FH-OÖ Hagenberg/ESD
Advanced Methods of Verification, SS 2015
Rainer Findenig © 2008

10. Übung: PSL

1 LTL vs. CTL vs. PSL

- LTL und CTL haben unterschiedliche Mächtigkeiten, es können also mit jeder der beiden Logiken Ausdrücke formuliert werden die in der jeweils anderen nicht möglich sind. Geben Sie ein Beispiel für eine CTL-Formel an, die in LTL nicht darstellbar ist, und erklären Sie dieses Beispiel[1]!
- Gegeben seien nun die beiden Formeln

  $$\varphi_{\text{LTL}} = \mathbf{G}(r) \rightarrow \mathbf{F}(\neg a \wedge \neg b)$$

  und

  $$\varphi_{\text{CTL}} = \mathbf{AG}(r) \rightarrow \mathbf{AF}(\neg a \wedge \neg b).$$

  Geben Sie an, welche der beiden Formeln für die Zähler-Kripke-Struktur aus der Vorlesung gilt! Sind beide Formeln sinnvoll?
- Welche Änderung gegenüber LTL und CTL verhilft PSL zu einer nochmals anderen Mächtigkeit? Geben Sie ein Beispiel für einen Ausdruck an, der in PSL, nicht aber in LTL und/oder CTL darstellbar ist!

2 PSL in Action

Erstellen Sie PSL-Assertions für die folgende Bus-Spezifikation. Schreiben Sie zusätzlich eine Testbench, um Ihre Assertions zu überprüfen.

- Der Bus besteht aus den Signalen `clk`, `request`, `grant`, `data_valid`, `done` und `abort_`.
- Wenn der Master den Bus anfordert (`request = '1'`), dann muss der Arbiter den Bus innerhalb der nächsten fünf Zyklen bereitstellen (`grant = '1'`). Sollte dies nicht möglich sein, muss der Arbiter dies dem Master mitteilen, indem er innerhalb der gleichen Zeit das Signal `abort_` setzt (Abbildung 2).
- Im ersten Zyklus, nachdem der Master den Buszugriff erhält (`grant = '1'`), muss der Master beginnen, Daten zu übertragen. Der Master kann 8, 16 oder 32 Zyklen

[1] Der umgekehrte Weg ist schwieriger zu beweisen, ein Beispiel sei hier aber gegeben: Die Formel $\mathbf{FG}p$ hat kein Pendant in CTL.

lang Daten übertragen (während dieser Zeit ist `data_valid` high). Im ersten Zyklus nach dem Transfer muss der Master durch Setzen des Signals `done` den Bus wieder freigeben (Abbildung 1).

- Nachdem der Master den Bus angefordert hat, darf `grant` nur genau einen Zyklus lang high sein, bis der Buszugriff beendet ist.
- Nachdem der Master den Bus angefordert hat, darf er ihn nicht nochmals anfordern, bis der Transfer entweder ordnungsgemäß beendet oder abgebrochen wurde.

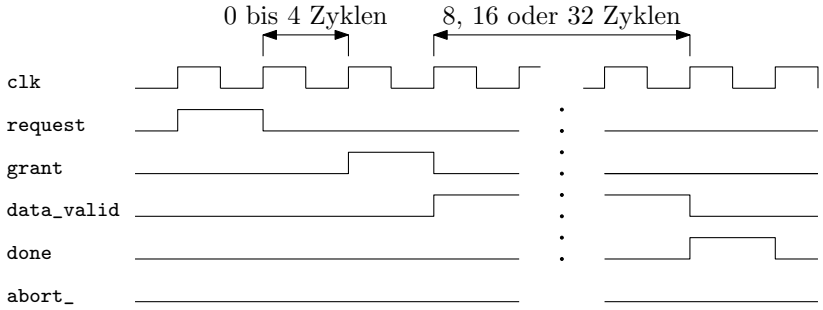

Abbildung 1: Erfolgreicher Transfer.

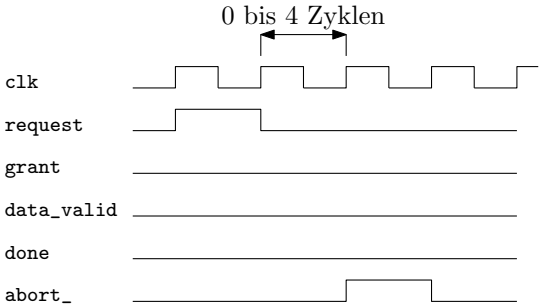

Abbildung 2: Arbiter bricht Transfer ab.

"A formal manipulator in mathematics often experiences the discomforting feeling that his pencil surpasses him in intelligence."

Howard W. Eves

1 LTL vs. CTL vs. PSL

- Die Formel

$$\varphi_{CTL} = \mathbf{EG}p$$

  ist nur in CTL möglich, da sie eine Aussage über die Existenz eines Pfades ist, was in LTL nicht möglich ist.

- Die LTL-Formel entspricht dem Reset und gilt für die Zähler-Kripke-Struktur aus der Vorlesung. Die CTL-Formel gilt nicht und ist sinnlos.

- Die Erweiterung mit regulären Ausdrücken verhilft PSL zu einer höheren Mächtigkeit.

  ```
  assert always (a -> {next_a}[3:5](b));
  ```

  Wie man in diesem Beispiel sieht kann die PSL auch zählen, was in CTL und LTL nicht möglich ist.

2 PSL in Action

../../src/SimpleBus–Bhv–ea.vhd

```
1  library IEEE;
2  use IEEE.numeric_std.ALL;
3  use IEEE.STD_LOGIC_1164.ALL;
4
5  entity SimpleBus is
6  end entity;
7
8  architecture Bhv of SimpleBus is
9    signal clk, request, grant, data_valid, done, aborting : std_ulogic :=
       '0';
10 begin
11   clk <= not clk after 10 ns;
12   request <= '0' after 0 ns,
13                '1' after 20 ns,
14                '0' after 40 ns,
15                '1' after 280 ns,
16                '0' after 300 ns;
17   grant <= '1' after 60 ns,
18            '0' after 80 ns;
19   data_valid <= '1' after 80 ns,
20                 '0' after 240 ns;
21   done <= '1' after 240 ns,
22           '0' after 260 ns;
23   aborting <= '1' after 300 ns,
24               '0' after 320 ns;
25
26
27 end Bhv;
```

../../src/vunit.psl

```
1  vunit vSimpleBus(SimpleBus(Bhv)) {
2    default clock is (clk'event and clk = '1');
3
4    property req is always (rose(request) -> {[*0 to 4]; (grant or aborting);
       not grant});
```

```
5    assert req;
6
7    property data_valid_prop is always ({rose(grant)} |=> {data_valid[*8] |
        data_valid[*16] | data_valid[*32]; not data_valid});
8    assert data_valid_prop;
9
10   property isdone is always (fell(data_valid) -> done);
11   assert isdone;
12
13   property not_req is always ({request} |=> not request until (done or
        aborting));
14   assert not_req;
15 }
```

../../sim/ComSim.do

```
1  vlib work
2
3  vcom ../src/SimpleBus-Bhv-ea.vhd -pslfile ../src/vunit.psl
4
5  vsim -novopt SimpleBus
6
7  add wave -position end  sim:/simplebus/clk
8  add wave -position end  sim:/simplebus/request
9  add wave -position end  sim:/simplebus/grant
10 add wave -position end  sim:/simplebus/data_valid
11 add wave -position end  sim:/simplebus/done
12 add wave -position end  sim:/simplebus/aborting
13
14 run 500 ns
```