Open Source Formal Verification

Sequential Extended Regular Expressions

Yann Thoma

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

July 2025

Sequential Extended Regular Expressions

- PSL offers SEREs
- A powerful way to express complex temporal behaviors
- Example:
  - {a;b[4];c}
  - a is active, followed by b active 4 cycles, and then c

Suffix Implication

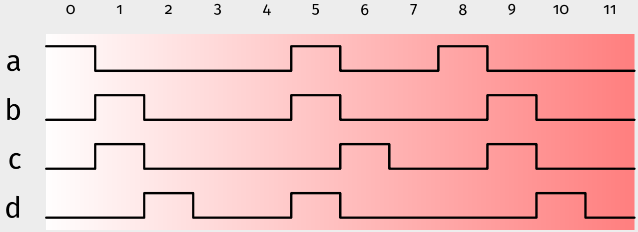
- Two *suffix implication* operators are defined:
    - `|->`
        - When the left-hand side finishes, then the right-hand side starts
    - `|=>`
        - When the left-hand side finishes, the right-hand side starts on the next clock cycle

Example

```
-- Sequence {c;d} is observed when {a;b} finishes
{a;b} |-> {c;d}
-- Sequence {c;d} is observed immediately after {a;b} finishes
{a;b} |=> {c;d}
```

Suffix Implication

Example



PSL

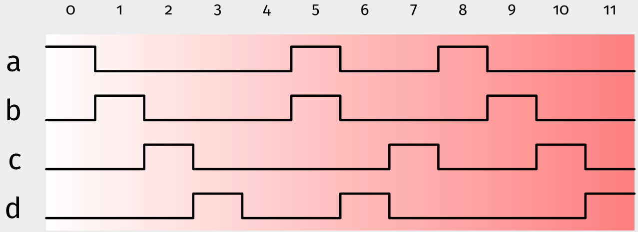
```
assert always({a;b} |-> {c;d});

assert always({a;b} |=> {c;d});

assert always({a and b} |-> {d;c});
```

Suffix Implication

Example



PSL

```
assert always({a;b} |-> {c;d});

assert always({a;b} |=> {c;d});
```

Operator never

- The `never` operator can be used with SEREs to ensure a sequence never happens
- Example:
  - Sequence a, b, c shall never be observed

```
assert never {a;b;c};
```

  - A read shall never precede a write

```
assert never {rd_o;wr_o};
```

  - A signal shall never be asserted on two consecutive cycles

```
assert never {r;r};
```

Repetition operators

- Repeating a sequence like `{a;a;a;a;a}` is not ideal as is

Operator	Description
`[*]`	Repeat a number of times between 0 and ∞
`[+]`	Repeat a number of times between 1 and ∞
`[*n]`	Repeat exactly *n* times
`[*n to m]`	Repeat a number of times between *n* and *m*
`[=n]`	*n* non consecutive repetitions
`[->n]`	*n* non consecutive goto repetitions

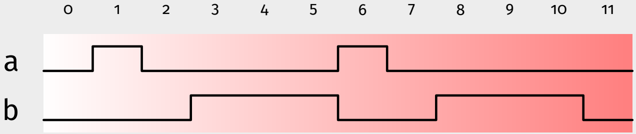
Repetitions: Examples

Sequence	Equivalent to
`{a;a;a;a;a}`	`{a[*5]}`
`{a;b;b;b;c}`	`{a;b[*3];c}`
`{a;b;c;b;c;d}`	`{a;{b;c}[*2];d}`

Sequence	Validated by
`{a;b;b;b;c}`	`{a;b[+];c}`
`{a;c}`	`{a;b[*];c}`
`{a;b;c;b;c;d}`	`{a;b[=2];d}`
`{a;b;c;b;c;d}`	`{a;c[->2];d}`

Repetitions

Exemple



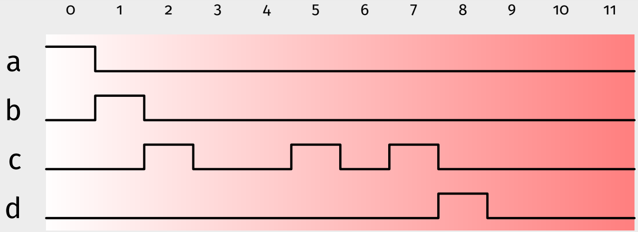
Assertions

```
assert always (a |-> {[*2];b});

assert always (a |-> {[*2];b[*3]});
```

Non consecutive repetitions

Example



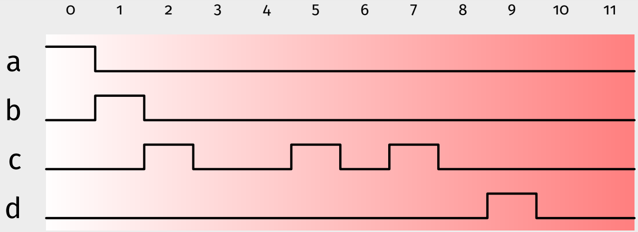
PSL

```
assert always({a} |=> {b;c[=3];d});
```


```
assert always({a} |=> {b;c[->3];d});
```

Non consecutive repetitions

Example



PSL

```
assert always({a} |=> {b;c[=3];d});

assert always({a} |=> {b;c[->3];d});
```

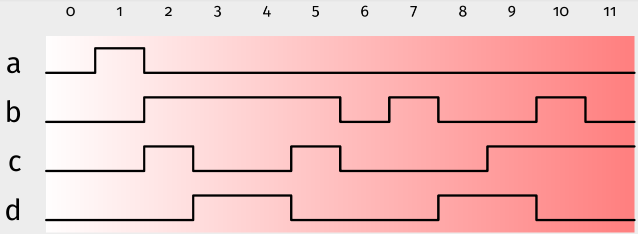
Sequences Combinations

Function	Description
`s1 ; s2`	concatenation of two sequences
`s1 : s2`	fusion of two sequences[1]
`s1 & s2`	non-length-matching *and* operator
`s1 && s2`	length-matching *and* operator
`s1 | s2`	*Or* operator
`s1 within s2`	s1 shall be observed during the execution of s2

[1]Concatenation, but s2 starts on the last cycle of s1. Not really used

Non-length-matching *and* operator

Exemple



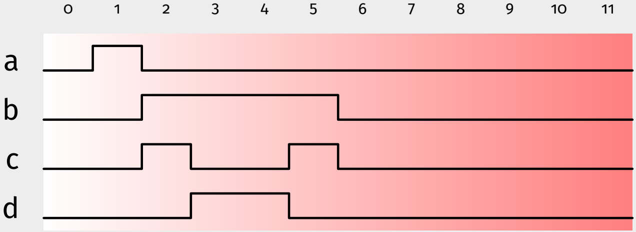
Assertions

```
assert always {a}|=>{{b[*4]} & {c; d}};

assert always {d[*2]}|=>{{c} & {b; not b}};
```

Length-matching *and* operator

Exemple



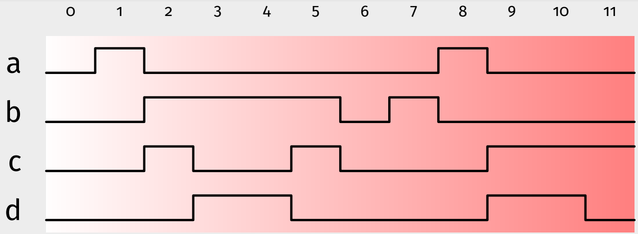
Assertions

```
assert always {a}|=>{{b[*4]} && {c; d[*2]; c}};
```


```
assert always {a}|=>{{b[*4]} && {c[*1 to 2]; d[*1 to 3]}};
```

Or operator

Exemple



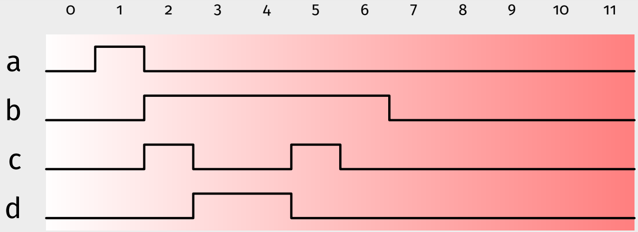
Assertions

```
assert always {a}|=>{{b[*4]} | {d; c}};

assert always {a}|=>{{[*1];d; b} | {d[*2]}};
```

s1 within s2

Example



PSL

```
assert always({a} |=> {{c;d[*2];c} within {b[*5]}});

assert always({a} |=> {{d[*2]} within {c;d[*2];c}});
```

Named sequences

- Like the properties, sequences can be named and then reused
  - Named sequences with parameters are not currently supported
- Particularly useful if a sequence is used in different contexts
  - For instance in an assert and a cover

PSL

```
sequence seq is {a; not a; a};

assert always({b} |=> {seq;d[*2];c});

cover seq;
```

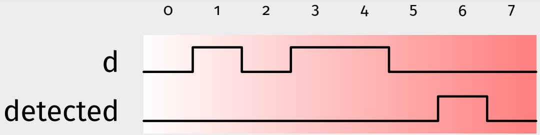
Detecting the end of a sequence

- Dealing with a signal that shall be asserted at the end of a sequence is tricky
- Examples:
  - A timer triggers after a certain number of clock cycles
  - A sequence detector triggers at the end of the sequence
- Some constructs exist, but are not supported by SBY (`ended()`)
- So, how to solve it?

Sequence detector

- Let's have a sequence detector that shall assert the signal `detected` when the sequence $d; d; \overline{d}; \overline{d}$ is observed (asserted at the same cycle as the last `not d`)

Example: valid chronogram

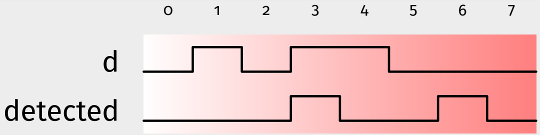


A nice assertion

```
   assert always({d;d;not d;not d} |-> detected);
-- or
   assert always({d[*2];(not d)[*2]} |-> detected);
```

Sequence detector

Invalid chronogram



- What happens then?

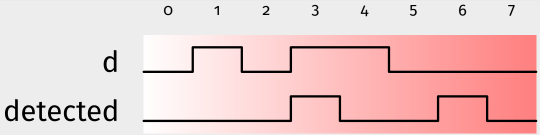
A nice assertion

```
assert always({d[*2];(not d)[*2]} |-> detected);
```

- Is it what we want?
- Well, our assertion does not say when `detected` shall stay low

Sequence detector

Invalid chronogram



- Idea : The signal should not be asserted twice before the sequence ends
  - Use `within` and the non consecutive goto repetition
  - Use the length-matching operator and the non consecutive goto repetition

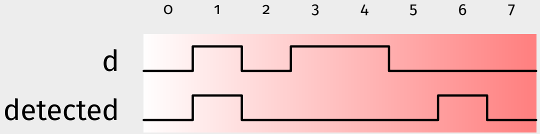
Assert that the signal does not go high twice within the sequence

```
assert never {{detected[->2]} within {d;d;not d;not d}};
assert never {{detected[->2]} && {d;d;not d;not d}};
```

- What is the issue here?

Sequence detector

Invalid chronogram



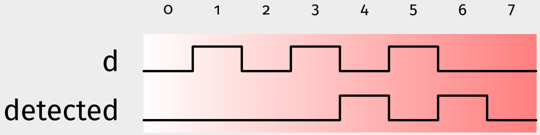
Assert that the signal does not go high twice within the sequence

```
assert never {{detected[->2]} within {d;d;not d;not d}};
assert never {{detected[->2]} && {d;d;not d;not d}};
```

- Well, maybe not the best idea
- And some issues with sequence overlapping

Sequence detector

Sequence now d;not d;d;not d : valid chronogram



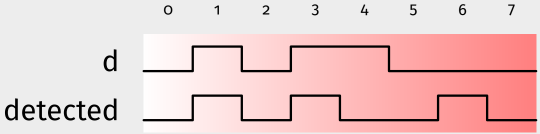
Assert that the signal does not go high twice within the sequence

```
assert never {{detected[->2]} within {d;not d;d;not d}};
assert never {{detected[->2]} && {d;not d;d;not d}};
```

- Well, once again, maybe not the best idea

Sequence detector

Example



- Idea : reverse the implication
  - Instead of `seq |-> det`, write `det |-> seq has occured`

Assert that the signal does not go high before the sequence ends

```
assert always (detected |->
            (d = '0' and prev(d) = '0' and prev(d,2) = '1' and prev(d,3) = '1'));
```

- Not easily generalizable. Imagine if the sequence is `{d[*3 to 12];(not d)[*5 to 15]}`

Sequence detector: Best option

- Use the following construct, specific to SBY:

```
attribute anyseq : boolean;
signal detected_test: std_logic;
attribute anyseq of detected_test : signal is true;   the signal is then a free variable
gen_test: assume always({d[*2];(not d)[*2]} |-> detected_test);
assert_low: assert always (detected -> detected_test);
assert_high: assert always({d[*2];(not d)[*2]} |-> detected);
```

- `gen_test` ensures that `detected_test` is high at the end of the sequence
  - Nothing more
  - So, it could be `'0'` or `'1'` at any other time
- `assert_low` only says that if `detected` is high, then `detected_test` **has to be high**
  - Nothing more, so
  - If `detected` was to be high at another time, as `detected_test` could have any value, `assert_low` would fail
- QED

Sequence detector: Conclusion

- The approach based on the assertion of the sequence detection and the free variable can be applied to any kind of sequence
  - A pure sequence detector
  - Triggering of a timer
  - ...

LTL - SERE equivalences

- LTL formula can be expressed in SERE-style

LTL	SERE
`a until b`	`{a[*];b}`
`a and next b`	`{a;b}`
`next[i] (a)`	`{[*i];a}`
`[next_a[i to j] (b)`	`{[*i];b[*j-i+1]}`
`[next_e[i to j] (b)`	`{[*i to j];b}`

Conclusion

- SEREs are powerful
- But careful with their definition you should be
- Do not be afraid of using coverage to know if a sequence has been observed or not
- Quite often LTL vs SERE is a matter of preference