Software Verification Example: Using Tables in PVS to Analyze Power Conditioning Requirements

Dr. Mark Lawford
Assistant Professor
Dept. of Computing And Software
Faculty of Engineering
McMaster University

lawford@mcmaster.ca

Outline

- CAV: Computer Aided Verification
- Power Conditioning Overview
- PVS Specification
- Using PVS to debug the specification

Computer Aided Verification

What is CAV? . . . Prove, prove, prove!

Use tools to mathematically "prove" a design implements a well defined specification. E.g.

- Automated theorem proving of functional equivalence (i.e. use PVS or IMPS to prove for all inputs x: Spec(x) = Design(x))
- Model-checking automatically verifies that a Design is a model of a Spec written as a logical formula

Why use CAV Tools?

- Independent check of system unaffected by verifier's expectations
- Domain coverage - Tools can often be used to check ALL input cases
- Tools let you automate verification and reverification
- Provide additional capabilities (e.g. generation of counter example for debugging, type checking, verifying whole classes of systems, etc.)

Example:
Reactor Shutdown System (SDS)

What is an SDS?

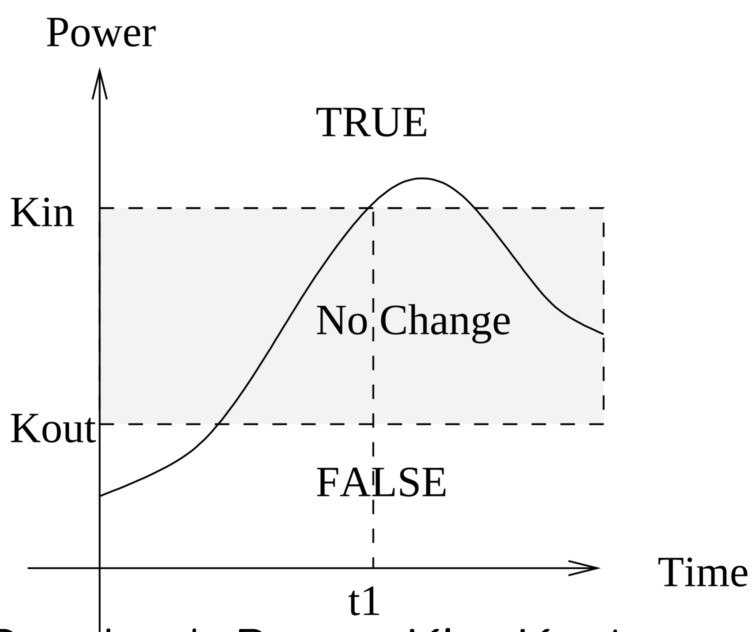
- watchdog system that monitors system parameters
- shuts down (trips) reactor if it observes "bad" behavior
- process control is performed a separate Digital Control computer (DCC) - not as critical

Consider simple subsystem: Power Conditioning

- Many sensors have a Power threshold below (or above) which readings are unreliable so it's "conditioned out" for certain Power levels.
- A deadband is used to eliminate sensor "chatter"

Idea: Use code reuse - write one general routine and pass in sensor parameters for different sensors

General Power Conditioning Function



PwrCond(Prev:bool, Power, Kin, Kout:posreal):bool =

$Power \leq Kout$	$Kout < Power < Kin$	$Power \geq Kin$
$FALSE$	$Prev$	$TRUE$

PVS (Prototype Verification System), a "proof assistant" can automatically check for completeness (coverage) and determinism (disjointness) of tables.

i.e. PVS checks that a table defines a total function.

When Power:

- drops below $Kout$, sensor is unreliable so it's "conditioned out" $(PwrCond = FALSE)$.
- exceeds Kin, the sensor is "conditioned in" and is used to evaluate the system.
- is between $Kout$ and Kin, the value of $PwrCond$ is left unchanged by setting it to its previous value, $Prev$.

E.g. For the graph of $Power$ above, $PwrCond$ would start out FALSE, then become TRUE at time $t1$ and remain TRUE.

PVS Specification of a General $PwrCond$ Function

```
  PwrCond(Prev:bool, Power, Kin, Kout:posreal):bool = TABLE
    %-----------------------------------------------------%
    |[Power<=Kout | Power>Kout & Power<Kin | Power>=Kin]|
    %-----------------------------------------------------%
    |   FALSE      |         Prev             |   TRUE   ||
    %-----------------------------------------------------%
  ENDTABLE
```

The above PVS specification of the $PwrCond$ table produces the following proof obligations or "TCCs".

```
% Disjointness TCC generated (at line 14, column 55) for
  % unfinished
PwrCond_TCC1: OBLIGATION
  FORALL (Kin, Kout: posreal, Power):
    NOT (Power <= Kout AND Power > Kout & Power < Kin) AND
     NOT (Power <= Kout AND Power >= Kin) AND
      NOT ((Power > Kout & Power < Kin) AND Power >= Kin);

% Coverage TCC generated (at line 14, column 55) for
  % proved - complete
PwrCond_TCC2: OBLIGATION
  FORALL (Kin, Kout: posreal, Power):
    (Power <= Kout OR                 % Column1
     (Power > Kout & Power < Kin)  % Column2
     OR Power >= Kin)                % Column3
```

Type-checking PwrCond

The coverage TCC is easily proved by PVS. Thus we conclude that at least one column is always satisfied for every input.

But attempting the Disjointness TCC fails, indicating that the columns overlap. The resulting unprovable sequent for the disjointness TCC is:

```
PwrCond_TCC1 :

[-1]    Kin!1 > 0
[-2]    Kout!1 > 0
[-3]    Power!1 > 0
[-4]    Power!1 <= Kout!1
[-5]    (Kin!1 <= Power!1)
  |-------
[1]     FALSE
Rule?
```

Step 1: Characteristic Equation

Writing down the characteristic formula for the unprovable sequent.

$$\begin{gathered} Kin_1 > 0 \wedge Kout_1 > 0 \wedge Power_1 > 0 \wedge \\ Power_1 \leq Kout_1 \wedge Kin_1 \leq Power_1 \rightarrow \bot \end{gathered}$$

which is equivalent to:

$$\begin{aligned} &\neg(Power > 0 \wedge Kin > 0 \wedge Kout > 0 \wedge \\ &Power \leq Kout \wedge Kin \leq Power) \end{aligned}$$

We know that an interpretation structure (i.e., a program) will satisfy this formula iff it satisfies the formula's universal closure:

$$\begin{aligned} &(\forall Power, Kin, Kout : posreal) \\ &\neg(Power > 0 \wedge Kin > 0 \wedge Kout > 0 \wedge \\ &Power \leq Kout \wedge Kin \leq Power) \end{aligned} \tag{1}$$

Step 2: Find a Counter Example

Find a counter example that makes the characteristic formula false.

NOTE: The counter examples values for Kin, $Kout$ and $Power$ must be of the type

$$posreal = \{x : real | x > 0\}$$

So, to make (1) false, we make $\neg$(1), if equivalently the following true:

$$\begin{array}{l}(\exists Power, Kin, Kout : posreal)\\ (Power > 0 \land Kin > 0 \land Kout > 0 \land\\ Power \leq Kout \land Kin \leq Power)\end{array}$$

With a slight abuse of notation this simplifies to:

$$\begin{array}{l}(\exists Power, Kin, Kout : posreal)\\ (0 < Kin \leq Power \leq Kout)\end{array}$$

e.g Take $Kin_1 = 1, Kout_1 = 3, Power_1 = 2$

Step 3: Verify Counter Example

Verify that the counter example satisfies the conditions of two or more columns of the table for PwrCond.

In the case when $Kin = 1, Kout = 3, Power = 2$ we have the condition on column 1 and condition 3 satisfied since:

i) $Power \leq Kout$ because $2 \leq 3$, and

ii) $Power \geq Kin$ because $1 \leq 2$.

Step 4: Find the Error Source

What implicit assumption did the designers make regarding input arguments Kin and Kout that led them to omit the counter example case from the table?

They assumed that $Kout < Kin$.

Why is such an undocumented assumption dangerous in a setting where code may be reused by other developers?

When someone other than the code developer reuses the code, they may not know about any implicit assumption and may use the code in a way that it was not intended for.

E.g. Suppose a sensor was only valid at low values of $Power$. Someone may think that they could just use function PwrCond with $Kout > Kin$. In this case it is not specified what the function will do!

Step 5: Correct the Error

Problem: Determinism check fails when

$$Kin \leq Kout$$

Why? Implicit (undocumented) assumption from diagram that $Kin > Kout$

Fix: Make assumption explicit.

How? Use dependent typing to create a new version of the PwrCond table that makes the assumed relation between Kin and Kout explicit and thereby rules out any counter examples like those above.

```
PwrCond(Prev:bool, Power, Kin:posreal,
        Kout:{x:posreal| x<Kin}):bool = TABLE
  %---------------------------------------------------%
  |[Power<=Kout | Power>Kout & Power<Kin | Power>=Kin]|
  %---------------------------------------------------%
  |   FALSE     |         Prev           |   TRUE   ||
  %---------------------------------------------------%
ENDTABLE
```