OperationalSVA
An SVA Modeling Layer to Ease Assertion Development from Timing Diagrams

OneSpin Solutions

www.onespin-solutions.com

Overview

- Motivation
- DMA Controller Example
- Capturing Timing Diagrams in SVA
- Summary

Motivation

Standard Assertion languages (SVA/PSL) are rich, expressive, powerful, and broadly deployed.

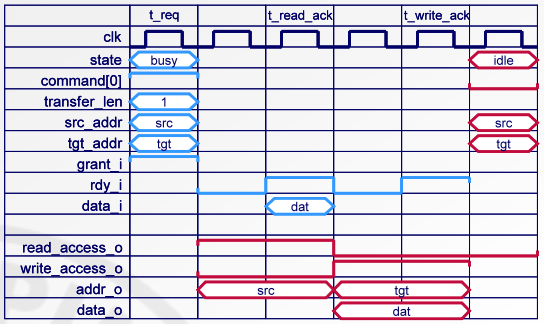
Use of advanced assertions is still limited:

- Advanced operators have complex semantics
- Advanced assertions become
  - hard to read and comprehend
  - hard to debug
  - difficult to communicate/share between teams
- There is no common approach for visualization

Most users **are reusing libraries** of simple generic assertions.
Use of **powerful advanced assertions** is mostly limited to experts.

An approach to ease working with SVA

- The most common way to express, share and communicate temporal behavior of designs are timing diagrams



Timing Diagrams

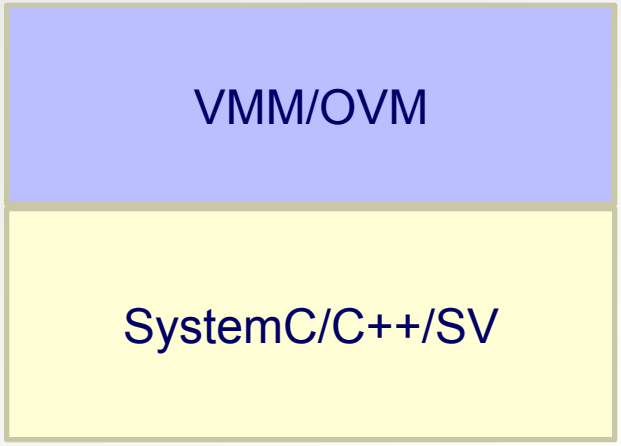
- Universally used to describe intended behavior of RTL designs
- All engineers familiar with timing diagrams
- Common basis to share design understanding

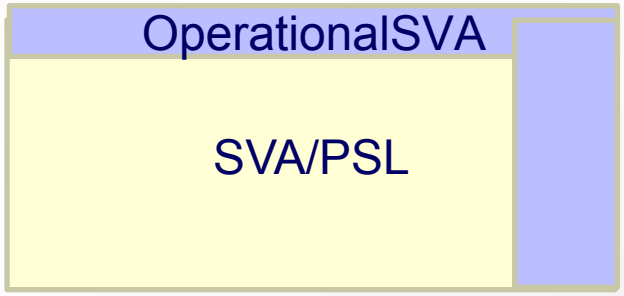
⇒ **Excellent basis for assertion development**

⇒ **But: timing diagrams difficult to express naturally in plain SVA/PSL**

Idea: A simple modeling layer for SVA (PSL) that allows to develop high-level assertions directly from timing diagrams

Modeling Layers in Functional Verification





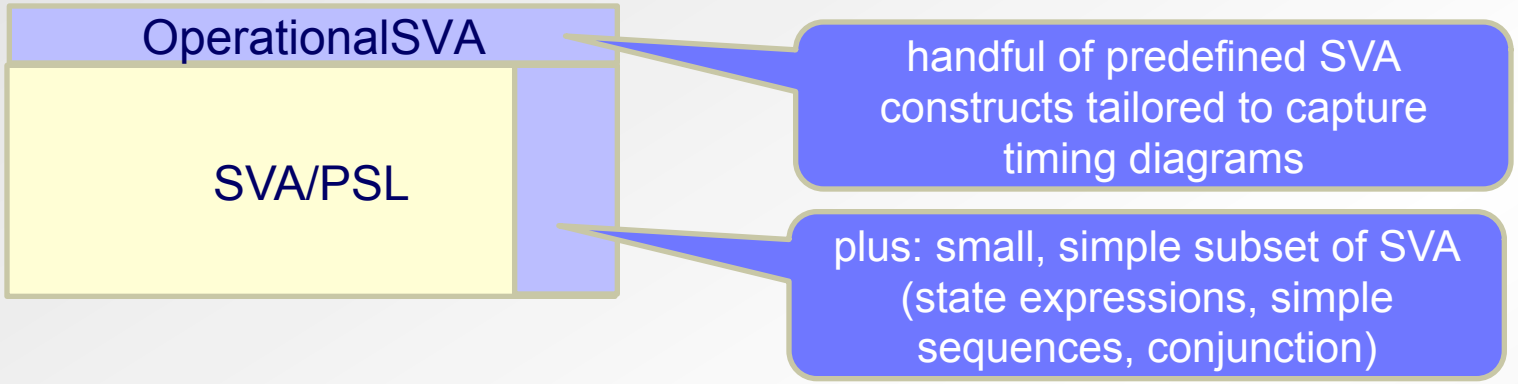
VMM/OVM:

- Modeling layers that ease and speed construction of transaction-level, constrained random testbenches

OperationalSVA:

- Simple, small modeling layer that eases and speeds assertion development working from timing diagrams
- Eases capture of functional requirements, design operations and transactions
- For simulation and formal verification

Basic Rationale of OperationalSVA



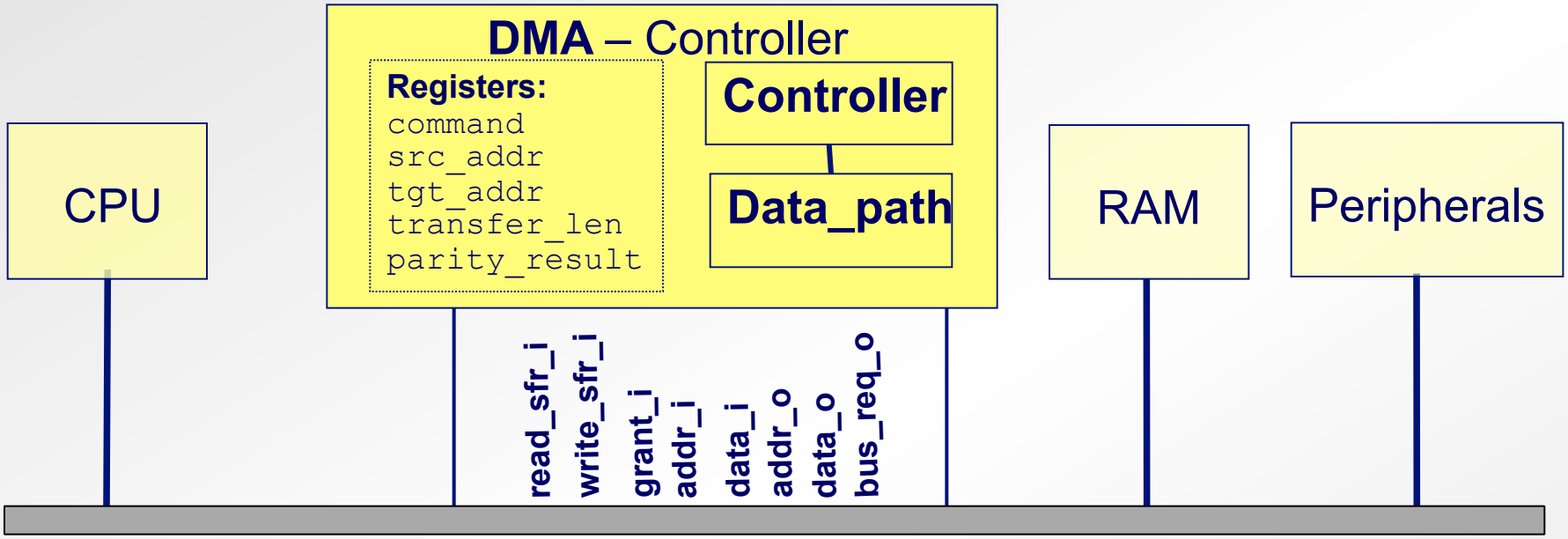
Why standardize OperationalSVA?

- Assertion development based on familiar “mind-set” (timing diagrams)
- Simple subset of SVA, easier to get started (does not require/use majority of complex operators)
- Less error-prone capture of complex temporal behavior
- Assertions are easy to visualize as timing diagrams
- Assertions become easier to read, understand, debug, maintain and share
- Assertions are easier to review for compliance with informal specification

Experience:

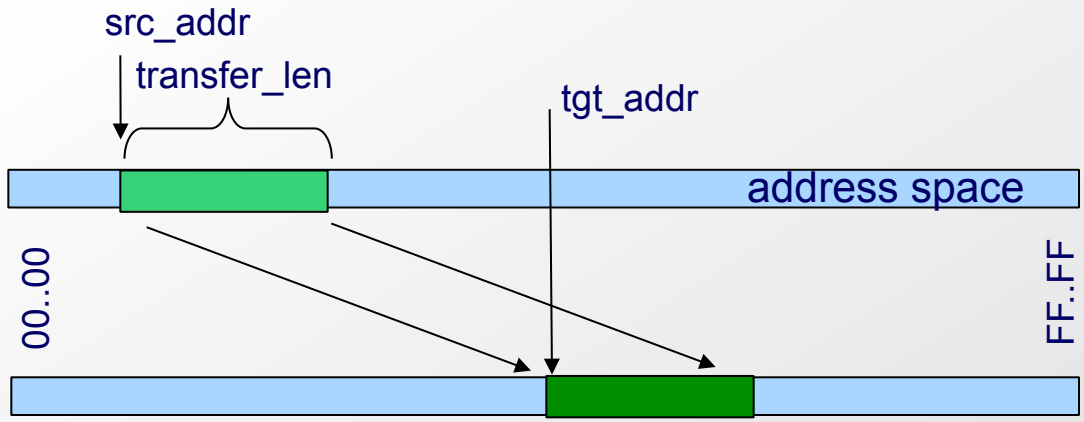
- The underlying modeling concepts of OperationalSVA have been established, tried and tested in verifying **more than 200 customer designs**

DMA Controller Example



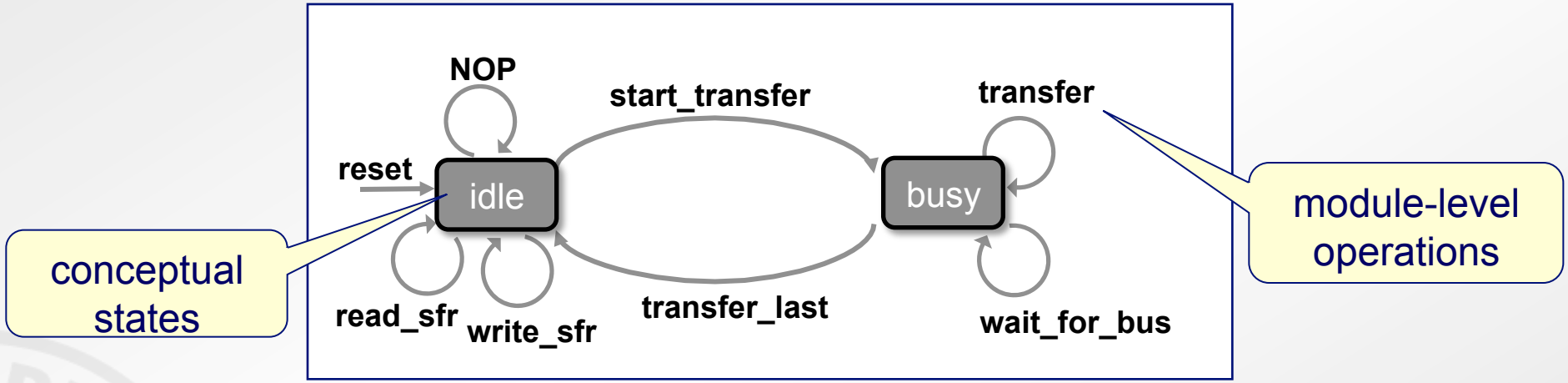
The Design

- CPU configures data transfer
- DMA: data transfers to/from RAM/peripherals, plus optional parity check
- 800 Lines of RTL Code



High-Level View of The DMA Controller

- the **module-level operations** of the DUV
- the **conceptual states** of the DUV

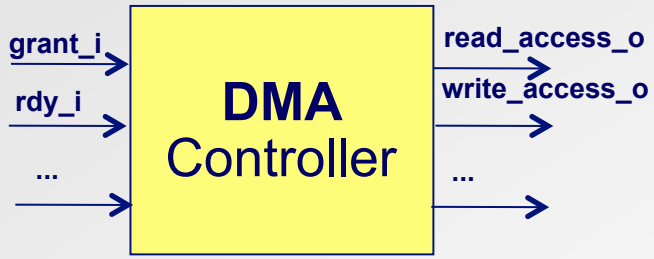


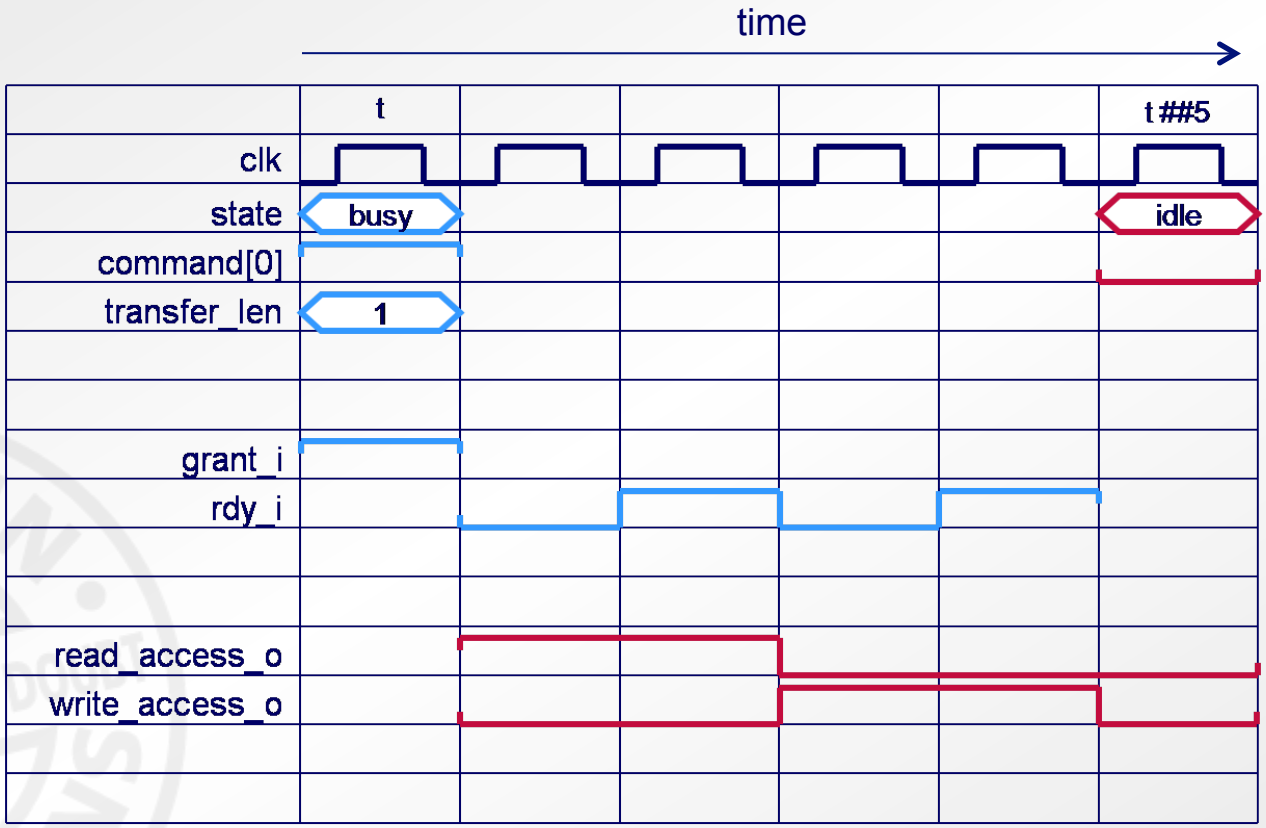
Intended behavior of module-level operations can be conveniently described by means of Timing Diagrams

Timing Diagram Concepts

- Separation of *cause* and *effect* in timing diagrams
- Defining and referencing multiple time points
- Capturing and referencing data
- Usage of time intervals
- Event-triggered time points

Operation "*transfer_last*": First Version





Timing Diagrams

cause

⬇

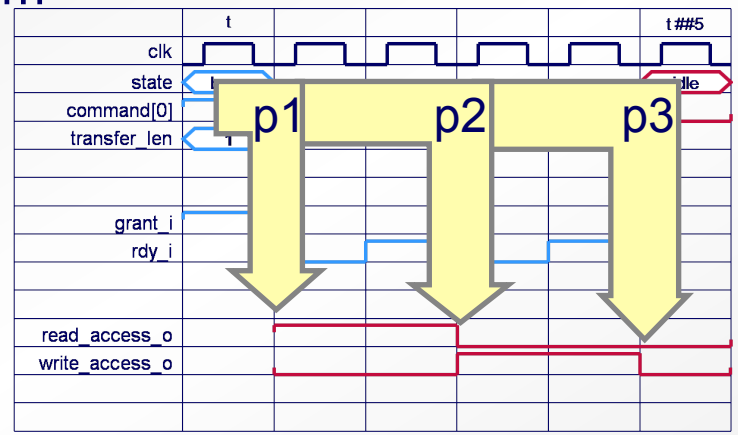
effect

SVA:

cause |=> effect

"*transfer_last*" in plain SVA

Standard approach: "slicing" of timing diagram




```
property transfer_last_p1;
    ( state == busy && transfer_len = 1 &&
      command[0] && grant_i
     ##1 !rdy_i
      |-> (read_access_o && !write_access) [*2])
endproperty
property transfer_last_p2;
   ( state == busy && transfer_len = 1 && command[0] && grant_i
     ##1 !rdy_i  ##1 rdy_i ##1 !rdy_i
     |-> (!read_access_o &&  write_access_o) [*2])
endproperty
property transfer_last_p3;
    ((state == busy && transfer_len = 1 && command[0] && grant_i
     ##1 !rdy_i  ##1 rdy_i ##1 !rdy_i  ##1 rdy_i)
     |=> (state == idle && command == 0 && !read_access_o &&  !write_access_o));
endproperty
```

"*transfer_last*" modeled using OperationalSVA

starting condition

expected input sequence

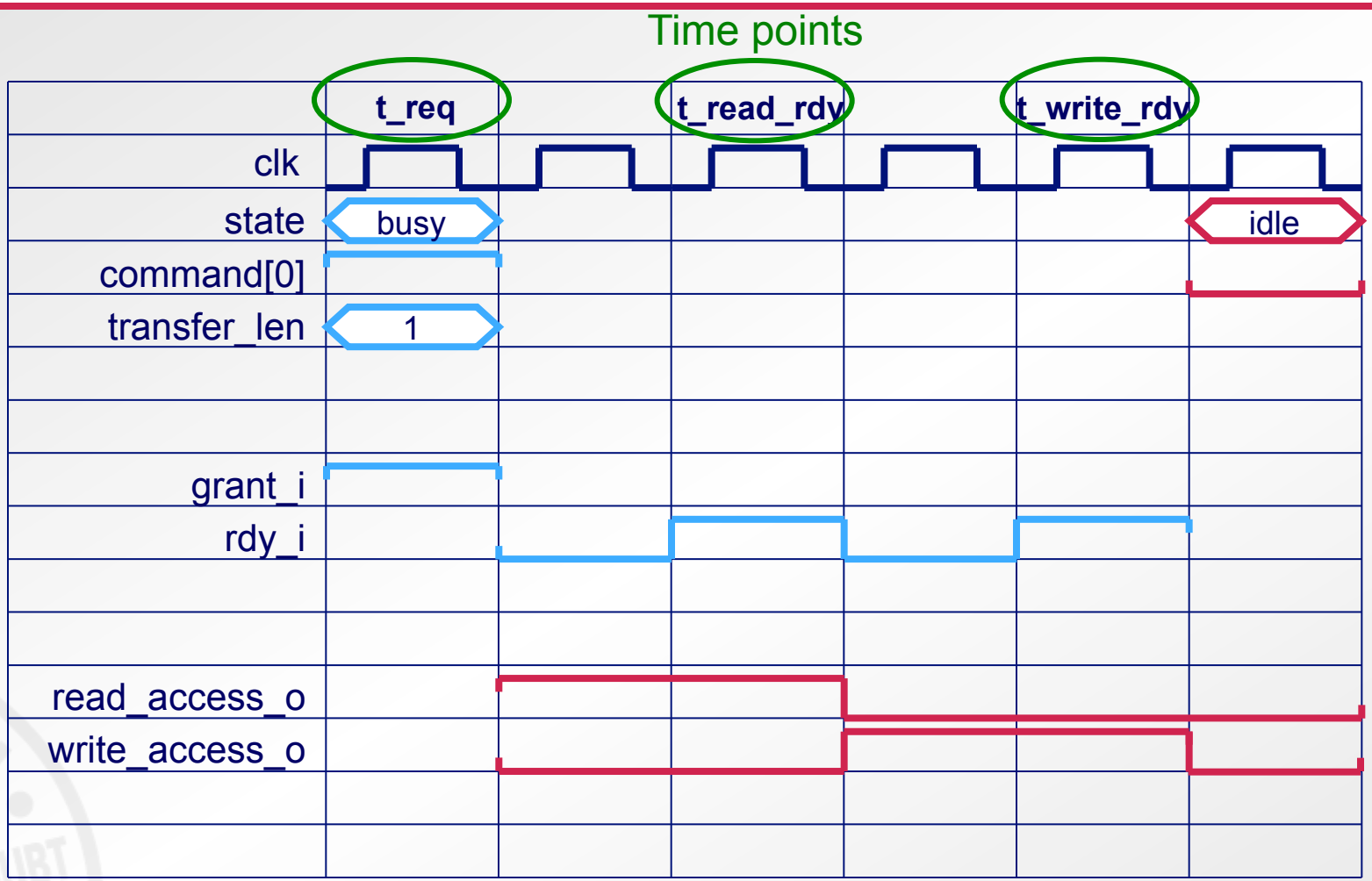
```
property transfer_last;
        state == busy  && transfer_len == 1 && command[0] && grant_i
        ## 1 ! rdy_i ## 1 rdy_i ## 1 ! rdy_i ## 1 rdy_i
implies
        ## 1  ( read_access_o && !write_access_o) [*2]
        ## 1  (!read_access_o && write_access_o) [*2]
        ## 1  (state == idle && command[0]  ==  0 && !read_access_o && !write_access_o);
endproperty
```

intended output behavior

Timing Diagram Concepts

- Separation of *cause* and *effect* in timing diagrams
- Defining and referencing multiple time points
- Capturing and referencing data
- Usage of time intervals
- Event-triggered time points

Operation "*transfer_last*": Second Version



Referencing Time Points Using OperationalSVA

```
sequence t_req; t; endsequence
sequence t_read_rdy; nxt(t_req, 2); endsequence
sequence t_write_rdy; nxt(t_read_rdy, 2); endsequence

property transfer_last;

    t_req       ## 0  (state == busy && transfer_len == 1 && command[0] && grant_i) and
    t_req       ## 1  (! rdy_i ## 1 rdy_i ) and
    t_read_rdy  ## 1  (! rdy_i ## 1 rdy_i )

implies
     t_req       ## 1  ( read_access_o && !write_access_o) [*2] and
     t_read_rdy  ## 1  (!read_access_o && write_access_o)[*2] and
     t_write_rdy ## 1  (state == idle && command[0]  ==  0 && !read_access_o && !write_access_o);

endproperty
```

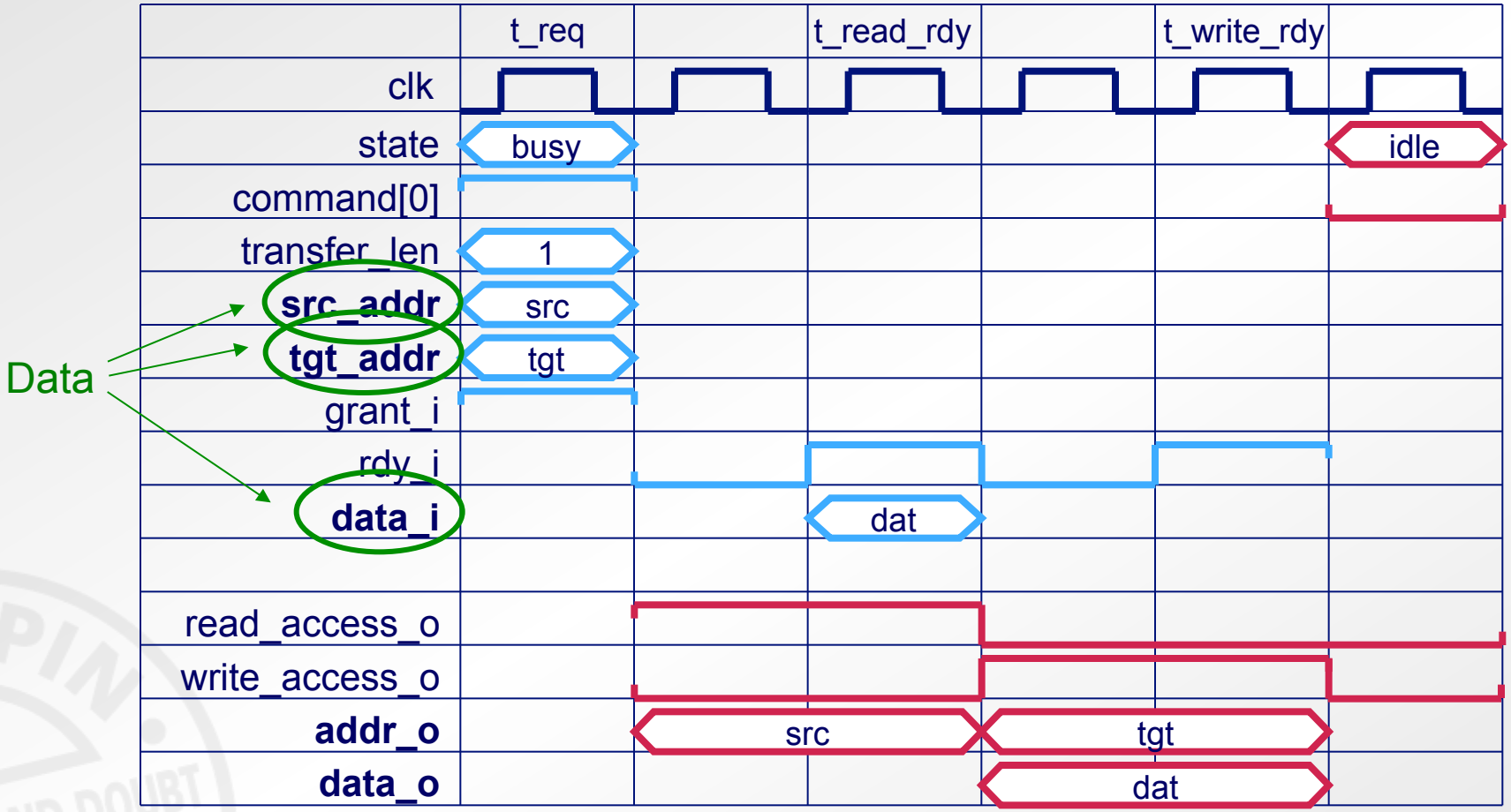
define timepoints

describe behavior relative to timepoints

Timing Diagram Concepts

- Separation of *cause* and *effect* in timing diagrams
- Defining and referencing multiple time points
- Capturing and referencing data
- Usage of time intervals
- Event-triggered time points

Operation "*transfer_last*": Third Version



Referencing Data Using OperationalSVA

```
sequence t_req; t; endsequence
sequence t_read_rdy; nxt(t_req, 2); endsequence
sequence t_write_rdy; nxt(t_read_rdy, 2); endsequence

property transfer_last;
forall bit[data_msb_g:0]  src, dat, tgt;
    t_req        ## 0  (state == busy  && transfer_len == 1 && command[0]  && grant_i) and
    t_req        ## 1  (! rdy_i ## 1 rdy_i ) and
    t_read_rdy   ## 1  (! rdy_i ## 1 rdy_i ) and
    t_req        ## 0  src == src_addr  &&
                       tgt == tgt_addr  and
    t_read_rdy   ## 0  dat == data_i

implies
    t_req        ## 1  ( read_access_o && !write_access_o && addr_o == src) [*2] and
    t_read_rdy   ## 1  (!read_access_o &&  write_access_o && addr_o == tgt && data_o== dat)[*2] and
    t_write_rdy  ## 1  (state == idle && command[0]  ==  0 && src_addr  == src  && tgt_addr  == tgt
                        && !read_access_o && !write_access_o );
endproperty
```

registers to store data

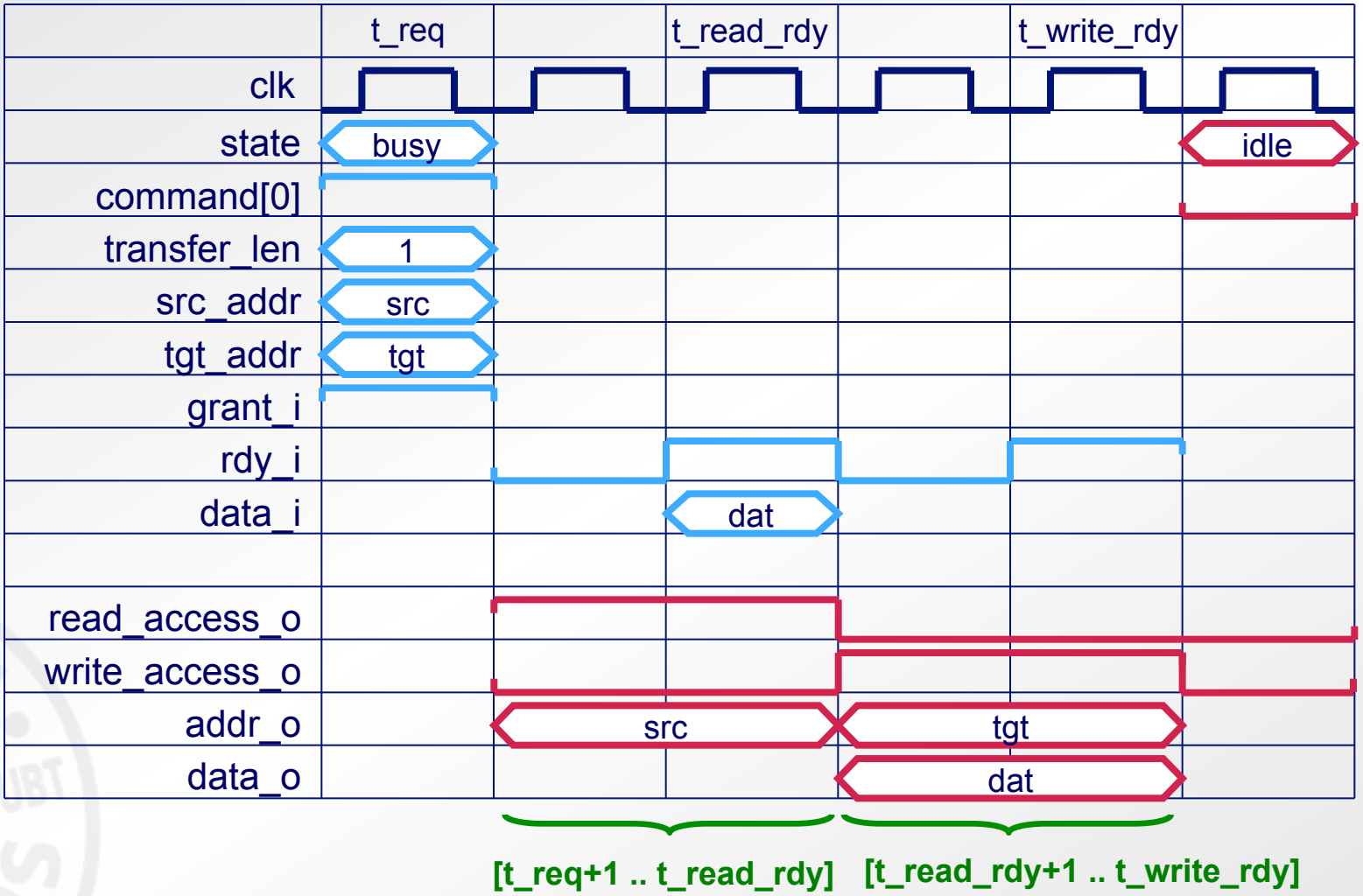
„freeze“ data

reference data

Timing Diagram Concepts

- Separation of *cause* and *effect* in timing diagrams
- Defining and referencing multiple time points
- Capturing and referencing data
- Usage of time intervals
- Event-triggered time points

Operation "*transfer_last*": Fourth Version



Referencing time intervals using OperationalSVA

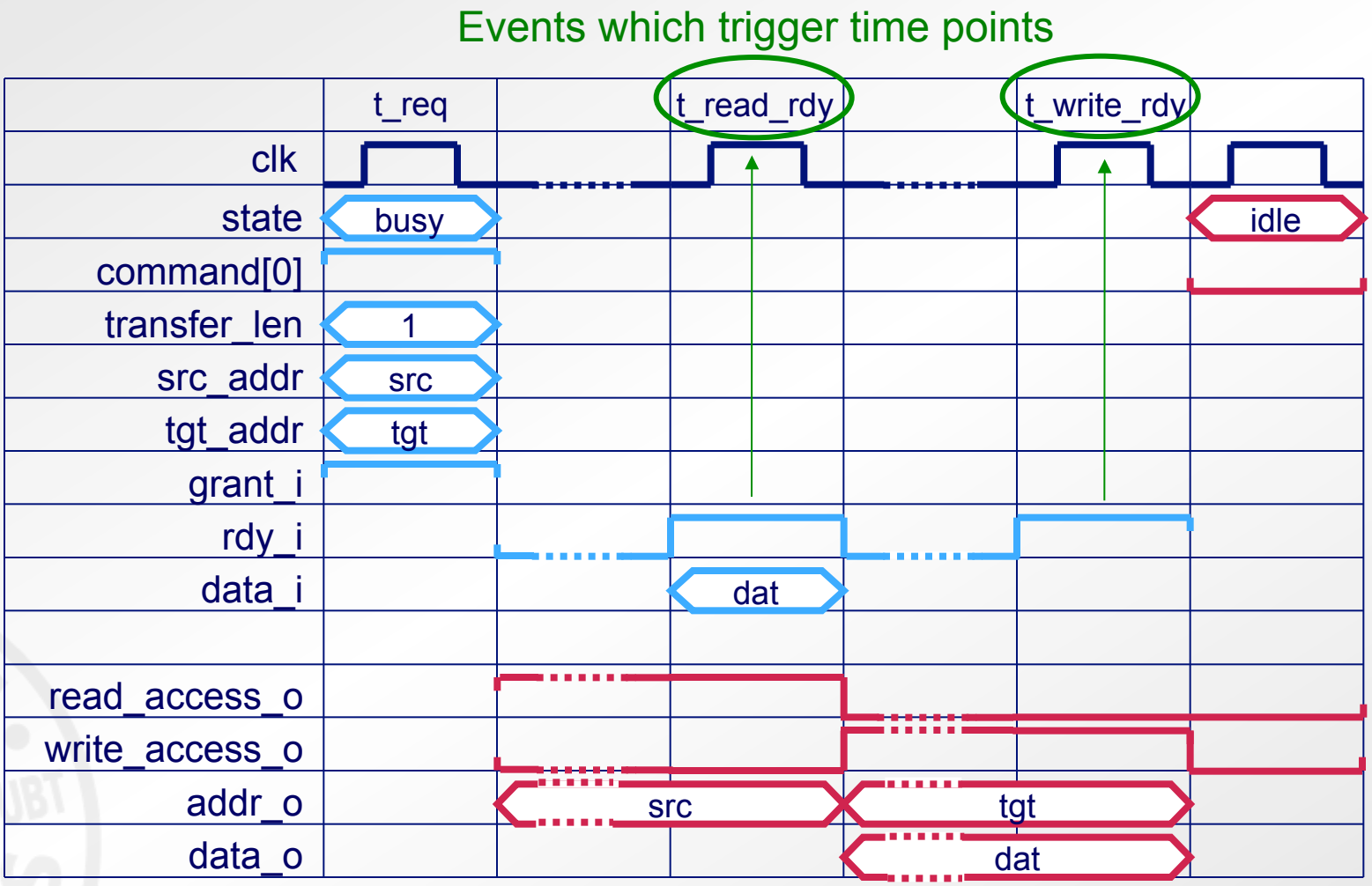
```
sequence t_req; t; endsequence
sequence t_read_rdy; nxt(t_req, 2); endsequence
sequence t_write_rdy; nxt(t_read_rdy, 2); endsequence

property transfer_last;
forall bit[data_msb_g:0]  src, dat, tgt;
    t_req        ## 0  (state == busy  && transfer_len == 1 && command[0]  && grant_i) and
    t_req        ## 1  (! rdy_i ## 1 rdy_i ) and
    t_read_rdy   ## 1  (! rdy_i ## 1 rdy_i ) and
     t_req       ## 0  src == src_addr  &&
                       tgt == tgt_addr  and
    t_read_rdy   ## 0  dat == data_i

implies
     during(nxt(t_req,1),         t_read_rdy,  (read_access_o && !write_access_o && addr_o == src)) and
     during(nxt(t_read_rdy,1), t_write_rdy, (!read_access_o &&  write_access_o && addr_o == tgt &&
                                             data_o== dat) and
    t_write_rdy ## 1 (state == idle && command[0]  ==  0 && src_addr  == src  && tgt_addr  == tgt &&
                      !read_access_o && !write_access_o );
endproperty
```

expression holds during time interval

Timing Diagram Concepts

- Separation of *cause* and *effect* in timing diagrams
- Defining and referencing multiple time points
- Capturing and referencing data
- Usage of time intervals
- Event-triggered time points

Operation "*transfer_last*": Final Version



Referencing events using OperationalSVA

```
sequence t_req; t; endsequence
sequence t_read_rdy; await(nxt(t_req, 1),         rdy_i, max_wait); endsequence
sequence t_write_rdy; await(nxt(t_read_rdy, 1), rdy_i, max_wait); endsequence


property transfer_last;
forall bit[data_msb_g:0]  src, dat, tgt;
    t_req          ## 0  (state == busy  && transfer_len == 1 && command[0]  && grant_i) and
    t_req          ## 0  src == src_addr  &&
                         tgt == tgt_addr  and
    t_read_rdy   ## 0  dat == data_i

implies
     during(nxt(t_req,1),          t_read_rdy, ( read_access_o && !write_access_o && addr_o == src)) and
     during(nxt(t_read_rdy,1), t_write_rdy, (!read_access_o &&  write_access_o && addr_o == tgt &&
                                                 data_o== dat) and
     t_write_rdy ## 1  (state == idle && command[0]  ==  0 && src_addr  == src  && tgt_addr  == tgt &&
                                                 !read_access_o && !write_access_o);
endproperty
```

event-triggered timepoint

OperationalSVA Constructs Summary

defined in SVA

Mechanisms to define time points

- keyword: **t** (first time point in timing diagram)
- offsets: t_data = **nxt(t,3)** (3 cycles after t)
- **await**: event-triggered time point

Simple operators to describe behavior relative to time points

- **tp ##n sequence** (sequence starts n cycles after time point tp)
- **during(tp1, tp2, exp)** (exp holds during time interval [tp1 .. tp2])

Property implication

- **„implies“** keyword as in SVA2009

Proposed Extension:

Allow declaring free variables

- **forall <type> <local_var>** (treat local_var as a free variable)

Summary of OperationalSVA

OperationalSVA :

- handful of predefined SVA constructs tailored to develop assertions from timing diagrams + very simple subset of SVA
- simple, intuitive, yet powerful way to capture temporal behavior
- Concept is neutral to the choice of **SVA vs PSL**: fosters the communication **between the two communities**.
- Provides a basis for **formal coverage analysis**

- Assertion development based on familiar “mind-set” (timing diagrams)
- Simple subset of SVA, easier to get started
- Less error-prone capture of complex temporal behavior
- Assertions are easy to visualize (as timing diagrams)
- Assertions become easier to read, understand, debug, maintain and share
- Assertions are easier to review for compliance with informal specification

Only required SVA extension: “forall variables” (as in PSL)

Thank You