```
equiv: LEMMA sign_vtable = sign_cond AND sign_vtable = sign_htable
```

PVS Tables: Extensions

- Blank entries: allow explicit indication of "impossible" cases
  - Coverage TCC modified in this case to ensure blank entries cannot be encountered
  - Requires coverage TCC even when `ELSE` case present
- Enumeration tables: used when conditions all of form $\mathtt{x} = \mathtt{v}_i$
  - Visually more attractive
  - Translates to `CASES` rather than `IF-THEN-ELSE` if $\mathtt{v}_i$ are constructors of some datatype
    (e.g., values of an enumerated type)
  - Typechecker checks the disjointness TCC internally in cases where it is trivially decidable (to avoid having to construct external representation of TCC with n^2 conjuncts)
- Two dimensional tables: treated internally as a pair of nested one-dimensional tables

Example Application: Space Shuttle HAC

- Critical descent maneuver is initial bank onto the "Heading Alignment Cylinder" (HAC)
- Space Shuttle flight software development process is generally considered exemplary (CMM Level 5)
- Requirements for modifications are documented in Change Requests (CRs)
- Subjected to extensive review by Requirements Analysts
- CR (20 pages) for improved display of HAC flight information to Shuttle pilots documents key portions of the required control logic in tabular form
- These were formalized using PVS tables by DiVito and Roberts

Shuttle HAC: Small Example Table

```
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)} #]
```

Continued...

```
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
```

Shuttle HAC: Issues Discovered Through Formal Methods

- Attempts to prove TCCs revealed need to document implicit "domain knowledge," such as
  - `Major_mode = 305` or `603` implies `IPHASE` ≤ 3
  - `WOWLON` can be true only if `Major_Mode = 305` or `603`
- Domain knowledge iteratively incorporated into the specification using dependent predicate subtyping
  - Allows automated proof
- This explicit formulation of domain knowledge was considered valuable in its own right
- Conditions for several rows in one table still overlapped
- Led to identification of missing conjunct

Another Example Application: SRT Division

- The much publicized Pentium FDIV bug cost Intel $475 million
- Bug was in quotient-prediction table for radix-4 SRT divider
- SRT division is now something of a test-piece for formal verification, and has been done by many groups
- PVS treatment (by Rueß, Srivas, and Shankar) specifies and verifies the general theory of SRT division, and its implementation in terms of a datapath and lookup table
- Derives the properties required of lookup table
- And verifies specific table similar to Pentium
- Subsequent work by Miner and Leathrum connected the treatment with IEEE floating point representations, and derived several different tables in under three hours each

SRT Division in PVS

- Rueß, Srivas, and Shankar verified a particular quotient lookup table, similar to the Pentium's, due to Taylor (1981)
- Table has blank entries, believed inaccessible
  - The FDIV bug was here
- PVS specification uses the `TABLE` construct to generate proof obligations (TCCs) for blank entries automatically
- Must also verify that nonblank entries have the right values
- Proofs largely automatic
  (Manual case-split needed to deal a with nonlinearity)

Quotient Lookup Table in PVS

```
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
```

SRT Division: Counterexample in PVS

- The PVS `table` construct automatically generates the TCCs to show that the blank areas in the table are unreachable
- These 23 TCCs are proved by same strategy as correctness theorem
- 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

```
 P!1(6) = 1, P!1(5) = 1, P!1(4) = 0,
 P!1(3) = 0, P!1(2) = 0
 estimation_bound?(...)
 D!1(2) = 0, D!1(1) = 1, D!1(0) = 0
|-------
 lookup_legitimate?(...)
```

Counterexample: `D = 010`, `P = 11000`

PVS Tables: Integration with Other Capabilities

Combined with other capabilities of PVS, the `TABLE` construct is able to provide visually attractive and semantically faithful representations for several tabular specification methods

- Decision tables used by Honeywell (patented by Sherry) for specifying avionics requirements
  - Mechanized by Hoover and Chen's `TableWise`
- And/Or tables used in Heimdahl and Leveson's RSML
  - Mechanized by Heimdahl's RSML checker
- Event tables used in the Naval Research Laboratory's Software Cost Reduction (SCR) method
  - Mechanized in NRL's SCR* tool

Example: SCR Tables and Model Checking

Specification for an automobile cruise control

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

Cruise Control Example in PVS

```
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
   |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 ||
...
```

Typechecking generates several unprovable disjointness TCCs that pinpoint problems in the specification

Cruise Control Example in PVS (continued)

- After correcting the errors revealed by typechecking, can proceed to deeper examination of the specification by checking that expected properties hold
- The table specifies a finite state transition relation
- Some expected properties can be expressed in CTL, such as

```
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)
```

- Can examine these properties using PVS's integration of theorem proving and model checking
- Reveals that `safe5` is true, but `safe4` is not
- No changes needed to the theorem prover or model checker

Conclusions

- Compared with lightweight, special-purpose tools, PVS's mechanization of tabular specifications
  - Required little effort to develop, but is more powerful
    - Can discharge proof obligations that require general theorem proving (e.g., arithmetic relations)
    - Can specify and prove arbitrary properties about tabular specifications, using theorem proving and model checking
  - And more general: tables available for use in traditional specifications; users can develop customizations
  - But is less smooth
  - Scales less well syntactically, better semantically
- Currently working with Heimdahl and Skakkebæk to analyze TCAS II specification (RSML to PVS to SVC);
  May have found some real issues in the specification
- Illustrates value of integration: whole is more than sum of parts

To Learn More About PVS

- Browse papers and technical reports at
  `http://www.csl.sri.com/fm.html`
  - `tse95.html` is a good overview
  - `pvs-bib.html` links to a bibliography with over 140 entries
  - Links to Stanford papers and to Murϕ there also
- Information about PVS is available from
  `http://www.csl.sri.com/pvs.html`
  - `http://www.csl.sri.com/pvs/examples` gets you to a directory of tutorials and their supporting specification files
- You can get PVS from `ftp://ftp.csl.sri.com/pub/pvs`
  - Allegro Lisp for SunOS 4.1.X or Solaris, IBM AIX, or Linux
  - Need 64M memory, 100M swap space, Sparc 20 or better (Best is 200 MHz P6, costs less than $3,000 in USA)