Shutdown with Agreements in a UVM Testbench

Mark Glasser

NVIDIA Corporation

March 22-23, 2017

Silicon Valley

Agenda

Shutdown Issues

Quiescence

Barriers

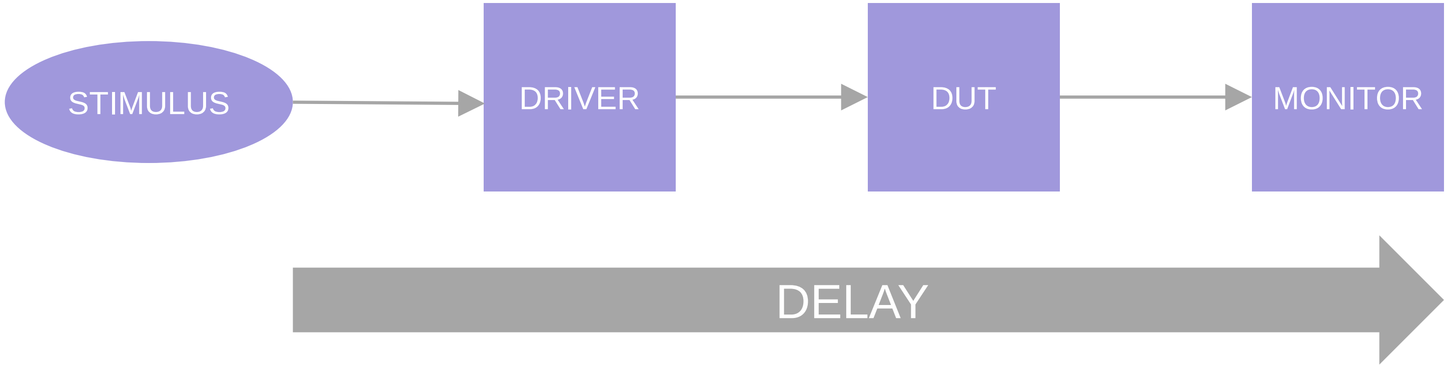
Agreements

What about Objections?

Conclusion

Shutdown Issues

- Delay between stimulus generation and response observation.



- Incomplete stimulus means incomplete or incorrect coverage
- Many things can be going on simultaneously, some affect coverage, some do not.
- Each entity knows whether it is done or not, but it does not know the state of other entities.

Quiescence

- In a state of inactivity or dormancy. Quiet.
- E.g. between instructions or operations; between I/O transfers; etc.
- Nothing of significance is going on
  - Clocks may be operating
  - Idle state
- No transaction is in flight
- Each entity can identify its own quiescent point

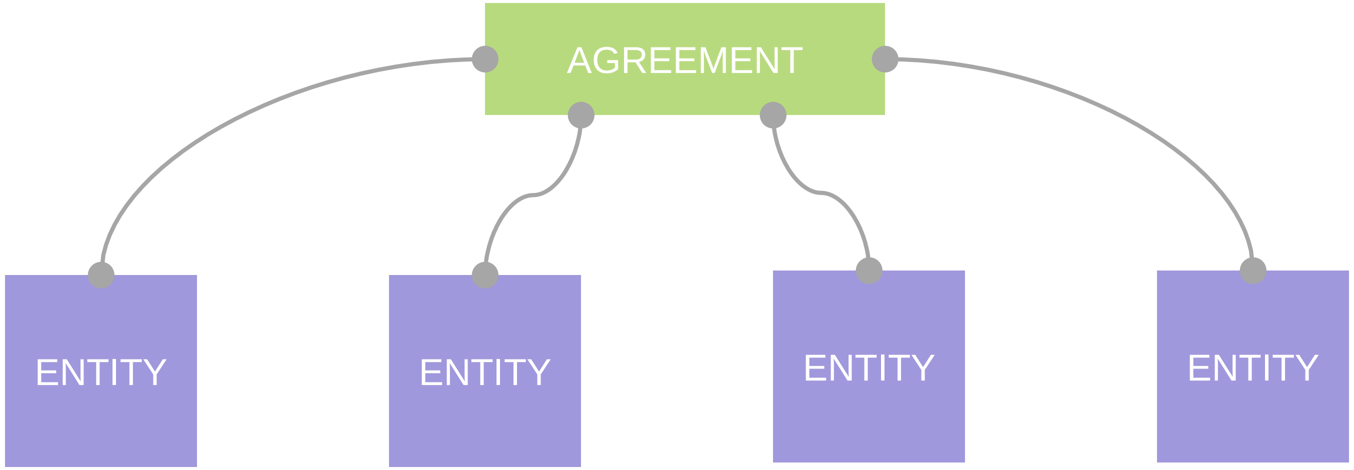
Shutdown on Quiescence

- Quiescent points are opportunities for shutdown.
- No transactions are in flight.
- Coverage is up-to-date.

- Shutdown problem: find quiescent points then initiate shutdown.

Agreements

- An `agreement` is an object used to determine quiescence.
- Agreements are shared amongst entities with an interest in when shutdown occurs.
  - Not all entities necessarily have an interest.
- Entities use agreement to vote for or against shutdown.
  - Drivers, monitors, coverage collectors, scoreboards, etc.



Agreements and Quiescence

- Quiescent points are synthesized by agreements.
- The system is quiescent when all participating entities agree that it is.
- Each participating entity is responsible for identifying local quiescence by registering its vote with the agreement object.

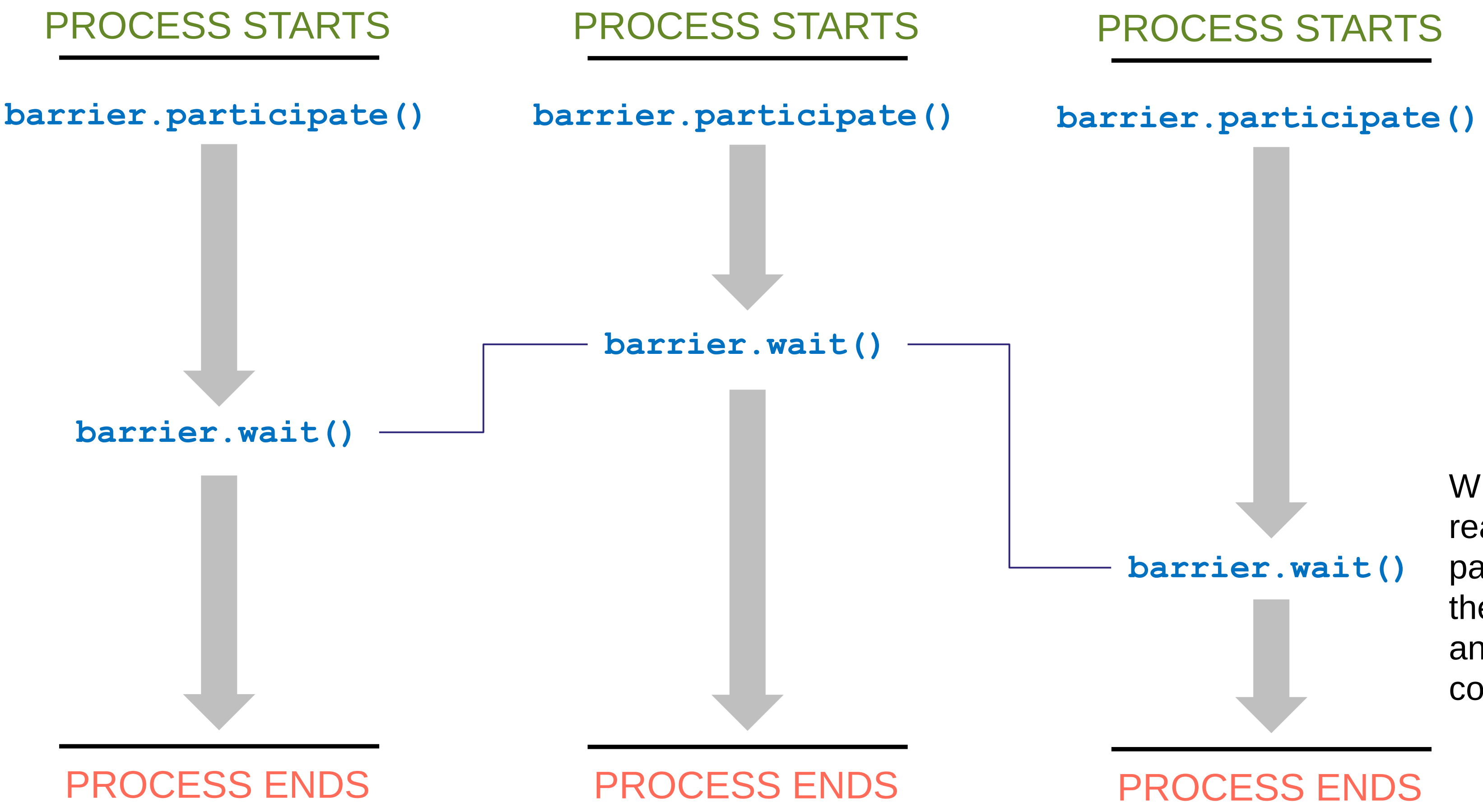
Barriers

- Shared primitive for synchronizing processes.
- All participating processes block until they all reach the synchronization point.
- **`uvm_barrier`** is implemented using a simple integer and wait(expr).

Process Synchronization with Barriers

Announce participation; increment number of waiters.

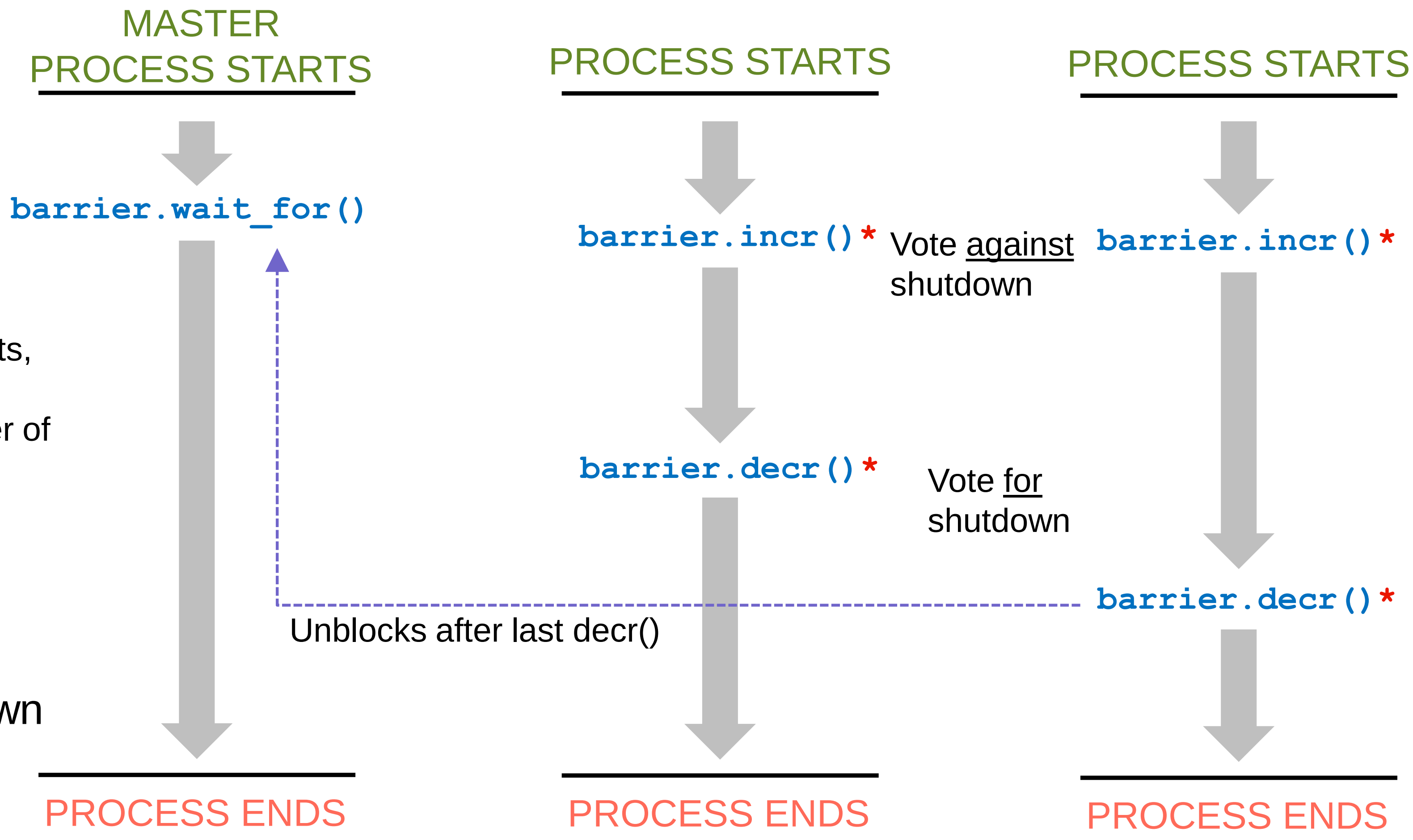
Wait() blocks until all processes reach their respective wait() calls.



When wait() is reached for all participating barriers, the block is released and all processes continue forward.

A Different Use Model for Barriers

Only one process waits, others increment and decrement the number of waiters on the barrier.



MASTER PROCESS STARTS

`barrier.wait_for()`

Unblocks after last decr()

Shutdown

PROCESS ENDS

PROCESS STARTS

`barrier.incr()`*

Vote against shutdown

`barrier.decr()`*

Vote for shutdown

PROCESS ENDS

PROCESS STARTS

`barrier.incr()`*

`barrier.decr()`*

PROCESS ENDS

Shutdown begins when all participating barriers have zero waiters.

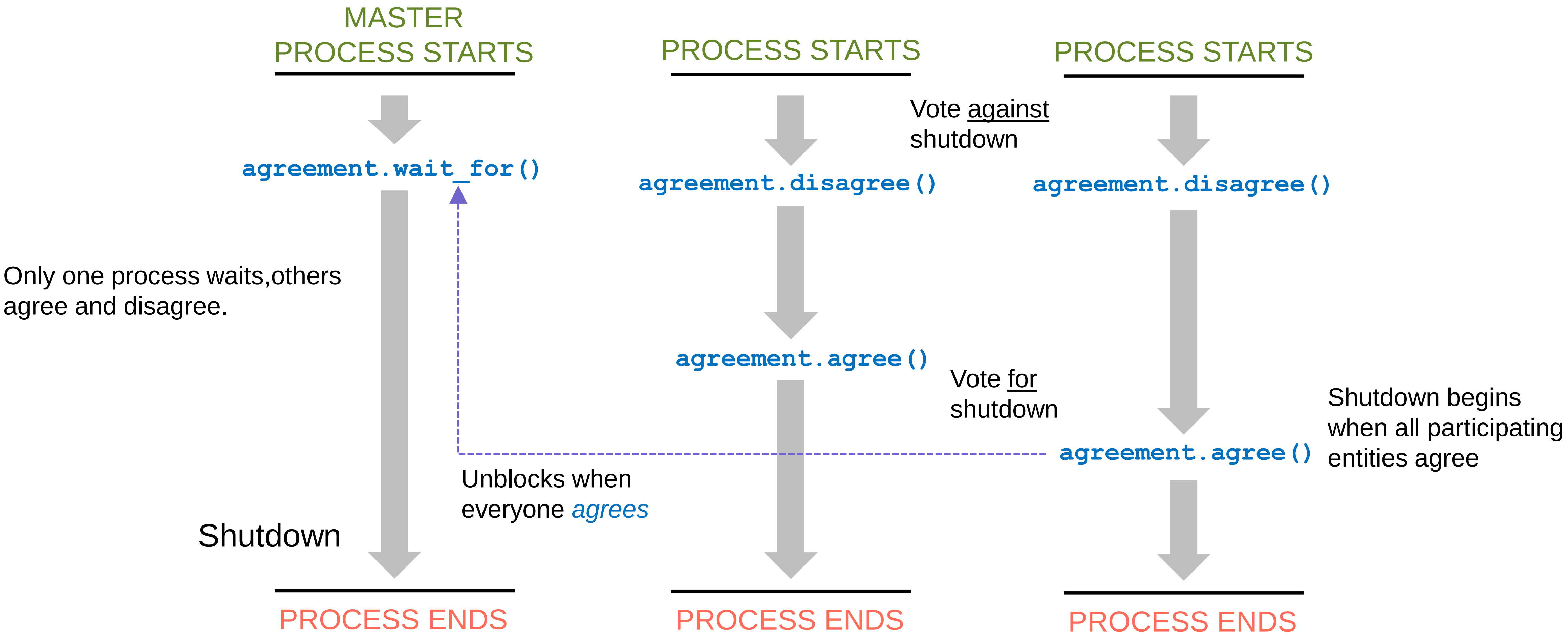
*real API [incr] ➡ `barrier.set_threshold(barrier.get_threshold() + 1)`

real API [decr] ➡ `barrier.set_threshold(barrier.get_threshold() - 1)`

Agreements

- “Inside out” barrier.
- One wait() call in master process.
- Multiple calls to increment and decrement number of waiters by other processes.
- Incr() == don’t shutdown yet.
    - Vote against shutdown
- Decr() == it’s OK to shutdown now.
    - Vote for shutdown

Agreements instead of Barriers



Only one process waits,others agree and disagree.

Vote against shutdown

Vote for shutdown

Unblocks when everyone *agrees*

Shutdown begins when all participating entities agree

Changing Your Vote

- An entity can change its vote at any time.
- Multiple calls to agree() without any intervening disagree() calls can be changed with one disagree() call.
- Multiple calls to disagree() without any intervening agree() calls can be changed with one agree() call.
- Do not have to exactly match agrees and disagrees.
- Important when clearing state when asynchronous events occur such as resets and interrupts.

```
agmt.agree();
agmt.agree();
agmt.agree();
agmt.disagree();
```

Current state == “disagree”

Agreement API

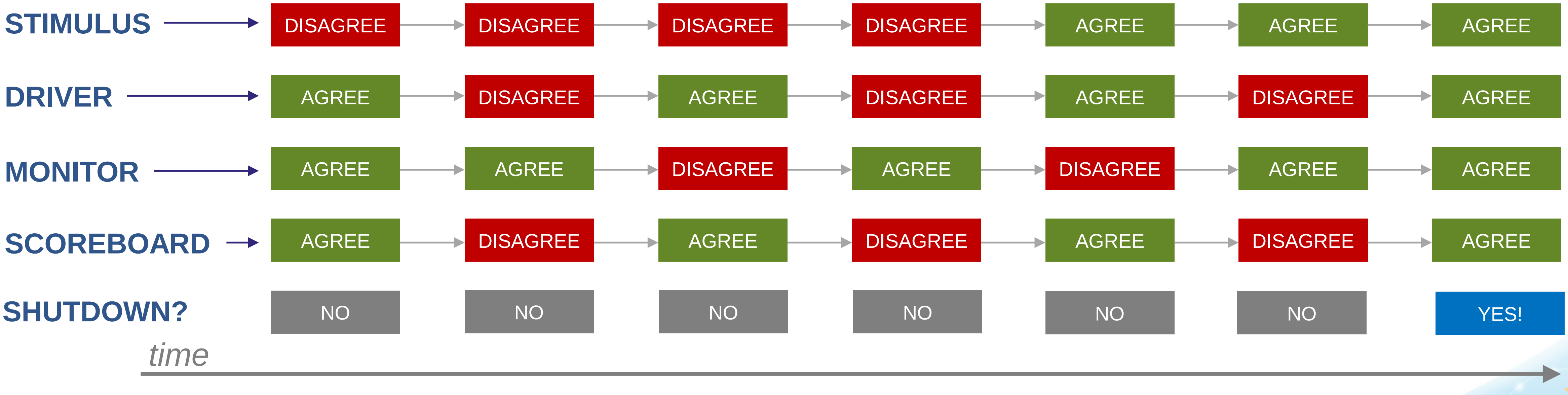
```
class agreement extends uvm_barrier;

  function new(string name = "agreement");
  static function void set_debug();
  static function void clr_debug();
  function void agree(uvm_object obj = null);
  function void disagree(uvm_object obj = null);
  task wait_for_agreement(uvm_object obj = null);
  function void clear(uvm_object obj);
  local function string status_msg(uvm_object obj = null);

endclass
```

Using Agreements

- Agreements provide a distributed voting mechanism.
- Using resource database or pools to provide access to agreement objects.
- Agreements do not require agree() and disagree() calls to match exactly.
  - Agreements allow increment and decrement to occur only once.
  - A single disagree() calls can change vote for multiple agree() calls and vice versa.



STIMULUS	DISAGREE	DISAGREE	DISAGREE	DISAGREE	AGREE	AGREE	AGREE
DRIVER	AGREE	DISAGREE	AGREE	DISAGREE	AGREE	DISAGREE	AGREE
MONITOR	AGREE	AGREE	DISAGREE	AGREE	DISAGREE	AGREE	AGREE
SCOREBOARD	AGREE	DISAGREE	AGREE	DISAGREE	AGREE	DISAGREE	AGREE
SHUTDOWN?	NO	NO	NO	NO	NO	NO	YES!

What About Objections?

- Objections are inefficient and cumbersome.
- Objections spawn a process to manage drain time.
- Objections do hierarchical objections (no longer by default).
- Objections require raises and drops to match exactly.
  - only increment and decrement counter.
  - Very difficult to match raises and drops in the presence of asynchronous events.

Agreements + Objections

```
task run_phase(uvm_phase phase);
  agreement ok_to_stop;

  if(!uvm_resource_db#(agreement)::read_by_name(get_full_name(),
                                   "ok_to_stop", ok_to_stop, this))
      `uvm_fatal("ENV/AGREEMENT", "No shutdown agreement available")

  phase.raise_objection(this);

  fork
    seq.start(sqr); // could be multiple sequences or other processes
  join_none

  #0; #0; #0; #0;  // 3 delta cycles to start a sequence
  ok_to_stop.wait_for_agreement(this, `__LINE__);
  phase.drop_objection(this);
endtask
```

Conclusion

- Agreements are simple to use.
- Agreements are safer than Objections.
- Agreements provide a lightweight primitive for managing shutdown.
  - Minimal overhead

Thank You