Specification using PVS of a Majority Voting Device

JSO, Software Engineering Laboratory, York University

September 4, 2015

In many safety critical systems, majority voting is used to improve the reliability of the systems to hazardous conditions. For example, in nuclear reactors, if high pressures and temperatures are detected, then we may need to rapidly shutdown the system. Often there will be three independent shutdown systems with a majority vote taken when to act.

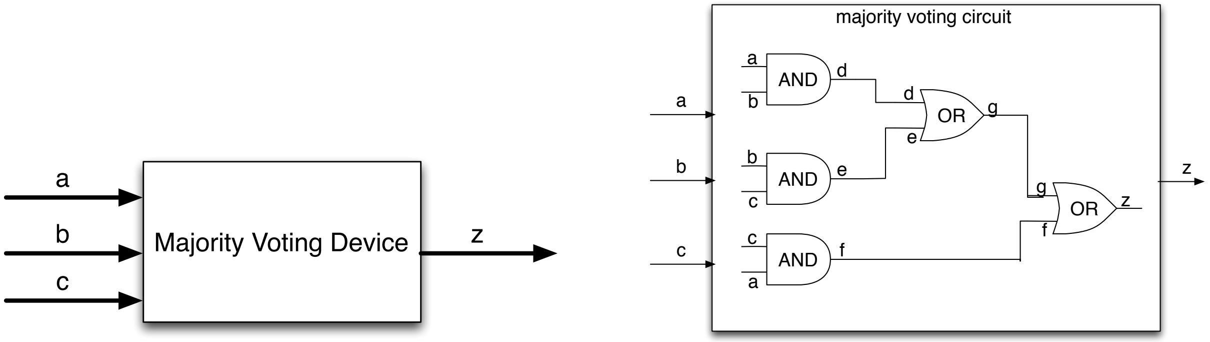


Figure 1: Majority Voting Specification (left) and Implementation (right)

Assuming all the monitored and controlled variables are of type BOOLEAN, the *specification* of the majority voting circuit is:

$$\begin{aligned} \mathit{spec}\,(a,b,c,z:\mathbb{B}):\mathbb{B} &\;\widehat{=}\; z \equiv (\mathit{cnf}(a)+\mathit{cnf}(b)+\mathit{cnf}(c) >= 2) \\ \mathit{cnf}(x:\mathbb{B}):\mathbb{Z} &\;\widehat{=}\; (\text{IF } x \text{ THEN } 1 \text{ ELSE } 0) \end{aligned} \tag{1}$$

Aside: The importance of well-definedness and type correctness

Notice that the above definition of the conversion function *cnf* declares that *cnf* is a function with type $[\mathbb{B} \rightarrow \mathbb{Z}]$, i.e. *cnf* takes an argument of type boolean $\mathbb{B}$ and returns a result of type integer $\mathbb{Z}$. The type of the function imposes a proof obligation on the user of the function to use it only where it is a well-defined, i.e. there will be an obligation to prove a type correctness condition. For example, suppose in PVS we have:

```
f(i:nat): bool = (if i <= 10 then true else false endif)
g: bool = f(9)
```

All is well because $f(9)$ is well-defined, i.e. $9 \in \mathbb{N}$. However, changing the definition of g to $g : \mathbb{B} = f(-1)$, generates a type correctness condition:

```
Subtype TCC generated (at line 12, column 13) for  -1
  expected type  nat
  unfinished
g_TCC1: OBLIGATION -1 >= 0;
```

The above type condition $(-1 \geq 0)$ cannot be proved (i.e. $-1 \notin \mathbb{N}$) indicating that $f(-1)$ is not well-defined. If we implemented this model in code, exceptions or segmentation errors will happen, which is toxic for mission critical systems.

Specification

On the left hand side of Fig. 1, we show the system under description (SUD) as a black box. Also shown, are the *monitored* variables a, b and c at the input, and the *controlled* variable z at the output. Formula (1) provides the specification relating output to input.

Note that we have specified the majority voting circuit using a *relational style* rather than the *functional style* we used (earlier) for the NAND gate.[1] Relational specifications will provide us with more flexibility in describing systems at the requirements level. A PVS theory for the specification is shown on the next page.

[1] A functional specification of an inverter gate is: $inverter(x : \mathbb{B}) \ \widehat{=}\ \neg x$. A relational specification is: $inverter(x, z : \mathbb{B}) \ \widehat{=}\ z \equiv \neg x$.

```
majority_vote: THEORY
BEGIN
  % input and output variables
  a,b,c,z,d: VAR bool
  % conversion function
  cnf(x:bool): int = (if x then 1 else 0 endif)

% specify the required behaviour
  spec(a,b,c,z): bool = (z = (cnf(a) + cnf(b) +cnf(c) >= 2))

% the result of cnf is either 0 or 1
  sanity_check_1: CONJECTURE  (FORALL d: cnf(d) = 0 OR cnf(d) = 1)
...
```

What you must do: Prove the sanity check using only flatten, split, prop, assert, expand, iff, lift-if, skolem (or skeep) and inst (and variants).

The alternative (IF ... ELSE ..) can be expanded using "lift-if" as follows:

```
sanity_check_1 :

  |-------
[1]   (IF d THEN 1 ELSE 0 ENDIF) = 0
[2]   (IF d THEN 1 ELSE 0 ENDIF) = 1

Rule? (lift-if)
Lifting IF-conditions to the top level,
this simplifies to:
sanity_check_1 :

  |-------
{1}   IF d THEN 1 = 0 ELSE TRUE ENDIF
{2}   IF d THEN TRUE ELSE 0 = 1 ENDIF
```

Implementation

Fig. 1 (on the right) proposes an implementation that will satisfy the specification of the majority voting circuit. We would like to check that the implementation indeed satisfies the specification. Here is some PVS specification to get you started:

```
% define and_gate
  and_gate(v,w,x: bool): bool =
        x IFF (v AND w)
% define or_gate
  or_gate(v,w,x:bool): bool =
        x IFF (v OR w)

% describe the implementation in terms of 'and' and 'or' gates
  implementation(a,b,c,z): bool =
    ???

% Hint: use the existential quantifier
% to hide the internal connections

% the result of cnf is either 0 or 1
  sanity_check_1: THEOREM
        (FORALL (d: bool): cnf(d) = 0 OR cnf(d) = 1)

  implementation_correctness: THEOREM
        implementation(a,b,c,z) => spec(a,b,c,z)

END majority_vote
```

What you must do: Create a new pvs file `majority_vote.pvs` (with theory `majority_vote`). Complete the description of the implementation using the *and* and *or* gates. Prove the the sanity and correctness conjecture using only flatten, split, prop, case, assert, expand, iff, lift-if, skolem and inst (and variants). You may **not** use *grind* (which will automatically prove the conjecture).

Thoughts on material to come

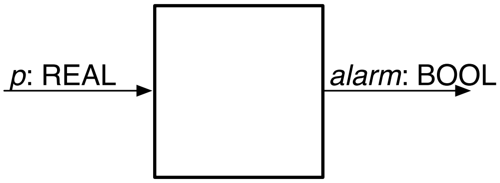
Note that the PVS command `(grind)` will automatically prove the conjectures.[2] But at this point we would like you to practice the use of the individual proof rules. "Grind"

[2] The property `implementation(a,b,c,z) = spec(a,b,c,z)` will not blow away with *grind*. You can try this as a challenge question.

will not always succeed, and you must then know how to guide PVS manually.

Also, the above example can be solved just by completing a truth table. It's a bit tedious but manageable. There are many automated tools for solving problems with digital circuits using just 1 and 0. Not so common are tools that will deal with continuos real-valued variables or floating point arithmetic.

Consider a device with a monitored variable pressure $p : \mathbb{R}$ and controlled variable *alarm* as the output as shown in Fig. 2.



	$alarm$
$p \leq normal$	$False$
$normal < p < hi$	No Change
$p \geq hi$	$True$

Figure 2: Input pressure $p : \mathbb{R}$ and output $alarm : \mathbb{B}$

On the left is a function table that specifies the output in terms of the input and a real-valued constants *normal* and *hi*. The function table specifies when the alarm will go high, i.e. generate an alarm signal. "No Change" means that *alarm* stays the same as in the previous state.

In validating such specifications we now have to deal with an infinite number of possible inputs (some slice of the real line representing pressures).

Furthermore, how would we specify "No Change" from the previous time an input was received?

We will be learning how to do that in subsequent work.