

Example from Accellera Formal Verification working group document

Property 45.2

English: For an IO cycle, if the initiator asserts byte-enables of lesser significance than what is indicated by AD[1:0] the target must terminate the transaction with target abort

?? **Sugar:**

```
define target_abort := rose(devsel_) & fell(stop_);
define io_cmd := PCI_BE(3..0) in {IOREAD, IOWRITE};
define byte_enable_mismatch := (current_trans_ad(1..0)=1 & PCI_BE(0)!=1b) |
                                (current_trans_ad(1..0)=2 & PCI_BE(1..0)!=11b) |
                                (current_trans_ad(1..0)=3 & PCI_BE(2..0)!=111b);
define disconnect := !devsel_ & !stop_;

AG {(fell(frame_) & io_cmd), byte_enable_mismatch}
    (target_abort before (!trdy_ | disconnect))::clk=qclk
```

It is important to realize the Sugar ‘before’ operator is syntactic sugar and that the property above is (I think!) equivalent to:

```
{(fell(frame_) & io_cmd), byte_enable_mismatch} |=> (!(!trdy_ | disconnect))[*], target_abort ::clk=qclk
```

which is equivalent to:

```
{(fell(frame_) & io_cmd), byte_enable_mismatch} |=> (trdy_ & (devsel_|stop_))[*], target_abort ::clk=qclk
```

?? **ForSpec:**

```
less_significant(x,y) :=
    (y[1:0]=1 & x[0] < 0b1) | (y[1:0]=2 & x[1:0] < 0b11) | (ad[1:0]=3 & x[2:0] < 0b111);
bit[32] current_trans_ad;
```

```
assign_on(b_fall(frame)) current_trans_ad' = PCI_AD';
```

```
target_abort := fall(frame), !frame*, fall(devsel), !devsel*, devsel & fall(stop);
```

```
f := change_if(qclk) always io_cycle & less_significant(BE, current_trans_ad) -> next !frame*,
target_abort;
```

Note 1: it is doubtful the ‘target_abort’ expression is correct, because fall(frame) will only become true in the *next* transaction, rather than the current one. Looking at the diagram below, it is probable the definition should be something like:

```
target_abort := !devsel* & trdy, devsel & trdy & fall(stop);
```

?? **E**

```
event data_phase is (@irdy_assr and @trdy_assr)@qclk;
```

```
event io_data_phase is {
    @frame_fall
    and true(~PCI_BE'.as_a(PCI_COMMAND) in [IOREAD, IOWRITE]);
    ~[.] * (@frame_assr or @irdy_assr);
    @data_phase
} @qclk;
```

```
// signal target abort
event target_abort is {
    @frame_fall;
    {[.] * fail @frame_chng; @devsel_fall};
    {[.] * fail @devsel_chng; (@devsel_chng and @stop_fall)}
```

```

}@qclk;

expect (@io_data_phase and true(
    (current_trans_ad[1:0] == 1
    and `PCI_BE'[0:0] == 0)
    or
    (current_trans_ad[1:0] == 2
    and (`PCI_BE'[1:1] == 0 or `PCI_BE'[0:0] == 0))
    or
    (current_trans_ad[1:0] == 3
    and (`PCI_BE'[2:2] == 0 or `PCI_BE'[1:1] == 0 or `PCI_BE'[0:0] == 0))))
=> {
    [..] * @frame_assr;
    @target_abort
}@qclk;

```

Note 1 : because the irdy signal can be asserted when frame is de-asserted and this transition can occur before devsel is asserted, the satisfaction of the 'expect' should really be through:

```

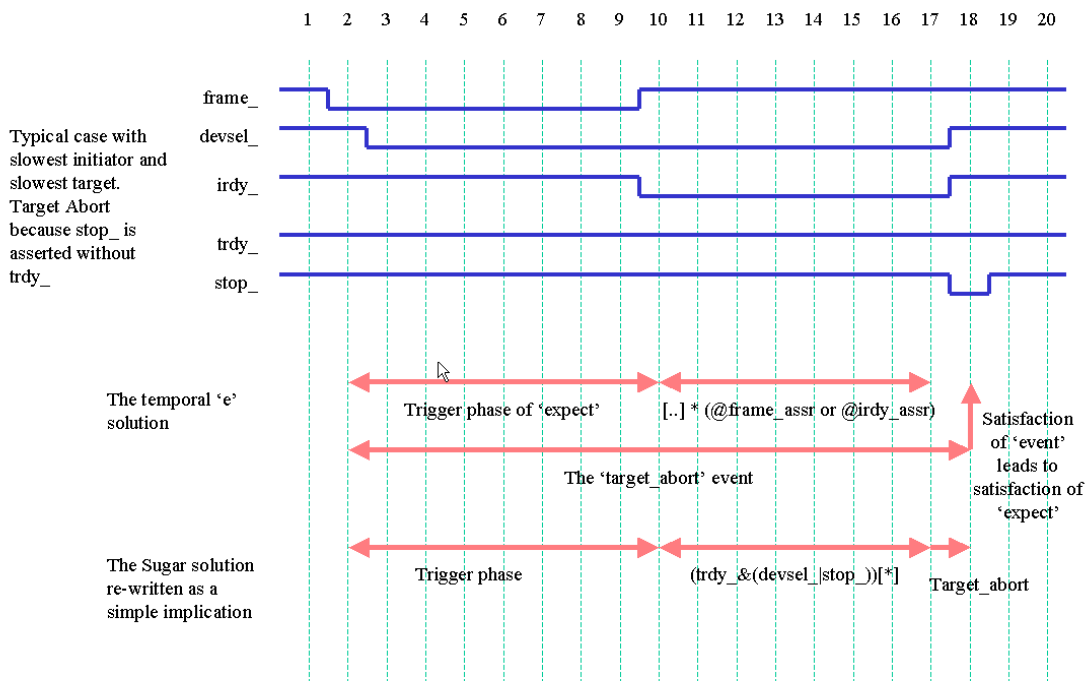
=> { [..] * (@frame_assr or @irdy_assr); @target_abort } @qclk;

```

.Note 2 : there is clearly a problem in the use of the 'data_phase' event at the end of the 'io_data_phase' event as the trdy_ signal will not be asserted during the Target-Abort sequence (see PCI 2.2 spec section 3.3.3.2.1)

Comparison

Ignoring issues of potential non-conformance with the PCI spec, the examples are believed to function in the following manner



It therefore seems that both ForSpec and the Linear Fragment of Sugar function as:

```

assert trigger_sequence -> satisfaction_sequence;

```

that is to say they expect the satisfaction_sequence to begin immediately after the trigger_sequence.

It seems that in ForSpec the beginning of the satisfaction_sequence could be delayed by m cycles through use of

```
-> next [m] !frame*, target_abort;
```

In contrast an 'e' expect is constructed as

```
expect @trigger_event => { [..]*@fill_time; @satisfaction_event} @qclk;
```

The major difference seems to be the success point for the temporal 'e' event is always at the end (tail) of the event, whereas both ForSpec and Sugar define where the start (head) of a sequence should begin.

The author of the 'e' example clearly attempted to get around the uncertainty of when the event will be satisfied through the (incorrect) "[..]* @frame_assr" construct with the expectation this would be satisfied all the time the transaction was being continued. This 'e' concept is clearly very powerful because it inherently allows many, many sequences (or 'events') to exist in parallel, indeed in this example the 'target_abort' event has a multi-cycle duration with multiple non-deterministic delays and, as can be seen from the diagram, the start of the event is (coincidentally in this case) coincident with the start of the trigger phase.

The nearest equivalent in 'e' might be to write

```
-> next [-m] !frame*, target_abort;
```

though the problem in this case is there is no knowledge of the magnitude of m, even if a negative value were to be allowed.

Questions

The **BIG** questions are:

- 1. What do the hard to comprehend semantics of 'next' in ForSpec really mean in English?***
- 2. If the interpretation of 'next' is correct above, how could two languages with such radically different approaches to sequences / events be unified?***
- 3. Again, if the interpretation of 'next' is correct above, on the basis of this example how can ForSpec be considered closer to Temporal 'e' than the Linear Fragment of Sugar?***