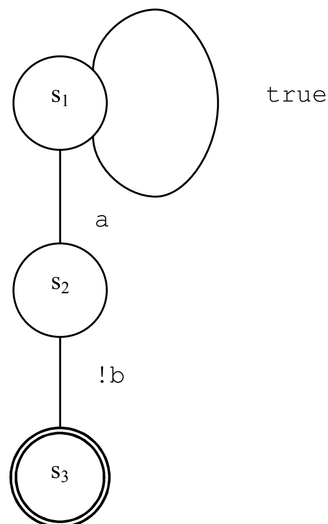
Rationale

The goal of the global clocking concept proposed here is to provide a definition of the primary clock synchronizing transitions in formal models.

DUT for formal verification purposes are usually considered as a set of states and transitions between the states. In RTL-level models verification the transitions are synchronous, i.e., at each step one and only one transition (possibly idle) is taken. This sequence of steps defines the primary clock in a natural way, s.t. all system transitions are synchronized by it. The primary clock is fair, i.e., it never stops ticking. In the metalanguage describing the formal semantics of SystemVerilog (see Annex F), the primary clock is denoted as @1, but this notation cannot be used in the language itself, since the event @1 ever happens. To be able to specify the primary clock in the language, a special notation is needed, and it is suggested to call it $global_clock in this proposal.

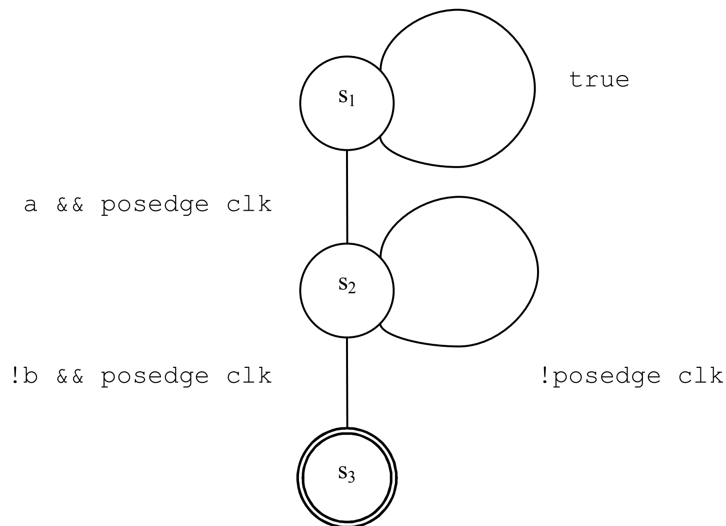
Formal verification of assertions controlled by primary clock is more efficient then the verification of assertions controlled by any other clock. Compare the failure automata generated for the following two assertions:

always p1: assert property (@$global_clock a |=> b);



and

always p2: assert property (@(posedge clk) a |=> b);



The first automaton is simpler: it does not have clocking conditions along its edges, and it does not have an extra idle edge for awaiting the clock. (Of course, these assertions are not equivalent, but often for practical needs either of them is acceptable.)

The notion of the primary clock allows specifying the next value of a variable – the value of the variable assigned at the end of a given transition: this value is set at the next tick of the primary clock $future_gclk(v) (this function is not part of this proposal and is referenced here for illustration purposes only). Using primary clock and next value functions is very natural for specifying stability properties. Consider the assertion saying that some signal sig should be always stable. The natural way to write this assertion is:

always assert property(@$global_clock sig == $ future_gclk(sig));

Without the notion of global clock this assertion has to be rewritten as:

always assert property(@clk ##1 sig == $past(sig));

which has the following drawbacks (I intentionally use $past and not $stable here to avoid a discussion about future stability functions):

1. Most users will write the above assertion as always assert property(@clk sig == $past(sig)); without skipping the initial phase, and thus will check that sig is always 0 (if sig has a two-value type), instead of the intended assertion.

2. The automaton of the second assertion is more expensive than that one of the first assertion since it contains more states.

3. The first assertion is more robust since it doesn't need to be changed when a faster clock is introduced in the design.

(Again, as in the previous example, these assertions are not equivalent, but often for practical needs either of them is acceptable.)

To be able to use primary clock in simulation, there is a need to associate a real clocking event with it. Therefore the global clocking construct is suggested. Nevertheless there is no obligation that all property clocks are synchronized with the global clock in simulation – this is a pure methodology or tool implementation issue.

Thus the introduction of global clocking and $global_clock constructs is backward compatible and introduces no penalty in simulation.

14.3 Clocking block declaration

REPLACE

clocking_declaration ::=
[**default**] **clocking** [clocking_identifier] clocking_event **;**
{ clocking_item }
endclocking [**:** clocking_identifier]

WITH

clocking_declaration ::=
[**default**] **clocking** [clocking_identifier] clocking_event **;**
{ clocking_item }
endclocking [**:** clocking_identifier]
| **global clocking** [clocking_identifier] clocking_event **; endclocking** [**:** clocking_identifier]

14.13 Global clocking

Note to editor: Shift the numeration of the following sections accordingly.

One clocking can be specified as the *global clocking* for the design.

The syntax for the global clocking specification statement is as follows:

clocking_declaration ::=
. . .
| **global clocking** [clocking_identifier] clocking_event ; **endclocking** [**:** clocking_identifier]

Only one global clocking can be specified anywhere in the design in any one compilation unit of the design (i.e., global clocking must have at most one definition on the entire model after elaboration). It is an error if there is more than one such specification of global clocking.

The system function `$global_clock` returns the event expression specified in the global clocking statement. The function has no arguments. It shall be a compiler error to invoke `$global_clock` function when no global clocking has been defined.

The main purpose of the global clocking is to specify which clocking events correspond to the primary clock used in formal verification.

Example: Declaring a global clocking

```
module top ;
	logic clk1, clk2;
	global clocking sys @(clk1 or clk2); endclocking
	// ...
```

```
endmodule
```

In this example a global clocking event sys is defined as a change of one of two clocks `clk1` and `clk2`. The global clocking name `sys` is optional, since it may be referred to as `$global_clock` from anywhere in the design.
The global clock does not need to be a real signal in the design, but rather an event.

16.4 Concurrent assertions overview

REPLACE

The clock expression that controls evaluation of a sequence can be more complex than just a single signal name. Expressions such as `(clk && gating_signal)` and `(clk` **iff** `gating_signal)` can be used to represent a gated clock. Other more complex expressions are possible. However, in order to ensure proper behavior of the system and conform as closely as possible to truly cycle-based semantics, the signals in a clock expression must be glitch-free and should only transition once at any simulation time.

WITH

The clock expression that controls evaluation of a sequence can be more complex than just a single signal name. Expressions such as `(clk && gating_signal)` and `(clk` **iff** `gating_signal)` can be used to represent a gated clock. Other more complex expressions are possible. However, in order to ensure proper behavior of the system and conform as closely as possible to truly cycle-based semantics, the signals in a clock expression must be glitch-free and should only transition once at any simulation time.

If an assertion is controlled by `$global_clock` (see Note to editor: insert a reference to *Global clocking* subclause here) then in simulation `$global_clock` is substituted by the clocking event defined in the **`global clocking`** statement. In formal verification the `$global_clock` is considered to be a primary system clock (see E.3.1). Thus, in the following example

```
global clocking @clk; endclocking

…

assert property(@$global_clock a);
```

the assertion states that `a` remains true at each tick of the global clock, which is interpreted in simulation as

```
assert property(@clk a);
```

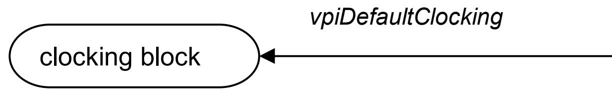
21.13.2 1800-2005 keywords

Note to editor: Insert a new keyword global into Table 21-13 (IEEE Std 1800-2005 reserved keywords).

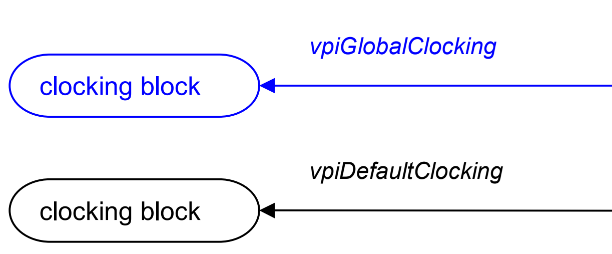
36.4 Module

Note to editor: in the diagram

REPLACE



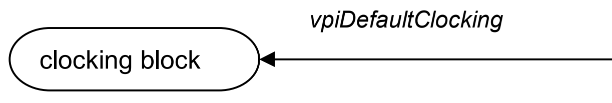
WITH



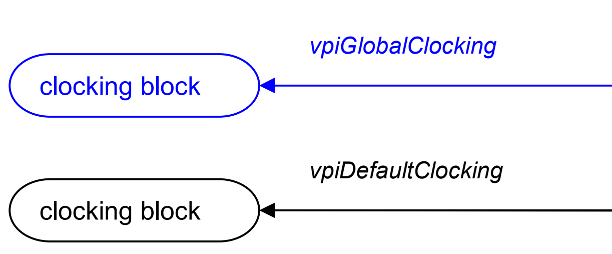
36.5 Interface

Note to editor: in the diagram

REPLACE



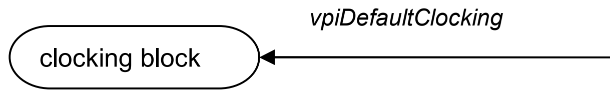
WITH



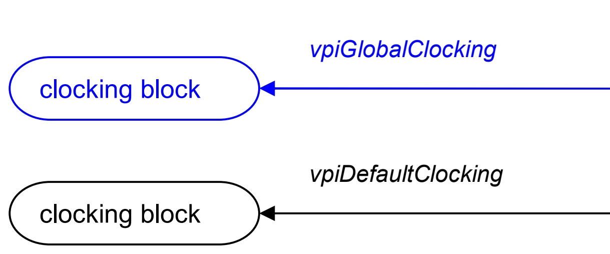
36.8 Program

Note to editor: in the diagram

REPLACE



WITH



A.6.11 Clocking block

REPLACE

clocking declaration ::=
[**default**] **clocking** [clocking identifier] clocking event **;**
{clocking item
endclocking [**:** clocking identifier]

WITH

clocking_declaration ::=
[**default**] **clocking** [clocking_identifier] clocking_event **;**
{ clocking_item }
endclocking [**:** clocking_identifier]
| **global clocking** [clocking_identifier] clocking_event **; endclocking** [**:** clocking_identifier]

Annex B

Keywords

Note to editor: Insert a new keyword global into Table B1 (Reserved keywords).

F.3.1 Rewrite rules for clocks

REPLACE

The semantics of clocked sequences and properties is defined in terms of the semantics of unclocked sequences and properties. The following rewrite rules define the transformation of a clocked sequence or property into an unclocked version that is equivalent for the purposes of defining the satisfaction relation. In this transformation, it is required that the conditions in event controls not be dependent upon any local variables.

— @(*c*) *b* (!*c* [*0:$] ##1 *c* & *b*) .
— @(*c*) (1, *v* = *e*) (@(*c*) 1 ##0 (1, *v* = *e*)) .
— @(*c*) (*P*) (@(*c*) *P*).
— @(*c*) (*R*1 ##1 *R*2) (@(*c*) *R*1 ##1 @(*c*) *R*2) .
— @(*c*) (*R*1 ##0 *R*2) (@(*c*) *R*1 ##0 @(*c*) *R*2) .
— @(*c*) (*R*1 **or** *R*2) (@(*c*) *R*1 **or** @(*c*) *R*2) .
— @(*c*) (*R*1 **intersect** *R*2) (@(*c*) *R*1 **intersect** @(*c*) *R*2) .
— @(*c*) **first_match** (*R*) **first_match** (@(*c*) *R*) .
— @(*c*) *R* [*0] (@(*c*) *R*) [*0] .
— @(*c*) *R* [*1:$] (@(*c*) *R*) [*1:$] .
— @(*c*) **disable iff** (*b*) *P* **disable iff** (*b*) @(*c*) *P* .
— @(*c*) **not** *P* **not** @(*c*) *P*.
— @(*c*) (*R* |-> *P*) (@(*c*) *R* |-> @(*c*) *P*) .
— @(*c*) (*P*1 **or** *P*2) (@(*c*) *P*1 **or** @(*c*) *P*2).
— @(*c*) (*P*1 **and** *P*2) (@(*c*) *P*1 **and** @(*c*) *P*2).

WITH

The semantics of clocked sequences and properties is defined in terms of the semantics of unclocked sequences and properties. The following rewrite rules define the transformation of a clocked sequence or property into an unclocked version that is equivalent for the purposes of defining the satisfaction relation. In this transformation, it is required that the conditions in event controls not be dependent upon any local variables.

— *@($global_clock) b (b) .*
— @(*c*) *b* (!*c* [*0:$] ##1 *c* & *b*) .
— @(*c*) (1, *v* = *e*) (@(*c*) 1 ##0 (1, *v* = *e*)) .
— @(*c*) (*P*) (@(*c*) *P*).
— @(*c*) (*R*1 ##1 *R*2) (@(*c*) *R*1 ##1 @(*c*) *R*2) .
— @(*c*) (*R*1 ##0 *R*2) (@(*c*) *R*1 ##0 @(*c*) *R*2) .
— @(*c*) (*R*1 **or** *R*2) (@(*c*) *R*1 **or** @(*c*) *R*2) .
— @(*c*) (*R*1 **intersect** *R*2) (@(*c*) *R*1 **intersect** @(*c*) *R*2) .
— @(*c*) **first_match** (*R*) **first_match** (@(*c*) *R*) .
— @(*c*) *R* [*0] (@(*c*) *R*) [*0] .
— @(*c*) *R* [*1:$] (@(*c*) *R*) [*1:$] .
— @(*c*) **disable iff** (*b*) *P* **disable iff** (*b*) @(*c*) *P* .
— @(*c*) **not** *P* **not** @(*c*) *P*.
— @(*c*) (*R* |-> *P*) (@(*c*) *R* |-> @(*c*) *P*) .
— @(*c*) (*P*1 **or** *P*2) (@(*c*) *P*1 **or** @(*c*) *P*2).
— @(*c*) (*P*1 **and** *P*2) (@(*c*) *P*1 **and** @(*c*) *P*2).

M.2 Source code

REPLACE

```
#define vpiInputSkew 706
#define vpiOutputSkew 707
#define vpiDefaultClocking 709
```

WITH

```
#define vpiInputSkew 706
#define vpiOutputSkew 707
#define vpiGlobalClocking editor to fill
#define vpiDefaultClocking 709
```