# Open Source Formal Verification

## Linear Temporal Logic - LTL

Yann Thoma

Reconfigurable and Embedded Digital Systems Institute
Haute Ecole d'Ingénierie et de Gestion du Canton de Vaud

This work is licensed under a Creative Commons Attribution-NonCommercial-ShareAlike 3.0 Unported License

July 2025

1. Verification engineer mindset
2. State machine verification
3. Conclusion

# LTL: Introduction

- Linear Temporal Logic
  - Allows to express temporal relationships between signals
  - Basic operations:

| | | |
|---|---|---|
| $\phi ::=$ | $\top$ | true |
| | $\bot$ | false |
| | $\neg(\phi)$ | negation |
| | $p$ | proposition |
| | $(\phi \wedge \phi)$ | conjunction |
| | $(\phi \vee \phi)$ | disjunction |
| | $X\phi$ | next time $\phi$ |
| | $F\phi$ | eventually $\phi$ (strong operator) |
| | $G\phi$ | always $\phi$ |
| | $\phi U \phi$ | $\phi U \psi$ : $\phi$ until $\psi$ (strong operator) |

# LTL: Introduction

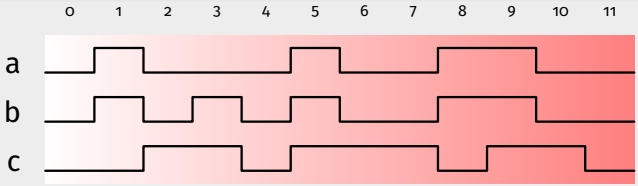
- PSL offers various operators to ease writing temporal properties
  - Presented in the following slides
- *Safety* properties
  - To check the correctness of the system
- *Liveness* properties
  - To check that something eventually happen
  - Not supported by the current tool

# `next` operator and implication (1)

- `next`
  - Advance in time by one clock cycle, so check if its operand is verified on the next clock cycle
- Implication `->`
  - Direct implication, left-hand side in the boolean domain, right-hand side not necessarily
  - Example: `a -> b`
    - If `a` then `b`
  - Example: `a -> next b`
    - If `a` then `b` on the next clock cycle
- `next` *moves in time*, and as such can not be on a left-hand side of an implication
  - Not allowed: `(a and next b) -> c`

# `next` operator and implication (2)

## Example

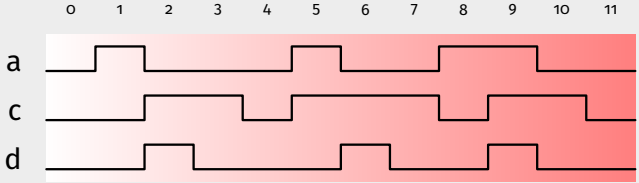



## PSL

```
assert always (a -> b);

assert always (b -> a);

assert always (a -> next c);

assert always (b -> (a or c));
```

# `next` operator and implication (3)

## Example




## PSL

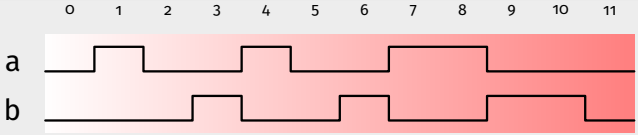
```
assert always (a -> next c);

assert always (a -> next d);
```

# `next` operator: variants (1)

- `next[n](a)`
  - Indicates that the property if verified after *n* clock cycles
  - Example: `a-> next[3](b)`
    - If *a* then *b* after 3 clock cycles
- `next_a[n to m](a)`
  - Indicates that the property is verified every clock cycles from *n* to *m*
  - Warning: The `to` corresponds to VHDL, not PSL.
  - In Verilog it would be [n:m]
  - Example: `a-> next_a[3 to 5](b)`
    - If *a* then *b* must be verified after 3,4 and 5 clock cycles
- `next_e[n to m](a)`
  - Indicates that the property is verified in at least once between *n* and *m* clock cycles from the current one
  - Example: `a-> next_e[3 to 5](b)`
    - If *a* then *b* must be verified at least once after 3, 4 or 5 clock cycles

# `next` operator: variants (2)

## Example

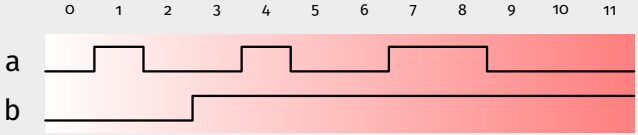



## PSL

```
assert always (a -> next[2](b));

assert always (a -> next_a[2 to 4](b));

assert always (a -> next_e[2 to 4](b));
```

# `next` operator: variants (3)

## Example

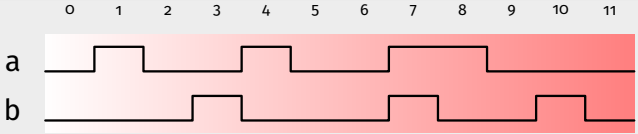



## PSL

```
assert always (a -> next[2](b));

assert always (a -> next_a[2 to 4](b));

assert always (a -> next_e[2 to 4](b));
```

# `next` operator: variants (4)

## Example

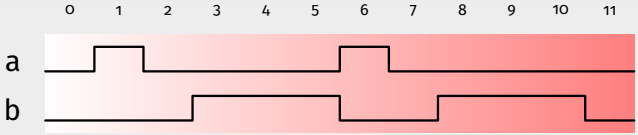



## PSL

```
assert always (a -> next[2](b));

assert always (a -> next_a[2 to 4](b));

assert always (a -> next_e[2 to 4](b));
```

# `next` operator: variants (5)

## Exemple




## PSL

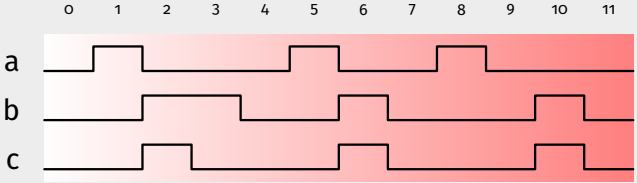
```
assert always (a -> next[2](b));

assert always (a -> next_a[2 to 4](b));

assert always (a -> next_e[2 to 4](b));
```

# `next_event` operator (1)

- `next_event(a)(b)`
  - Verifies that *b* holds the next time *a* does
  - Example: `c-> next_event(a)(b)`
    - If *c* then next time *a* is observed, then *b* must hold
  - Warning: if *a* holds on the same cycle as *c*, `next_event` will trigger
- `next_event(a)[i](b)`
  - Verifies that *b* holds on the $i^{th}$ occurence of *a*
  - Example: `c-> next_event(a)[3] b`
    - If *c* then *b* must be true on the third time *a* is observed

# `next_event` operator (2)

## Example

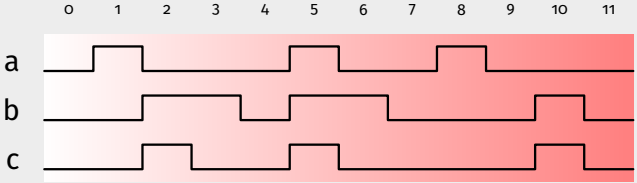



## PSL

```
assert always (a -> next_event(b)(c));

assert always (a -> next next_event(b)(c));
```

# `next_event` operator (3)

## Example

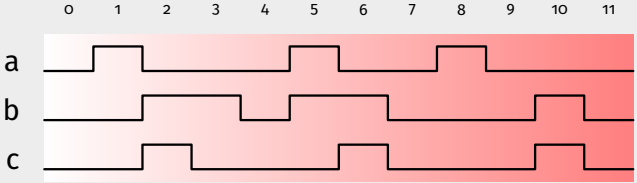



## PSL

```
assert always (a -> next_event(b)(c));

assert always (a -> next next_event(b)(c));
```

# next_event operator (4)

## Example

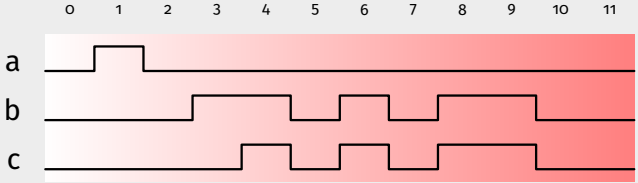



## PSL

```
assert always (a -> next_event(b)(c));

assert always (a -> next next_event(b)(c));
```

# next_event operator (5)

## Example




## PSL

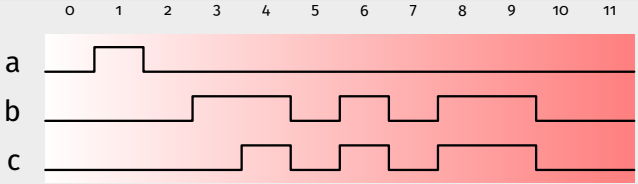
```
assert always (a -> next_event(b)(c));

assert always (a -> next_event(b)[3](c));
```

# `next_event` operator: variants (1)

- `next_event_a(b)[n to m](c)`
  - Indicates that *c* must be verified at occurences *n* to *m* of *b*
  - Example: `a-> next_event_a(b)[3 to 5](c)`
    - If *a* then *c* must be verified at occurences 3,4 and 5 of *b*
- `next_event_e(b)[n to m](c)`
  - Indicates that *c* must be verified on at least one occurence of *b* from *n* to *m*
  - Example: `a-> next_event_e(b)[3 to 5](c)`
    - If *a* then *c* must b verified at least once within occurences 3, 4 or 5 of *b*

# next_event operator: variants (2)

## Example




## PSL

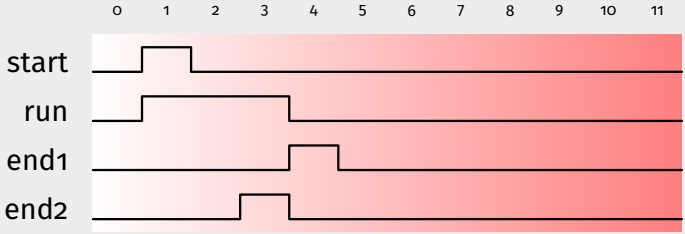
```
assert always (a -> next_event_a(b)[2 to 4](c));

assert always (a -> next_event_a(b)[2 to 5](c));

assert always (a -> next_event_e(b)[1 to 2](c));
```

# `until` operator (1)

- `until`
  - Indicates that a condition must hold until a second one does (does not include the cycle where the second holds)
  - Example: `a until b;`
    - Condition *a* must be verified until *b* is.
- `until_`
  - Indicates that a condition must hold until a second one does, including the cycle the second does
  - Example: `a until_ b;`
    - Condition *a* must be verified until *b* holds, including the cycle where *b* holds

# `until` operator (2)

## Example

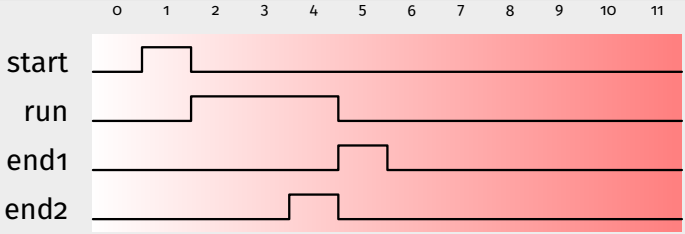



## PSL

```
assert always (start -> run until end1);

assert always (start -> run until_ end1);

assert always (start -> run until_ end2);
```

# `until` operator (3)

## Example

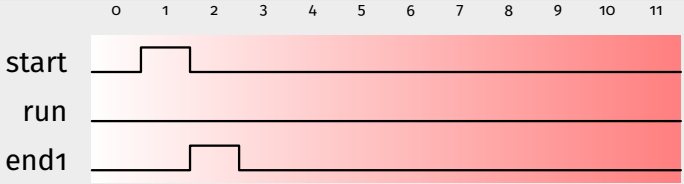



## PSL

```
assert always (start -> next (run until end1));

assert always (start -> next (run until_ end1));

assert always (start -> next (run until_ end2));
```

# `until` operator (4)

## Example




## PSL

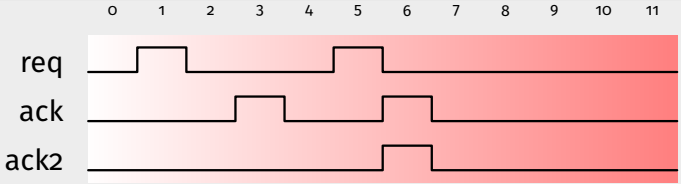
```
assert always (start -> next (run until end1));

assert always (start -> next (run until_ end1));
```

# `before` operator (1)

- `before`
  - Indicates that a condition must be verified stricly before a second one is
  - Exemple: `a before b;`
    - Condition *a* must be verified strictly before *b*
- `before_`
  - Indicates that a condition must be verified before a second one is. Both can hold at the same time.
  - Example: `a before_ b;`
    - Condition *a* must be verified strictly before *b* or at the same time

# `before` operator (2)

## Example

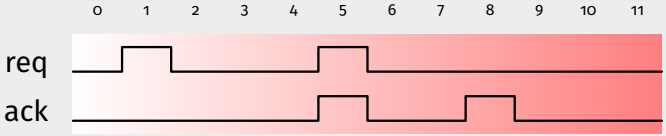



## PSL

```
assert always (req -> next (ack before req));

assert always (req -> next (ack2 before req));
```

# before operator (3)

## Example

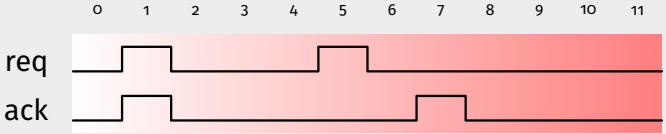



## PSL

```
assert always (req -> next (ack before req));

assert always (req -> next (ack before_ req));
```

# before operator (4)

## Example




## PSL

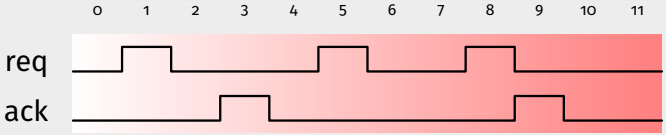
```
assert always (req -> next (ack before req));

assert always (req -> next (ack before_ req));

assert always (req -> (ack or next (ack before req)));
```

# Operator eventually! (1)

- `eventually!`
  - Indicates that a condition must be observed at least once in the future
  - Example: `a -> eventually! b;`
    - Condition `b` must be verified at least once after `a`
- `eventually!` is a strong operator: if `a` occurs, then `b` *must* be observed once

# Operator eventually! (2)

## Example




## PSL

```
assert always (req -> eventually! ack);
```

- Actually we'll have to wait a bit for this operator, as GHDL currently has issues with it.

# Handling the reset

- Usually an `abort rst` is necessary for assertions
- But... It does not verify what happens at reset
- So, we must handle the check of some outputs when the reset is applied

## Synchronous reset

```
assert always (rst -> next counter = 0);
```

## Asynchronous reset

```
assert ((rst = '0') or (counter = 0));
```

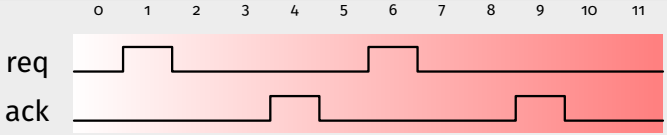
A VHDL assert always checked

- For the asynchronous version, we use the equivalence between ($A \Rightarrow B$ and $\overline{A}$ or $B$)

# How to think about specs?

- In the specs: `req` active is followed by `ack` active 3 clock cycles later

## Example 1




## PSL

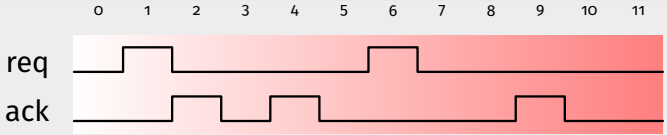
```
assert always (req -> next[3] ack);
```

- Sounds good, no?

# How to think about specs?

- In the specs: `req` active is followed by `ack` active 3 clock cycles later

## Example 2




## PSL

```
assert always (req -> next[3] ack);
```

- Mmh...

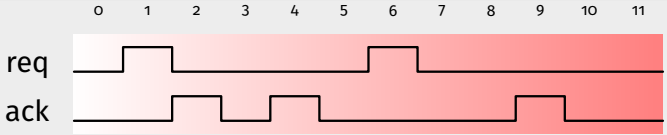
# How to think about specs?

- In the specs: `req` active is followed by `ack` active 3 clock cycles later
- What does it mean exactly?
- It only says what it says: `req -> next[3] ack`
- But `ack` could be `'1'` every clock cycle
- Maybe the specs meant also: If `req` was not active 3 clock cycles ago, the `ack` shall not be active
- Was it a mistake in the specs? Most probably
- So... the information about a certain signal being active at a certain time can hide the fact that it shall not be active any other time
- ⚠ That's a big danger for formal verification
- Mathematically speaking:
- $A \Rightarrow B$ only says that $A$ implies $B$, but it does not say $\overline{A} \Rightarrow \overline{B}$

# How to think about specs?

- In our case, we can add a new assertion: `ack -> prev(req, 3)`

## Example 2



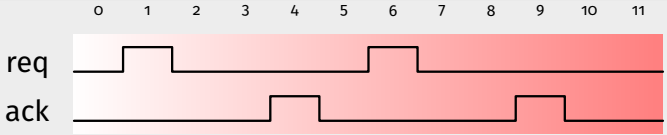

## PSL

```
assert always (req -> next[3] ack);
assert always (ack -> prev(req, 3));
```

# How to think about specs?

- In our case, we can add a new assertion: `ack -> prev(req, 3)`

## Example 1




## PSL

```
assert always (req -> next[3] ack);
assert always (ack -> prev(req, 3));
```

# State machine verification

- Let's consider a finite state machine with potentially some internal counters
- How to verify its behavior?
  - Are there assumptions on the inputs?
  - How to express complex output behaviors?
  - Should we test the internal architecture?

# State machine - assumptions

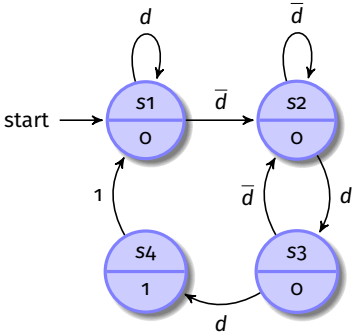
- Usually a state machine interacts with other parts of the system
- Are there really assumptions on the inputs?
  - The fewer assumptions the more reliable the FSM
  - Really depends on the system
  - Be careful not to restrict too much the inputs
- Helper code can be used to model part of the outside world

# State machine - output behaviors

- Not always easy to test the outputs
- Using sequences can be an option, in comparison with LTL
- **Start to write what you do expect and then implement the corresponding assertions** (and maybe assumptions)
- Do not forget things like:
  - `ack` shall be `'1'` 3 cycles after `req`
  - Does it mean it shall be `'0'` any other time?
- FSM often have such kind of inputs

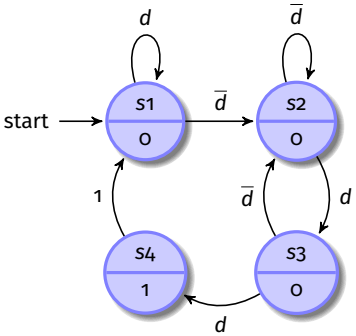
# State machine - transitions

- Example: A sequence detector (011)
- The output shall go high when the sequence is detected (Moore machine)



# State machine - transitions

- If we have access to the internal architecture
- Detect invalid transitions

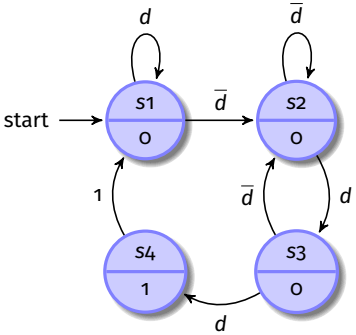



## Check allowed transitions

```
assert always (
    ((state = S1) -> next((state = S1) or (state = S2)))
    abort rst_i);
assert always (
    ((state = S2) -> next((state = S2) or (state = S3)))
    abort rst_i);
assert always
    (((state = S3) -> next((state = S2) or (state = S4)))
    abort rst_i);
assert always (
    ((state = S4) -> next(state = S1))
    abort rst_i);
```

# State machine - transitions

- If we have access to the internal architecture
- Detect transitions with the help of the inputs (often too complex)




## Check allowed transitions

```
assert always (
    (((state = S1) and d) -> next(state = S1))
    abort rst_i);
assert always (
    (((state = S1) and not d) -> next(state = S2))
    abort rst_i);
assert always (
    (((state = S2) and d) -> next(state = S3))
    abort rst_i);
assert always (
    (((state = S2) and not d) -> next(state = S2))
    abort rst_i);
...
```

# Key takeaway

**Check the expected behavior**

- In our example: `ack` must be high 3 cycles after `req`

**Do not forget to check the unwritten behavior**

- In our example: `ack` should be low every other clock cycle