🕮 **Rule:** *[1] An empty sequence fused with a sequence is empty. An empty sequence is a sequence that uses the 0 as the repetition number, such as* `a[*0]`. For a sequence overlapping to match, the one cycle overlap must exist in both sequences. In other words, if either of the two sequences have an empty match, then there cannot be an overlap because one of the two sequences is of zero length. Thus, an empty sequence fused with any sequence results in no match.

🕮 **Rule:** Table 2.4.1 defines fusion rules with empty sequences. An empty sequence is denoted as *empty*, and a sequence is denoted as *seq*. Assume `start==1, b==1, c==1` at all cycles.

Table 2.4.1 Fusion Rules with Empty Sequences

Ch2/fusiont.sv, wave_fusiont.bmp, fusiont_assertion_report.txt

Sequence Fusion	Result	Example	Comments
`empty ##0 seq`	No match	`start -> b[*0] ##0 c;` // 💣FAIL	Empty fused with a sequence is a **no match,** assertion fails.

`empty ##n seq`
// n > 0
	Equivalent to
	`(##(n-1) seq )`
		`start |-> b[*0] ##1 c;`
		---------------------- 	
		`start |-> b[*0] ##2 c;`
			`start |-> ##0 c;` *// same as*
			`start |-> c;` *// assertion passes*
			-------------------------------- 	
			`start |-> ##1 c;` *// same as*
			// assertion passes

`seq ##0 empty`	No match	`start |-> c ##0 b[*0];` // 💣 FAIL	sequence fused with empty is a **no match**, assertion fails

`seq ##n empty`
// n > 0
	Equivalent to
	`(seq ##(n-1) `true)`
		`start |-> c ##1 b[*0];`
		------------------------ 	
		`start |-> c ##2 b[*0];`
			`start |-> c ##0'true;` *// same as*
			`start |-> c;` *// assertion passes*
			-------------------------------- 	
			`start |-> c ##1 1 ##1 b[*0]` *// same as*
			`start |-> c ##1 1;`
			// assertion passes one cycle after c==1

Guideline: Carefully evaluate the meaning of the fusion (##0) operator with other sequences. It could represent unwanted or unclear properties. The empty sequence is most commonly found in the definition of ranges. For example, the following property statement `(req |=> rdy[*0:1] ##[0:1] ack)` states that if `req==1` then at the next cycle `rdy` need not occur if `ack` occurs. However, if `rdy` occurs then `ack` can occur in that cycle or in the next cycle. This property expression can be analyzed as follows:

req **|=>** rdy[*0:1] ##[0:1] ack;	original property
req **|=>** (rdy[*0] ##[0:1] ack) `or` (rdy[*1] ##[0:1] ack);	Equivalency
req **|=>** ((rdy[*0] ##0 ack) **or** (rdy[*0] ##1 ack)) `or` (rdy[*1] ##0 ack `or` rdy[*1] ##1 ack);	Equivalency
((rdy[*0] ##0 ack) or rdy[*0] ##1 ack) `or` rdy ##0 ack `or` rdy ##1 ack;	Equivalency
<--No match-> <--##0 ack ----> <- fusion -> <2 cycles seq ->	
req **|=>** (ack) `or` rdy ##0 ack `or` rdy ##1 ack;	Equivalency
req **|=>** ack `or` rdy && ack `or` rdy ##1 ack	
req **|=>** ack `or` (rdy ##1 ack);	Final reduction

Other examples where the application of fusion can be misguided include the following:

- **GOTO: a |-> a[->1];**

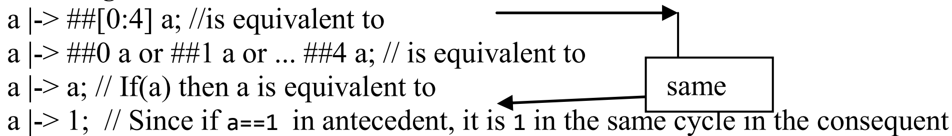
The following DOES NOT express the notion that "if `a`, then another `a` at some time in the future:

```
a |-> a[->1];   // is equivalent to
a |-> !a[*0:$] ##1 a;   // same as
a |-> (!a[*0] ##1 a) or (!a[*1] ##1 a) or .. (!a[*n] ##1 a);
However, (!a[*0] ##1 a) is equivalent to "a".   Thus,
a |-> a;    // Final equivalence, which is not the intended property
```

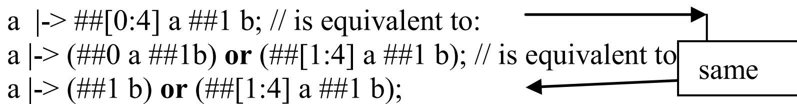
Note: To express the notion that "if `a`, then another `a` at some time in the future

```
a |=> strong(a[->1]);  // OK
a |-> strong(##[1:$] a);
```

- **Range**

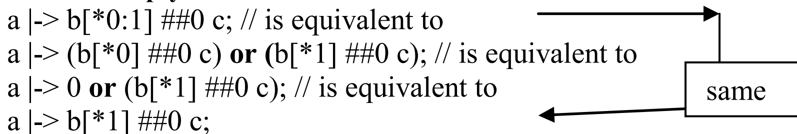


a |-> ##[0:4] a; //is equivalent to
a |-> ##0 a or ##1 a or ... ##4 a; // is equivalent to
a |-> a; // If(a) then a is equivalent to
a |-> 1; // Since if `a==1` in antecedent, it is `1` in the same cycle in the consequent



a |-> ##[0:4] a ##1 b; // is equivalent to:
a |-> (##0 a ##1b) **or** (##[1:4] a ##1 b); // is equivalent to
a |-> (##1 b) **or** (##[1:4] a ##1 b);

- **with empty match:**



a |-> b[*0:1] ##0 c; // is equivalent to
a |-> (b[*0] ##0 c) **or** (b[*1] ##0 c); // is equivalent to
a |-> 0 **or** (b[*1] ##0 c); // is equivalent to
a |-> b[*1] ##0 c;

2.4.2 Sequence disjunction (or)

🕮 <u>**Rule**</u>: The sequence **`or`** operator constructs a sequence in which one of two alternative sequences need to succeed for the disjunction of two sequences to succeed. <u>Each of two sequences starts at the first clocking event of their respective sequence</u>. Both sequences <u>need not end at the same cycle</u>. When Oring two sequences, the result is a match if one of the sequences is a match. If both sequences match, then the one that matched the soonest causes the Oring to match. If both sequences do not match, then the ORing of the sequences is not a match. If the sequence is used as a property, then the no-match is a failure of the property. The sequence `(a ##1 b) or (c ##1 d)` states that either sequence `a b` or sequence `c d` would satisfy the composite sequence. When one of the sequences matches, the whole composite sequence (i.e., *`(a ##1 b) or (c ##1 d)`*) is considered ended.

In a multiclocked operation where each sequence has its own separate clock, the two operands of the **`or`** sequences start at the first clocking event of their respective sequence.[26] For example, in the following property

```
(@(posedge clk) s0 |=> (@(posedge clk1) s1) or (@(posedge clk2) s2);)
```

sequence `s1` starts at the first clocking event of `clk1`, and sequence `s2` starts at the first clocking event of `clk2`.

[26] That rule also applies to the **`and`**, **`or`**, **`until`**, **`implies`**, **`iff`** operators. See 4.4 on multiclocked operations.