Section 5
Functional Modeling

This chapter specifies the functional modeling for synthesis, formal verification, and simulation.

5.1 Combinational functions

This section defines the different types of combinational functions in ALF.

5.1.1 Combinational logic

Combinational logic can be described by continuous assignments of boolean values (*True* or *False*) to output variables as a function of boolean values of input variables. Such functions can be expressed in either boolean expression format or statetable format.

Let us consider an arbitrary continuous assignment

$z = f(a_1 .,.. a_n)$

In a dynamic or simulation context, the left-hand side (LHS) variable *z* is evaluated whenever there is a change in one of the right-hand side (RHS) variables a_i. No storage of previous states is needed for dynamic simulation of combinational logic.

5.1.2 Boolean operators on scalars

Table 5-1, Table 5-2, and Table 5-3 list unary, binary, and ternary boolean operators on scalars.

Table 5-1 : Unary boolean operators

Operator	Description
!, ~	logical inversion

Table 5-2 : Binary boolean operators

Operator	Description
&&, &	logical AND
||, |	logical OR
~^	logic equivalence (XNOR)
^	logic anti valence (XOR)

Table 5-3 : Ternary operator

Operator	Description
?	boolean condition operator for construction of combinational if-then-else clause
:	boolean else operator for construction of combinational if-then-else clause

Combinational if-then-else clauses are constructed as follows:

```
<cond1>? <value1>: <cond2>? <value2>: <cond3>? <value3>: <default_value>
```

If `cond1` evaluates to boolean *True*, then `value1` is the result; else if `cond2` evaluates to boolean *True*, then `value2` is the result; else if `cond3` evaluates to boolean *True*, then `value3` is the result; else `default_value` is the result of this clause.

5.1.3 Boolean operators on words

Table 5-4 and Table 5-5 list unary and binary reduction operators on words (logic variables with one or more bits). The result of an expression using these operators shall be a logic value.

Table 5-4 : Unary reduction operators

Operator	Description
&	AND all bits
~&	NAND all bits
|	OR all bits
~|	NOR all bits
^	XOR all bits
~^	XNOR all bits

Table 5-5 : Binary reduction operators

Operator	Description
==	equality for case comparison
!=	non-equality for case comparison
>	greater
<	smaller
>=	greater or equal
<=	smaller or equal

Table 5-6 and Table 5-7 list unary and binary bitwise operators. The result of an expression using these operators shall be an array of bits.

Table 5-6 : Unary bitwise operators

Operator	Description
~	bitwise inversion

Table 5-7 : Binary bitwise operators

Operator	Description
&	bitwise AND
|	bitwise OR
^	bitwise XOR
~^	bitwise XNOR

The following arithmetic operators, listed in Table 5-8, are also defined for boolean operations on words. The result of an expression using these operators shall be an extended array of bits.

Table 5-8 : Binary operators

Operator	Description
<<	shift left
>>	shift right
+	addition
-	subtraction
*	multiplication
/	division
%	modulo division

The arithmetic operations addition, subtraction, multiplication, and division shall be *unsigned* if all the operands have the datatype *unsigned*. If any of the operands have the datatype signed, the operation shall be *signed*. See Table 6-23 for the `DATATYPE` definitions.

5.1.4 Operator priorities

The priority of binding operators to operands in boolean expressions shall be from strongest to weakest in the following order:

1. unary boolean operator (`!`, `~`, `&`, `~&`, `|`, `~|`, `^`, `~^`)
2. `XNOR` (`~^`), `XOR` (`^`), relational (`>`, `<`, `>=`, `<=`, `==`, `!=`), shift (`<<`, `>>`)
3. `AND` (`&`, `&&`), `NAND` (`~&`), multiply (`*`), divide (`/`), modulus (`%`)

4. `OR` (`|, ||`), `NOR` (`~|`), add (+), subtract (-)

5. ternary operators (`?, :`)

5.1.5 Datatype mapping

Logical operations can be applied to scalars and words. For that purpose, the values of the operands are reduced to a system of three logic values in the following way:

`H` has the logic value `1`
`L` has the logic value `0`
`W, Z, U` have the logic value `X`
A word has the logic value `1`, if the unary OR reduction of all bits results in `1`
A word has the logic value `0`, if the unary OR reduction of all bits results in `0`
A word has the logic value `X`, if the unary OR reduction of all bits results in `X`

Case comparison operations can also be applied to scalars and words. For scalars, they are defined in Table 5-9.

Table 5-9 : Case comparison operators

A	B	A==B	A!=B	A>B	A<B
1	1	1	0	0	0
1	H	0	1	X	X
1	0	0	1	1	0
1	L	0	1	1	0
1	W, U, Z, X	0	1	X	0
H	1	0	1	X	X
H	H	1	0	0	0
H	0	0	1	1	0
H	L	0	1	1	0
H	W, U, Z, X	0	1	X	0
0	1	0	1	0	1
0	H	0	1	0	1
0	0	1	0	0	0
0	L	0	1	X	X
0	W, U, Z, X	0	1	0	X
L	1	0	1	0	1
L	H	0	1	0	1
L	0	0	1	X	X
L	L	1	0	0	0
L	W, U, Z, X	0	1	0	X
X	X	1	0	X	X
X	U	X	X	X	X
X	0, 1, H, L, W, Z	0	1	X	X

Table 5-9 : Case comparison operators, *continued*

A	B	A==B	A!=B	A>B	A<B
W	W	1	0	X	X
W	U	X	X	X	X
W	0, 1, H, L, X, Z	0	1	X	X
Z	Z	1	0	X	X
Z	U	X	X	X	X
Z	0, 1, H, L, X, W	0	1	X	X
U	0, 1, H, L, X,W, Z, U	X	X	X	X

For word operands, the operations > and < are performed after reducing all bits to the 3-value system first and then interpreting the resulting number according to the datatype of the operands. For example, if datatype is *signed*, `'b1111` is smaller than `'b0000`; if datatype is *unsigned*, `'b1111` is greater than `'b0000`. If two operands have the same value `'b1111` and a different datatype, the unsigned `'b1111` is greater than the signed `'b1111`.

The operations >= and <= are defined in the following way:

```
(a >= b) === (a > b) || (a == b)
(a <= b) === (a < b) || (a == b)
```

5.1.6 Rules for combinational functions

If a boolean expression evaluates *True*, the assigned output value is `1`. If a boolean expression evaluates *False*, the assigned output value is `0`. If the value of a boolean expression cannot be determined, the assigned output value is `X`. Assignment of values other than `1`, `0`, or `X` needs to be specified explicitly.

For evaluation of the boolean expression, input value `'bH` shall be treated as `'b1`. Input value `'bL` shall be treated as `'b0`. All other input values shall be treated as `'bX`.

Examples:

In equation form, these rules can be expressed as follows.

```
BEHAVIOR {
      Z = A;
}
```

is equivalent to

```
BEHAVIOR {
      Z = A ? ’b1 : ’b0;
}
```

More explicitly, this is also equivalent to

```
BEHAVIOR {
      Z = (A==’b1 || A==’bH)? ’b1 : (A==’b0 || A==’bL)? ’b0 : ’bX;
}
```

In table form, this can be expressed as follows:

```
STATETABLE {
      A     :     Z;
      ?     :     (A);
}
```

which is equivalent to

```
STATETABLE {
      A     :     Z;
      0     :     0;
      1     :     1;
}
```

More explicitly, this is also equivalent to

```
STATETABLE {
      A     :     Z;
      0     :     0;
      L     :     0;
      1     :     1;
      H     :     1;
      X     :     X;
      W     :     X;
      Z     :     X;
      U     :     X;
}
```

5.1.7 Concurrency in combinational functions

Multiple boolean assignments in combinational functions are understood to be concurrent. The order in the functional description does not matter, as each boolean assignment describes a piece of a logic circuit. This is illustrated in Figure 5-1.

```
BEHAVIOR {
      Q1 = <1st_boolean_expression(D1..Di)> ;
      ...
      Qn = <nth_boolean_expression(D1..Di)> ;
}
```

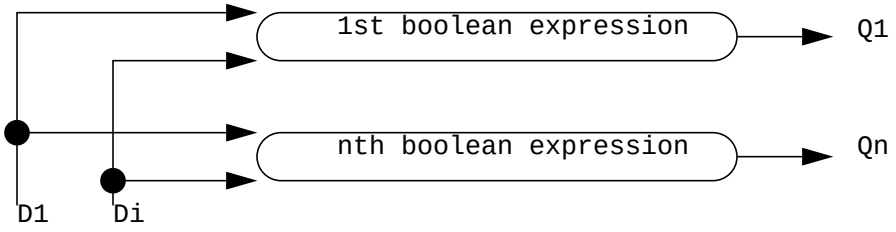

Figure 5-1: Concurrency for combinational logic

5.2 Sequential functions

This section defines the different types of sequential functions in ALF.

5.2.1 Level-sensitive sequential logic

In sequential logic, an output variable z_j can also be a function of itself, i.e., of its previous state. The sequential assignment has the form

$$z_j = f(a_1 ..,.. a_n , z_1 ..,.. z_m)$$

The RHS cannot be evaluated continuously, since a change in the LHS as a result of a RHS evaluation shall trigger a new RHS evaluation repeatedly, unless the variables attain stable values. Modeling capabilities of sequential logic with continuous assignments are restricted to systems with oscillating or self-stabilizing behavior.

However, using the concept of *triggering conditions* for the LHS enables everything which is necessary for modeling *level-sensitive* sequential logic. The expression of a triggered assignment can look like this:

$$@\ g(b_1 ..,.. b_k)\ z_j = f(a_1 ..,.. a_n , z_1 ..,.. z_m)$$

The evaluation of *f* is activated whenever the *triggering function* g is *True*. The evaluation of g is self-triggered, i.e. at each time when an argument of g changes its value. If *g* is a boolean expression like *f*, we can model all types of *level-sensitive sequential logic*.

During the time when *g* is *True*, the logic cell behaves exactly like combinational logic. During the time when *g* is *False*, the logic cell holds its value. Hence, one memory element per state bit is needed.

5.2.2 Edge-sensitive sequential logic

In order to model *edge-sensitive sequential logic*, notations for logical transitions and logical states are needed.

If the triggering function *g* is sensitive to logical transitions rather than to logical states, the function *g* evaluates to *True* only for an infinitely small time, exactly at the moment when the transition happens. The sole purpose of *g* is to trigger an assignment to the output variable through evaluation of the function *f* exactly at this time.

Edge-sensitive logic requires storage of the previous output state and the input state (to detect a transition). In fact, all implementations of edge-triggered flip-flops require at least two storage elements. For instance, the most popular flip-flop architecture features a master latch driving a slave latch.

Using transitions in the triggering function for value assignment, the functionality of a positive edge triggered flip-flop can be described as follows in ALF:

```
@ (01 CP) {Q = D;}
```

which reads “at rising edge of `CP`, assign `Q` the value of `D`”.

If the flip-flop also has an asynchronous direct clear pin (`CD`), the functional description consists of either two concurrent statements or two statements ordered by priority, as shown in Figure 5-2.

```
// concurrent style
@ (!CD) {Q = 0;}
@ (01 CP && CD) {Q = D;}
// priority (if-then-else) style
@ (!CD) {Q = 0;} : (01 CP) {Q = D;}
```

Figure 5-2: Model of a flip-flop with asynchronous clear in ALF

The following two examples show corresponding simulation models in Verilog and VHDL.

```
// full simulation model
always @(negedge CD or posedge CP) begin
   if ( ! CD ) Q <= 0;
   else if (CP && !CP_last_value) Q <= D;
   else Q <= 1'bx;
end
always @ (posedge CP or negedge CP) begin
   if (CP===0 | CP===1'bx) CP_last_value <= CP ;
end
// simplified simulation model for synthesis
always @(negedge CD or posedge CP) begin
   if ( ! CD ) Q <= 0;
   else Q <= D;
end
```

Figure 5-3: Model of a flip-flop with asynchronous clear in Verilog

```
// full simulation model

process (CP, CD) begin
   if (CD = '0') then
      Q <= '0';
   elsif (CP'last_value = '0' and CP = '1' and CP'event) then
      Q <= D;
   elsif (CP'last_value = '0' and CP = 'X' and CP'event) then
      Q <= ’X’;
   elsif (CP'last_value = 'X' and CP = '1' and CP'event) then
      Q <= ’X’;
   end if;
end process;

// simplified simulation model for synthesis

process (CP, CD) begin
   if (CD = '0') then
      Q <= '0';
   elsif (CP = '1' and CP'event) then
      Q <= D;
   end if;
end process;
```

Figure 5-4: Model of a flip-flop with asynchronous clear in VHDL

The following differences in modeling style can be noticed: VHDL and Verilog provide the list of sensitive signals at the beginning of the `process` or `always` block, respectively. The information of level-or edge-sensitivity shall be inferred by `if-then-else` statements inside the block. ALF shows the level-or-edge sensitivity as well as the priority directly in the triggering expression. Verilog has another particularity: The sensitivity list indicates whether at least one of the triggering signals is edge-sensitive by the use of `negedge` or `posedge`. However, it does not indicate which one, since either none or all signals shall have `negedge` or `posedge` qualifiers.

Furthermore, `posedge` is any transition with `0` as initial state *or* `1` as final state. A positive-edge triggered flip-flop shall be inferred for synthesis, yet this flip-flop shall only work correctly if both the initial state is `0` *and* the final state is `1`. Therefore, a simulation model for verification needs to be more complex than the model in the synthesizeable RTL code.

In Verilog, the extra non-synthesizeable code needs to also reproduce the relevant previous state of the clock signal, whereas VHDL has built-in support for `last_value` of a signal.

5.2.3 Unary operators for vector expressions

A transition operation is defined using unary operators on a scalar net. The scalar constants (see Figure 11-6) shall be used to indicate the start and end states of a transition on a scalar net.

```
bit bit          // apply transition from bit value to bit value
```

For example,

`01` is a transition from `0` to `1`.

No whitespace shall be allowed between the two scalar constants. The transition operators shown in Table 5-10 shall be considered legal.

Table 5-10 : Unary vector operators on bits

Operator	Description
01	signal toggles from `0` to `1`
10	signal toggles from `1` to `0`
00	signal remains `0`
11	signal remains `1`
0?	signal remains `0` or toggles from `0` to arbitrary value
1?	signal remains `1` or toggles from `1` to arbitrary value
?0	signal remains `0` or toggles from arbitrary value to `0`
?1	signal remains `1` or toggles from arbitrary value to `1`
??	signal remains constant or toggles between arbitrary values
0*	a number of arbitrary signal transitions, including possibility of constant value, with the initial value `0`
1*	a number of arbitrary signal transitions, including possibility of constant value, with the initial value `1`
?*	a number of arbitrary signal transitions, including possibility of constant value, with arbitrary initial value
***0**	a number of arbitrary signal transitions, including possibility of constant value, with the final value `0`
***1**	a number of arbitrary signal transitions, including possibility of constant value, with the final value `1`
***?**	a number of arbitrary signal transitions, including possibility of constant value, with arbitrary final value

Unary operators for transitions can also appear in the `STATETABLE`.

Transition operators are also defined on words (and can appear the in `STATETABLE` as well):

`'`*`base word`* `'`*`base word`*

In this context, the transition operator shall apply transition from first word value to second word value.

For example,

`'hA'h5` is a transition of a 4-bit signal from `'b1010` to `'b0101`.

No whitespace shall be allowed between *base* and *word*.

The unary and binary operators for transition, listed in Table 5-11 and Table 5-12 respectively, are defined on bits and words.

Table 5-11 : Unary vector operators on bits or words

Operator	Description
?-	no transition occurs
??	apply arbitrary transition, including possibility of constant value
?!	apply arbitrary transition, excluding possibility of constant value
?~	apply arbitrary transition with all bits toggling

5.2.4 Basic rules for sequential functions

A sequential function is described in equation form by a boolean assignment with a condition specified by a boolean expression or a vector expression. If the condition evaluates to `1` (*True*), the boolean assignment is activated and the assigned output values follows the rules for combinational functions. If the vector expression evaluates to `0` (*False*), the output variables hold their assigned value from the previous evaluation.

For evaluation of a condition, the value `'bH` shall be treated as *True*, the value `'bL` shall be treated as *False*. All other values shall be treated as the unknown value `'bX`.

Example:

The following behavior statement

```
BEHAVIOR {
      @ (E) {Z = A;}
}
```

is equivalent to

```
BEHAVIOR {
      @ (E=='b1 || E=='bH) {Z = A;}
}
```

The following statetable statement, describing the same logic function

```
STATETABLE {
      E     A     :     Z;

      0     ?     :     (Z);
      1     ?     :     (A);
}
```

is equivalent to

```
STATETABLE {
      E     A     :     Z;

      0     ?     :     (Z);
      L     ?     :     (Z);
      1     ?     :     (A);
      H     ?     :     (A);
}
```

For edge-sensitive and higher-order event sensitive functions, transitions from or to `'bL` shall be treated like transitions from or to `'b0`, and transitions from or to `'bH` shall be treated like transitions from or to `'b1`.

Not every transition can trigger the evaluation of a function. The set of vectors triggering the evaluation of a function are called *active vectors*. From the set of active vectors, a set of *inactive vectors* can be derived, which shall clearly not trigger the evaluation of a function. There are is also a set of ambiguous vectors, which can trigger the evaluation of the function.

The set of active vectors is the set of vectors for which both observed states before and after the transition are known to be logically equivalent to the corresponding states defined in the vector expression.

The set of inactive vectors is the set of vectors for which at least one of the observed states before or after the transition is known to be not logically equivalent to the corresponding states defined in the vector expression.

Example:

For the following sequential function

```
@ (01 CP) { Z = A; }
```

the active vectors are

```
('b0'b1 CP)
('b0'bH CP)
('bL'b1 CP)
('bL'bH CP)
```

and the inactive vectors are

```
(’b1’b0 CP)
(’b1’bL CP)
(’b1’bX CP)
(’b1’bW CP)
(’b1’bZ CP)
(’bH’b0 CP)
(’bH’bL CP)
(’bH’bX CP)
(’bH’bW CP)
(’bH’bZ CP)
(’bX’b0 CP)
(’bX’bL CP)
(’bW’b0 CP)
(’bW’bL CP)
(’bZ’b0 CP)
(’bZ’bL CP)
(’bU’b0 CP)
(’bU’bL CP)
```

and the ambiguous vectors are

```
('b0'bX CP)
('b0'bW CP)
('b0'bZ CP)
('bL'bX CP)
('bL'bW CP)
('bL'bZ CP)
('bX'b1 CP)
('bW'b1 CP)
('bZ'b1 CP)
('bX'bH CP)
('bW'bH CP)
('bZ'bH CP)
('bX'bW CP)
('bX'bZ CP)
('bW'bX CP)
('bW'bZ CP)
('bZ'bX CP)
('bZ'bW CP)
('bU'bX CP)
('bU'bW CP)
('bU'bZ CP)
```

For vectors using exclusively based literals, the set of active vectors is the vector itself, the set of inactive vectors is any vector with at least one different literal, and the set of ambiguous vectors is empty.

Therefore, ALF does not provide a default behavior for ambiguous vectors, since the behavior for each vector can be explicitly defined in vectors using based literals.

5.2.5 Concurrency in sequential functions

The principle of concurrency applies also for edge-sensitive sequential functions, where the triggering condition is described by a vector expression rather than a boolean expression. In edge-sensitive logic, the target logic variable for the boolean assignment (LHS) can also be an operand of the boolean expression defining the assigned value (RHS). Concurrency implies that the RHS expressions are evaluated immediately *before* the triggering edge, and the values are assigned to the LHS variables immediately *after* the triggering edge. This is illustrated in Figure 5-5.

```
BEHAVIOR {
      @ ( <vector_expression(E1..Em)> ) {
            Q1 =
<1st_boolean_expression(D1..Di)> ;
            ...
            Qn =
<nth_boolean_expression(D1..Di)> ; } }
```

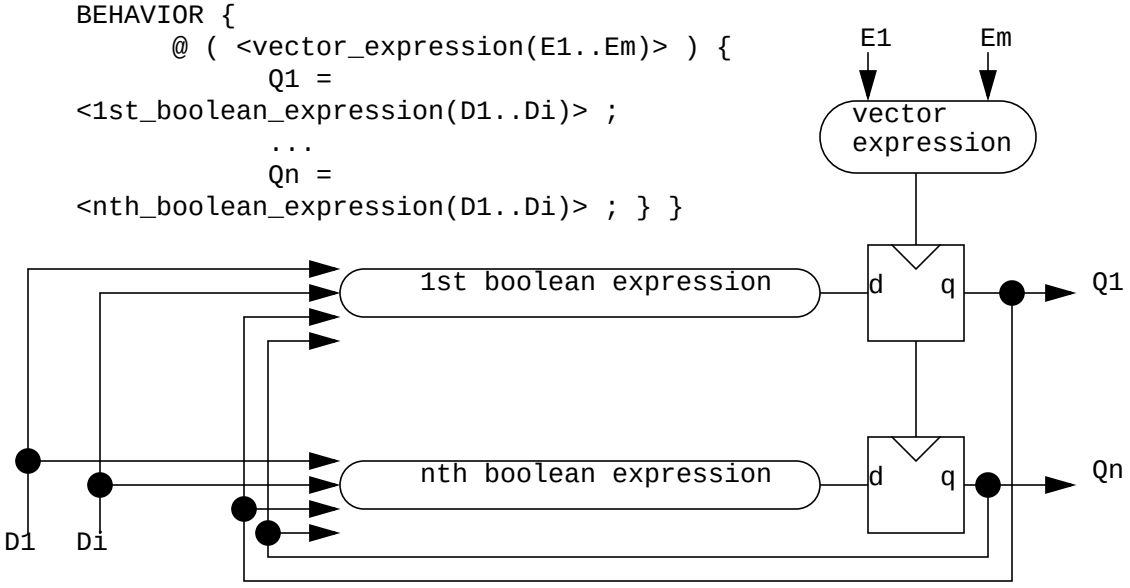

Figure 5-5: Concurrency for edge-sensitive sequential logic

Statements with multiple concurrent conditions for boolean assignments can also be used in sequential logic. In that case conflicting values can be assigned to the same logic variable. A default conflict resolution is not provided for the following reasons:

- Conflict resolution might not be necessary, since the conflicting situation is prohibited by specification.
- For different types of analysis (e.g., logic simulation), a different conflict resolution behavior might be desirable, while the physical behavior of the circuit shall not change. For instance, pessimistic conflict resolution always assigns X, more accurate conflict resolution first checks whether the values are conflicting. Different choices can be motivated by a trade-off in analysis accuracy and runtime.
- If complete library control over analysis is desired, conflict resolution can be specified explicitly.

Example:

```
BEHAVIOR {
      @ ( <condition_1> ) { Q = <value_1>; }
      @ ( <condition_2> ) { Q = <value_2>; }
}
```

Explicit pessimistic conflict resolution can be described as follows:

```
BEHAVIOR {
      @ ( <condition_1> && <condition_2>  ) { Q = 'bX; }
      @ ( <condition_1> && ! <condition_2>) { Q = <value_1>; }
      @ ( <condition_2> && ! <condition_1>) { Q = <value_2>; }
}
```

Explicit accurate conflict resolution can be described as follows:

```
BEHAVIOR {
      @ ( <condition_1> && <condition_2>  ) {
            Q = (<value_1>==<value_2>)? <value_1> : 'bX;
      }
      @ ( <condition_1> && ! <condition_2>) { Q = <value_1>; }
      @ ( <condition_2> && ! <condition_1>) { Q = <value_2>; }
}
```

Since the conditions are now rendered mutually exclusive, equivalent descriptions with priority statements can be used. They are more elegant than descriptions with concurrent statements.

```
BEHAVIOR {
      @ ( <condition_1> && <condition_2>  ) {
            Q = <conflict_resolution_value>;
      }
      : ( <condition_1> ) { Q = <value_1>; }
      : ( <condition_2> ) { Q = <value_2>; }
}
```

Given the various explicit description possibilities, the standard does not prescribe a default behavior. The model developer has the freedom of incomplete specification.

5.2.6 Initial values for logic variables

Per definition, all logic variables in a behavioral description have the initial value `U` which means “uninitialized”. This value cannot be assigned to a logic variable, yet it can be used in a behavioral description in order to assign other values than `U` after initialization.

Example:

```
BEHAVIOR {
      @ ( Q1 == 'bU ) { Q1 = 'b1 ; }
      @ ( Q2 == 'bU ) { Q2 = 'b0 ; }
      // followed by the rest of the behavioral description
}
```

A template can be used to make the intent more obvious, for example:

```
TEMPLATE VALUE_AFTER_INITIALIZATION {
      @ ( <logic_variable> == 'bU ) { <logic_variable> = <initial_value>
; }
}
BEHAVIOR {
      VALUE_AFTER_INITIALIZATION ( Q1 'b1' )
      VALUE_AFTER_INITIALIZATION ( Q2 'b0' )
      // followed by the rest of the behavioral description
}
```

Logic variables in a vector expression shall be declared as `PIN`s. It is possible to annotate initial values directly to a pin. Such variables shall never take the value `U`. Therefore vector expressions involving `U` for such variables (see the previous example) are meaningless.

Example:

```
PIN Q1 { INITIAL_VALUE = 'b1 ; }
PIN Q2 { INITIAL_VALUE = 'b0 ; }
```

5.3 Higher-order sequential functions

This section defines the different types of higher-order sequential functions in ALF.

5.3.1 Vector-sensitive sequential logic

Vector expressions can be used to model generalized higher order sequential logic; they are an extension of the boolean expressions. A *vector expression* describes sequences of logical events or transitions in addition to static logical states. A vector expression represents a description of a logical stimulus without timescale. It describes the order of occurrence of events.

The -> operator *(followed by*) gives a general capability of describing a sequence of events or a vector. For example, consider the following vector expression:

```
01 A -> 01 B
```

which reads “rising edge on `A` is followed by rising edge on `B`”.

A vector expression is evaluated by an event sequence detection function. Like a single event or a transition, this function evaluates *True* only at an infinitely short time when the event sequence is detected, as shown in Figure 5-6.

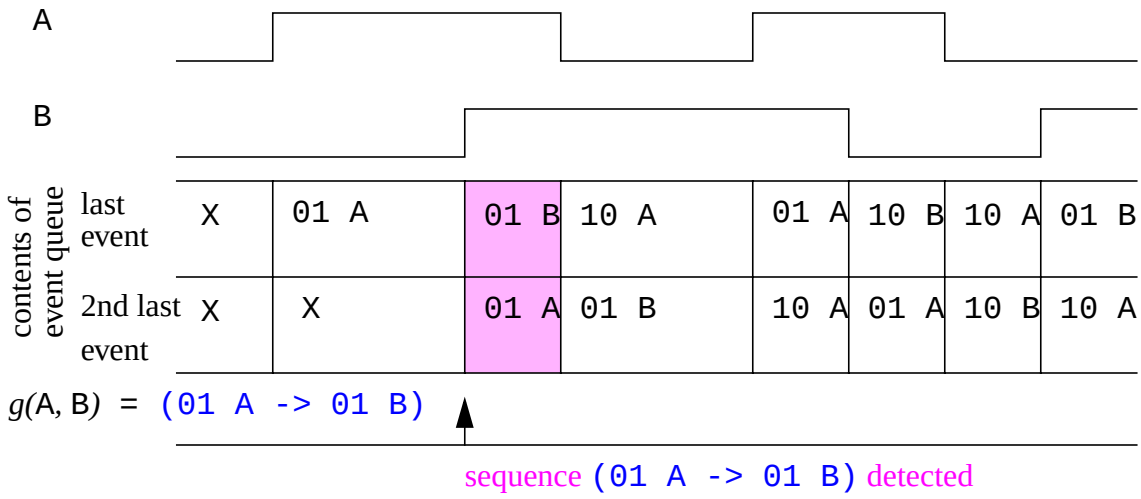

Figure 5-6: Example of event sequence detection function

The event sequence detection mechanism can be described as a queue that sorts events according to their order of arrival. The event sequence detection function evaluates *True* at exactly the time when a new event enters the queue and forms the required sequence, i.e., *the sequence specified by the vector expression* with its preceding events.

A vector-sensitive sequential logic can be called *(N+1) order sequential logic*, where *N* is the number of events to be stored in the queue. The implementation of *(N+1)* order sequential logic requires *N* memory elements for the event queue and one memory element for the output itself.

A sequence of events can also be gated with static logical conditions. In the example,

```
(01 CP -> 10 CP) && CD
```

the pin `CD` shall have `state 1` from some time before the rising edge at `CP` to some time after the falling edge of `CP`. The pin `CD` can not go low (`state 0`) after the rising edge of `CP` and go high again before the falling edge of `CP` because this would insert events into the queue and the sequence “rising edge on `CP` followed by falling edge on `CP`” would not be detected.

The formal calculation rules for general vector expressions featuring both states and transitions are detailed in Section 5.3.2 and Section 5.3.3.

The concept of vector expression supports functional modeling of devices featuring digital communication protocols with arbitrary complexity.

5.3.2 Canonical binary operators for vector expressions

The following canonical binary operators are necessary to define sequences of transitions:

- `vector_followed_by` for completely specified sequence of events
- `vector_and` for simultaneous events
- `vector_or` for alternative events
- `vector_followed_by` for incompletely specified sequence of events

The symbols for the boolean operators for `AND` and `OR` are overloaded for `vector_and` and `vector_or`, respectively. The new symbols for the `vector_followed_by` operators are shown in Table 5-12.

Table 5-12 : Canonical binary vector operators

Operator	Operands	LHS, RHS commutative	Description
`->`	2 vector expressions	no	Left-hand side (LHS) transition *is followed by* Right-hand side (RHS) transition, no transition can occur in-between
`&&, &`	2 vector expressions	yes	LHS *and* RHS transition *occur simultaneously*
`||, |`	2 vector expressions	yes	LHS *or* RHS transition *occur alternatively*
`~>`	2 vector expressions	no	Left-hand side (LHS) transition *is followed by* Right-hand side (RHS) transition, other transitions can occur in-between

Per definition, the `-> and ~>` operators shall not be commutative, whereas the `&&` and `||` operators on events shall be commutative.

```
01 a && 01 b === 01 b && 01 a
01 a || 01 b === 01 b || 01 a
```

The -> and ~> operators shall be freely associative.

```
01 a -> 01 b -> 01 c === (01 a -> 01 b) -> 01 c === 01 a -> (01 b -> 01 c)
01 a ~> 01 b ~> 01 c === (01 a ~> 01 b) ~> 01 c === 01 a ~> (01 b ~> 01 c)
```

The && operator is defined for single events and for event sequences with the same number of -> operators each.

```
(01 A1 .. -> ... 01 AN) & (01 B1 .. -> ... 01 BN)
===
01 A1 & 01 B1 ... -> ... 01 AN & 01 BN
```

The || operator reduces the set of edge operators (unary vector operators) to canonical and non-canonical operators.

```
(?? a)       === (?! a)||(?- a)  //a does or does not change its value
```

Hence ?? is non-canonical, since it can be defined by other operators.

If <value1><value2> is an edge operator consisting of two based literals value1 and value2 and word is an expression which can take the value value1 or value2, then the following vector expressions are considered equivalent:

```
<value1><value2> <word>
      ===         10 (<word> == <value1>) && 01 (<word> == <value2>)
      ===         01 (<word> != <value1>) && 01 (<word> == <value2>)
      ===         10 (<word> == <value1>) && 10 (<word> != <value2>)
      ===         01 (<word> != <value1>) && 10 (<word> != <value2>)
// all expressions describe the same event:
// <word> makes a transition from <value1> to <value2>
```

Hence vector expressions with edge operators using based literals can be reduced to vector expressions using only the edge operators 01 and 10.

5.3.3 Complex binary operators for vector expressions

Table 5-13 defines the complex binary operators for vector operators.

Table 5-13 : Complex binary vector operators

Operator	Operands	LHS, RHS commutative	Description
<->	2 vector expressions	yes	LHS transition follows or is followed by RHS transition
&>	2 vector expressions	no	LHS transition *is followed by or occurs simultaneously* with RHS transition
<&>	2 vector expressions	yes	LHS transition *follows or is followed by or occurs simultaneously* with RHS transition

The following expressions shall be considered equivalent:

```
(01 a <-> 01 b) === (01 a -> 01 b)||(01 b -> 01 a)
(01 a  &> 01 b) === (01 a -> 01 b)||(01 a && 01 b)
(01 a <&> 01 b) === (01 a -> 01 b)||(01 b -> 01 a)||(01 a && 01 b)
```

By their symmetric definition, the <-> and <&> operators are commutative.

```
01 a <-> 01 b === 01 b <-> 01 a
01 a <&> 01 b === 01 b <&> 01 a
```

The commutative complex binary vector operators are defined in Table 5-12. The commutativity rules are only defined for two operands:

- commutative “followed by”:

```
vect_expr1 <-> vect_expr2 ===
   vect_expr1 -> vect_expr2 // vect_expr1 occurs first
|  vect_expr2 -> vect_expr1 // vect_expr2 occurs first
```

- commutative “followed by or simultaneously occurring”:

```
vect_expr1 <&> vect_expr2 ===
   vect_expr1 -> vect_expr2 // vect_expr1 occurs first
|  vect_expr2 -> vect_expr1 // vect_expr2 occurs first
|  vect_expr1 && vect_expr2 // both occur simultaneously
```

5.3.3.1 Extension to N operands

This section defines how to use *N* operands.

A `complex_vector_expression` of the form

```
vector_expression { <-> vector_expression }
```

shall be commutative for all operands. The `complex_vector_expression` describes alternative event sequences in which the temporal order of each constituent `vector_expression` is completely permutable, excluding simultaneous occurrence of each constituent `vector_expression`.

A `complex_vector_expression` of the form

```
vector_expression { <&> vector_expression }
```

shall be commutative for all operands. The `complex_vector_expression` describes alternative event sequences in which the temporal order of each constituent `vector_expression` is completely permutable, including simultaneous occurrence of each constituent `vector_expression`.

Example:

```
01 A <-> 01 B <-> 01 C ===
   01 A -> 01 B -> 01 C
|  01 B -> 01 C -> 01 A
|  01 C -> 01 A -> 01 B
|  01 C -> 01 B -> 01 A
|  01 B -> 01 A -> 01 C
|  01 A -> 01 C -> 01 B
```

```
01 A <&> 01 B <&> 01 C ===
   01 A -> 01 B -> 01 C
|  01 B -> 01 C -> 01 A
|  01 C -> 01 A -> 01 B
|  01 C -> 01 B -> 01 A
|  01 B -> 01 A -> 01 C
|  01 A -> 01 C -> 01 B
|  01 A && 01 B -> 01 C
|  01 A -> 01 B && 01 C
|  01 B && 01 C -> 01 A
|  01 B -> 01 C && 01 A
|  01 C && 01 A -> 01 B
|  01 C -> 01 A && 01 B
|  01 A && 01 B && 01 C
```

5.3.3.2 Boolean rules

The following rule applies for a boolean `AND` operation with three operands:

```
rule 1:
   A & B & C === (A & B) & C | A & (B & C)
```

A corresponding rule also applies to the commutative followed-by operation with three operands:

```
rule 2:
   01 A <-> 01 B <-> 01 C ===
      (01 A <-> 01 B) <-> 01 C
   |  01 A <-> (01 B <-> 01 C)
```

The alternative boolean expressions `(A & B) & C` and `A & (B & C)` in rule 1 are equivalent. Therefore, rule 1 can be reduced to the following:

```
rule 3:
   A & B & C === (A & B) & C === (B & C) & A
```

A corresponding rule does *not* apply to complex vector operands, since each expression with associated operands generates only a subset of permutations:

```
(01 A <-> 01 B) <-> 01 C ===
      ((01 A <-> 01 B) -> 01 C)
|     (01 C -> (01 A <-> 01 B)) ===
      01 A -> 01 B -> 01 C
|     01 B -> 01 A -> 01 C
|     01 C -> 01 A -> 01 B
|     01 C -> 01 B -> 01 A
```

The permutations

```
01 A -> 01 C -> 01 B
01 B -> 01 C -> 01 A
```

are missing.

```
01 A <-> (01 B <-> 01 C) ===
   (01 A -> (01 B <-> 01 C))
|  ((01 B <-> 01 C) -> 01 A) ===
   01 A -> 01 B -> 01 C
|  01 A -> 01 C -> 01 B
|  01 B -> 01 C -> 01 A
|  01 C -> 01 B -> 01 A
```

The permutations

```
|  01 B -> 01 A -> 01 C
|  01 C -> 01 A -> 01 B
```

are missing.

5.3.4 Operators for conditional vector expressions

The definitions of the `&&`, `?`, and `:` operators are also overloaded to describe a *conditional vector expression* (involving boolean expressions and vector expressions), as shown in Table 5-14. The clauses are boolean expressions; while vector expressions are subject to those clauses.

Table 5-14 : Operators for conditional vector expressions

Operator	Operands	LHS, RHS commutative	Description
`&&, &`	1 vector expression, 1 boolean expression	yes	boolean expression (LHS or RHS) is *True* while sequence of transitions, defined by vector expression (RHS or LHS) occurs
`?`	1 vector expression, 1 boolean expression	no	boolean condition operator for construction of if-then-else clause involving vector expressions
`:`	1 vector expression, 1 boolean expression	no	boolean else operator for construction of if-then-else clause involving vector expressions

An example for conditional vector expression using `&&` is given below:

```
(01 a && !b)                    // a rises while b==0
```

The order of the operands in a conditional vector expression using `&&` shall not matter.

```
<vector_exp> && <boolean_exp> === <boolean_exp> && <vector_exp>
```

The `&&` operator is still commutative in this case, although one operand is a boolean expression defining a static state, the other operand is a vector expression defining an event or a sequence of events. However, since the operands are distinguishable per se, it is not necessary to impose a particular order of the operands.

An example for conditional vector expression using `?` and `:` is given below.

```
!b ? 01 a : c ? 10 b : 01 d
===
!b & 01 a | !(!b) & c & 10 b | !(!b) & !c & 01 d
```

This example shows how a conditional vector expression using ternary operators can be expressed with alternative conditional vector expressions.

A conditional vector expression can be reduced to a non-conditional vector expression in some cases (see Section 5.4.11).

Every binary vector operator can be applied to a conditional vector expression.

5.3.5 Operators for sequential logic

Table 5-15 defines the complex binary operators for vector operators.

Table 5-15 : Operators for sequential logic

Operator	Description
`@`	sequential `if` operator, followed by a boolean logic expression (for level-sensitive assignment) or by a vector expression (for edge-sensitive assignment)
`:`	sequential `else if` operator, followed by a boolean logic expression (for level-sensitive assignment) or by a vector expression (for edge-sensitive assignment) with lower priority

Sequential assignments are constructed as follows:

```
@ ( <trigger1> ) { <action1> } : ( <trigger2> ) { <action2> } :
  ( <trigger3> ) { <action3> }
```

If `trigger1` event is detected, then `action1` is performed; else if `trigger2` event is detected, then `action2` is performed; else if `trigger3` event is detected, then `action3` is performed as a result of this clause.

5.3.6 Operator priorities

The priority of binding operators to operands in non-conditional vector expressions shall be from strongest to weakest in the following order:

1. unary vector operators (edge literals)
2. complex binary vector operators (`<->`, `&>`, `<&>`)
3. vector `AND` (`&`, `&&`)
4. vector_followed_by operators (`->`, `~>`)
5. vector `OR` (`|`, `||`)

5.3.7 Using PINs in VECTORs

A VECTOR defines state, transition, or sequence of transitions of pins that are controllable and observable for characterization.

Within a CELL, the set of PINs with SCOPE=behavior or SCOPE=measure or SCOPE=both is the default set of variables in the event queue for vector expressions relevant for BEHAVIOR or VECTOR statements or both, respectively.

For detection of a sequence of transitions it is necessary to observe the set of variables in the event queue. For instance, if the set of pins consists of A, B, C, D, the vector expression

```
(01 A -> 01 B)
```

implies no transition on A, B, C, D occurs between the transitions 01 A and 01 B.

The default set of pins applies only for vector expressions without conditions. The conditional event AND operator limits the set of variables in the event queue. In this case, only the state of the condition and the variables appearing in the vector expression are observed.

Example:

```
(01 A -> 01 B) && (C | D)
```

No transition on A, B occurs between 01 A and 01 B, and (C | D) needs to stay *True* in-between 01 A and 01 B as well. However, C and D can change their values as long as (C | D) is satisfied.

5.4 Modeling with vector expressions

Vector expressions provide a formal language to describe digital waveforms. This capability can be used for functional specification, for timing and power characterization, and for timing and power analysis.

In particular, vector expressions add value by addressing the following modeling issues:

- *Functional specification*: complex sequential functionality, e.g., bus protocols.
- *Timing analysis*: complex timing arcs and timing constraints involving more than two signals.
- *Power analysis*: temporal and spatial correlation between events relevant for power consumption.
- *Circuit characterization and test*: specification of characterization and/or test vectors for particular timing, power, fault, or other measurements within a circuit.

Like boolean expressions, vector expressions provide the means for describing the functionality of digital circuits in various contexts without being self-sufficient. Vector expressions enrich this functional description capability by adding a “dynamic” dimension to the otherwise “static” boolean expressions.

The following subsections explain the semantics of vector expressions step-by-step. The vector expression concept is explained using terminology from simulation event reports. However, the application of vector expressions is not restricted to post-processing event reports.

Some application tools (e.g., power analysis tools) can actually evaluate vector expressions during post-processing of event reports from simulation. Other application tools, especially

simulation model generators, need to respect the causality between the triggering events and the actions to be triggered. While it is semantically impossible to describe cause and effect in the same vector expression for the purpose of functional modeling, both cause and effect can appear in a vector expression used for a timing arc description.

ALF does not make assumption about the physical nature of the event report. Vector expressions can be applied to an actual event report written in a file, to an internal event queue within a simulator, or to a hypothetical event report which is merely a mathematical concept.

5.4.1 Event reports

This section describes the terminology of event reports from simulation, which is used to explain the concept of ALF vector expressions. The intent of ALF vector expressions is not to *replace* existing event report formats. Non-pertinent details of event report formats are not described here.

Simulation events (e.g., from Verilog or VHDL) can be reported in a value change dump (VCD) file, which has the following general form:

```
<time1>
      <variableA> <stateU>
      <variableB> <stateV>
      ...
<time2>
      <variableC> <stateW>
      <variableD> <stateX>
      ...
<time3> ...
```

The set of variables for which simulation events are reported, i.e., the *scope* of the event report needs to be defined beforehand. Each variable also has a definition for the *set of states* it can take. For instance, there can be binary variables, 16-bit integer variables, 1-bit variables with drive-strength information, etc. Furthermore, the initial state of each variable shall be defined as well. In an ALF context, the terms *signal* and *variable* are used interchangeably. In VHDL, the corresponding term is *signal*. In Verilog, there is no single corresponding term. All `input`, `output`, `wire`, and `reg` variables in Verilog correspond to a `signal` in VHDL.

The time values `<time1>`, `<time2>`, `<time3>`, etc. shall be in increasing order. The order in which simultaneous events are reported does not matter. The number of time points and the number of simultaneous events at a certain time point are unlimited.

In the physical world, each event or change of state of a variable takes a certain amount of time. A variable cannot change its state more than once at a given point in time. However, in simulation, this time can be smaller than the resolution of the time scale or even zero (`0`). Therefore, a variable can change its state more than once at a given point in simulation time. Those events are, strictly speaking, not simultaneous. They occur in a certain order, separated by an infinitely small delta-time. Multiple simultaneous events of the same variable are not reported in the VCD. Only the final state of each variable is reported.

A VCD file is the most compact format that allows reconstruction of entire waveforms for a given set of variables. A more verbose form is the test pattern format.

```
<TIME>  <variableA> <variableB> <variableC> <variableD>
<time1> <stateU>    <stateV>    ...         ...
<time2> <stateU>    <stateV>    <stateW>    <stateX>
<time3> ...         ...         ...         ...
```

The test pattern format reports the state of each variable at every point in time, regardless of whether the state has changed or not. Previous and following states are immediately available in the previous and next row, respectively. This makes the test pattern format more readable than the VCD and well-suited for taking a snapshot of events in a time window.

An example of an event report in VCD format:

```
// initial values
A 0   B 1   C 1   D X   E 1
// event dump
109   A 1   D 0
258   B 0
573   C 0
586   A 0
643   A 1
788   A 0   B 1   C 1
915   A 1
1062  E 0
1395  B 0   C 0
1640  A 0   D 1
// end of event dump
```

An example of an event report in test pattern format:

```
time  A     B     C     D     E
0     0     1     1     X     1
109   1     1     1     0     1
258   1     0     1     0     1
573   1     0     0     0     1
586   0     0     0     0     1
643   1     0     0     0     1
788   0     1     1     0     1
915   1     1     1     0     1
1062  1     1     1     0     0
1395  1     0     0     0     0
1640  0     0     0     1     0
```

Both VCD and test pattern formats represent the same amount of information and can be translated into each other.

5.4.2 Event sequences

For specification of a functional waveform (e.g., the write cycle of a memory), it is not practical to use an event report format, such as a VCD or test pattern format. In such waveforms, there is no absolute time. And the relative time, for example, the setup time between address change and write enable change, can vary from one instance to the other.

The main purpose of vector_expressions is waveform specification capability. The following operators can be used:

- vector_unary (also called *edge operator* or *unary vector operator*)

  The edge operator is a prefix to a variable in a vector expression. It contains a pair of states, the first being the previous state, the second being the new state. Edge operators can describe a change of state or no change of state.
- vector_and (also called *simultaneous event operator*)

  This operator uses the overloaded symbol & or && interchangeably. The & operator is the separator between simultaneously occurring events
- vector_followed_by (also called *followed-by operator*)

  The "immediately followed-by operator" using the symbol -> is treated first. The -> operator is the separator between consecutively occurring events.

These operators are necessary and sufficient to describe the following subset of vector_expressions:

- vector_single_event

  A change of state in a single variable, for example:

  ```
  01 A
  ```
- vector_event

  A simultaneous change of state in one or more variables, for example:

  ```
  01 A & 10 B
  ```
- vector_event_sequence

  Subsequently occurring changes of state in one or more variables, for example:

  ```
  01 A & 10 B -> 10 A
  ```

The vector_and operator has a higher binding priority than the vector_followed_by operator.

We can now express the pattern of the sample event report in a vector_event_sequence expression:

```
01 A & X0 D -> 10 B -> 10 C -> 10 A -> 01 A
-> 10 A & 01 B & 01 C -> 01 A -> 10 E -> 10 B & 10 C -> 10 A & 01 D
```

We can define the *length* of a vector_event_sequence expression as the number of subsequent events described in the vector_event_sequence expression. The length is equal to the number of -> operators plus one (1).

Although the vector expression format contains an inherent redundancy, since the old state of each variable is always the same as the new state of the same variable in a previous event, it is more human-readable, especially for waveform description. On the other hand, it is more compact than the test pattern format. For short event sequences, it is even more compact than the VCD, since it eliminates the declaration of initial values. To be accurate, for variables with exactly one event the vector expression is more compact than the VCD. For variables with more than one event the VCD is more compact than the vector expression. In summary, the vector expression format offers readability similar to the test pattern format and compactness close to the VCD format.

5.4.3 Scope and content of event sequences

The *scope* applicable to a vector expression defines the set of variables in the event report. The *content* of a vector expression is the set of variables that appear in the vector expression itself. The content of a vector expression shall be a subset of variables within scope.

- `PIN`s with the annotation `SCOPE = BEHAVIOR` are applicable variables for vector expressions within the context of `BEHAVIOR`.
- `PIN`s with the annotation `SCOPE = MEASURE` are applicable variables for vector expressions within the context of `VECTOR`.
- `PIN`s with the annotation `SCOPE = BOTH` are applicable variables for all vector expressions.

A `vector_event_sequence` expression is an event pattern without time, containing only the variables within its own content. This event pattern is evaluated against the event report containing all variables within scope. The vector expression is *True* when the event pattern matches the event report.

Example:

```
time  A     B     C     D     E     // scope is A, B, C, D, E
0     0     1     1     X     1
109   1     1     1     0     1
258   1     0     1     0     1
573   1     0     0     0     1
586   0     0     0     0     1
643   1     0     0     0     1
788   0     1     1     0     1
915   1     1     1     0     1
1062  1     1     1     0     0
1395  1     0     0     0     0
1640  0     0     0     1     0
```

Consider the following vector expressions in the context of the sample event report:

```
01 A                                          //(1)  content is A

//event pattern expressed by (1):
//    A
//    0
//    1
```

`(1)` is *True* at time 109, time 643, and time 915.

```
10 B -> 10 C                                  //(2)  content is B, C

//event pattern expressed by (2):
//    B     C
//    1     1
//    0     1
//    0     0
```

`(2)` is *True* at time 573.

```
10 A -> 01 A                                        //(3)  content is A

//event pattern expressed by (3):
//    A
//    1
//    0
//    1
```

`(3)` is *True* at time 643 and time 915.

```
01 D                                                //(4)  content is D

//event pattern expressed by (4):
//    D
//    0
//    1
```

`(4)` is *True* at time 1640.

```
01 A -> 10 C                                        //(5)  content is A, C

//event pattern expressed by (5):
//    A     C
//    0     1
//    1     1
//    1     0
```

`(5)` is not be *True* at any time, since the event pattern expressed by `(5)` does not match the event report at any time.

5.4.4 Alternative event sequences

The following operator can be used to describe alternative events:

`vector_or`, also called *event-or operator* or *alternative-event operator*, using the overloaded symbol `|` or `||` interchangeably. The `|` operator is the separator between alternative events or alternative event sequences.

In analogy to boolean operators, `|` has a lower binding priority than `&` and `->`. Parentheses can be used to change the binding priority.

Example:

```
(01 A -> 01 B) | 10 C === 01 A -> 01 B | 10 C
01 A -> (01 B | 10 C) === 01 A -> 01 B | 01 A -> 10 C
```

Consider the following vector expressions in the context of the sample event report:

```
01 A | 10 C                                         //(6)

//event pattern expressed by (6):
//    A
//    0
//    1

//alternative event pattern expressed by (6):
//    C
//    1
//    0
```

`(6)` is *True* at time 109, time 573, time 643, time 915, and time 1395.

```
10 B -> 10 C | 10 A -> 01 A                    //(7)

//event pattern expressed by (7):
//    B     C
//    1     1
//    0     1
//    0     0

//alternative event pattern expressed by (7):
//    A
//    1
//    0
//    1
```

`(7)` is *True* at time 573, time 643, and time 915.

```
01 D | 10 B -> 10 C                                //(8)

//event pattern expressed by (8):
//    D
//    0
//    1

//alternative event pattern expressed by (8):
//    B     C
//    1     1
//    0     1
//    0     0
```

`(8)` is *True* at time 573 and time 1640.

```
10 B -> 10 C | 10 A                                //(9)

//event pattern expressed by (9):
//    B     C
//    1     1
//    0     1
//    0     0

//alternative event pattern expressed by (9):
//    A
//    1
//    0
```

`(9)` is *True* at time 573, time 586, time 788, and time 1640.

The following operators provide a more compact description of certain alternative event sequences:

- `&>` events occur simultaneously or follow each other in the order RHS after LHS
- `<->` a LHS event followed by a RHS event or a RHS event followed by a LHS event
- `<&>` events occur simultaneously or follow each other in arbitrary order

Example:

```
01 A &> 01 C      ===   01 A & 01 C | 01 A -> 01 C
01 A <-> 01 C     ===   01 A -> 01 C | 01 C -> 01 A
01 A <&> 01 C     ===   01 A <-> 01 C | 01 A & 01 C
```

The binding priority of these operators is higher than of `&` and `->`.

5.4.5 Symbolic edge operators

Alternative events of the same variable can be described in a even more compact way through the use of edge operators with symbolic states. The symbol `?` stands for “any state”.

- edge operator with `?` as the previous state:
  transition from any state to the defined new state
- edge operator with `?` as the next state:
  transition from the defined previous state to any state.

Both edge operators include the possibility no transition occurred at all, i.e., the previous and the next state are the same. This situation can be explicitly described with the following operator:

edge operator with next state = previous state, also called *non-event operator*
The operand stays in the state defined by the operator.

The following symbolic edge operators also can be used:

- `?-` no transition on the operand
- `?!` transition from any state to any state different from the previous state
- `??` transition from any state to any state or no transition on the operand
- `?~` transition from any state to its bitwise complementary state

Example: Let `A` be a logic variable with the possible states `1`, `0`, and `X`.

```
?0 A === 00 A | 10 A | X0 A
?1 A === 01 A | 11 A | X1 A
?X A === 0X A | 1X A | XX A
0? A === 00 A | 01 A | 0X A
1? A === 10 A | 11 A | 1X A
X? A === X0 A | X1 A | XX A
?! A === 01 A | 0X A | 10 A | 1X A | X0 A | X1 A
?~ A === 01 A | 10 A | XX A
?? A === 00 A | 01 A | 0X A | 10 A | 11 A | 1X A | X0 A | X1 A | XX A
?- A === 00 A | 11 A | XX A
```

For variables with more possible states (e.g., logic states with different drive strength and multiple bits) the explicit description of alternative events is quite verbose. Therefore the symbolic edge operators are useful for a more compact description.

This completes the set of `vector_binary` operators necessary for the description of a subset of `vector_expressions` called `vector_complex_event` expressions. All `vector_binary` operators have two `vector_complex_event` expressions as operands. The set of `vector_event_sequence` expressions is a subset of `vector_complex_event` expressions. Every `vector_complex_event` expression can be expressed in terms of alternative `vector_event_sequence` expressions. The latter could be called *minterms*, in analogy to boolean algebra.

5.4.6 Non-events

A `vector_single_event` expression involving a non-event operator is called a *non-event*. A rigorous definition is required for `vector_complex_event` expressions containing non-events. Consider the following example of a flip-flop with clock input `CLK` and data output `Q`.

```
01 CLK -> 01 Q    // (i)
01 CLK -> 00 Q    // (ii)
```

The vector expression `(i)` describes the situation where the output switches from `0` to `1` after the rising edge of the clock. The vector expression `(ii)` describes the situation where the output remains at `0` after the rising edge of the clock.

How is it possible to decide whether `(i)` or `(ii)` is *True*, without knowing the delay between `CLK` and `Q`? The only way is to wait until any event occurs after the rising edge of `CLK`. If the event is not on `Q` and the state of `Q` is `0` during that event, then `(ii)` is *True*.

Hence, a non-event is *True* every time when another event happens and the state of the variable involved in the non-event satisfies the edge operator of the non-event.

Example:

```
time  A     B     C     D     E
0     0     1     1     X     1
109   1     1     1     0     1
258   1     0     1     0     1
573   1     0     0     0     1
586   0     0     0     0     1
643   1     0     0     0     1
788   0     1     1     0     1
915   1     1     1     0     1
1062  1     1     1     0     0
1395  1     0     0     0     0
1640  0     0     0     1     0
```

The test pattern format represents an event, for example `01 A`, in no different way than a non-event, for example `11 E`. This non-event is *True* at times 109, 258, 573, 586, 643, 788, and 915; in short, every time when an event happens while `E` is constant `1`.

5.4.7 Compact and verbose event sequences

A `vector_event_sequence` expression in a compact form can be transformed into a verbose form by padding up every `vector_event` expression with non-events. The next state of each variable within a `vector_event` expression shall be equal to the previous state of the same variable in the subsequent `vector_event` expression.

Example:

```
01 A -> 10B === 01 A & 11 B -> 11 A & 10 B
```

A vector expression for a complete event report in compact form resembles the VCD, whereas the verbose form looks like the test pattern.

```
// compact form
01 A & X0 D -> 10 B -> 10 C -> 10 A -> 01 A
-> 10 A & 01 B & 01 C -> 01 A -> 10 E
-> 10 B & 10 C -> 10 A & 01 D
===
// verbose form
?0 A & ?1 B & ?1 C & ?X D & ?1 E->
01 A & 11 B & 11 C & X0 D & 11 E->
11 A & 10 B & 11 C & 00 D & 11 E->
11 A & 00 B & 10 C & 00 D & 11 E->
10 A & 00 B & 00 C & 00 D & 11 E->
01 A & 00 B & 00 C & 00 D & 11 E->
10 A & 01 B & 01 C & 00 D & 11 E->
01 A & 11 B & 11 C & 00 D & 11 E->
11 A & 11 B & 11 C & 00 D & 10 E->
11 A & 10 B & 10 C & 00 D & 00 E->
10 A & 00 B & 00 C & 01 D & 00 E
```

The transformation rule needs to be slightly modified in case the compact form contains a `vector_event` expression consisting only of non-events. By definition, the non-event is *True* only if a real event happens simultaneously with the non-event. Padding up a `vector_event` expression consisting of non-events with other non-events make this impossible. Rather, this `vector_event` expression needs to be padded up with unspecified events, using the `??` operator. Eventually, unspecified events can be further transformed into partly specified events, if a former or future state of the involved variable is known.

Example:

```
01 A -> 00 B
=== 01 A & 00 B -> ?? A & 00 B
```

In the first transformation step, the unspecified event `?? A` is introduced.

```
01 A & 00 B -> ?? A & 00 B
=== 01 A & 00 B -> 1? A & 00 B
```

In the second step, this event becomes partly specified. `?? A` is bound to be `1? A` due to the previous event on `A`.

5.4.8 Unspecified simultaneous events within scope

Variables which are within the scope of the vector expression yet do not appear in the vector expression, can be used to pad up the vector expression with unspecified events as well. This is equivalent to omitting them from the vector expression.

Example:

```
01 A -> 10 B // let us assume a scope containing A, B, C, D, E
===
01 A & 10 B & ?? C & ?? D & ?? E -> 11 A & 10 B & ?? C & ?? D & ?? E
```

This definition allows unspecified events to occur *simultaneously* with specified events or specified non-events. However, it disallows unspecified events to occur *in-between* specified events or specified non-events.

At first sight, this distinction seems to be arbitrary. Why not disallow unspecified events altogether? Yet there are several reasons why this definition is practical.

If a vector expression disallows simultaneously occurring unspecified events, the application tool has the burden not only to match the pattern of specified events with the event report but also to check whether the other variables remain constant. Therefore, it is better to specify this extra pattern matching constraint explicitly in the vector expression by using the `?-` operator.

There are many cases where it actually does not matter whether simultaneously occurring unspecified events are allowed or disallowed:

- *Case 1*: Simultaneous events are impossible by design of the flip-flop. For instance, in a flip-flop it is impossible for a triggering clock edge `01 CK` and a switch of the data output `? Q` to occur at the same time. Therefore, such events can not appear in the event report. It makes no difference whether `01 CK & ?- Q`, `01 CK & ?? Q`, or `01 CK` is specified. The only occurring event pattern is `01 CK & ?- Q` and this pattern can be reliably detected by specifying `01 CK`.
- *Case 2*: Simultaneous events are prohibited by design. For instance, in a flip-flop with a positive setup time and positive hold time, the triggering clock edge `01 CK` and a switch of the data input `?! D` is a timing violation. A timing checker tool needs the violating pattern specified explicitly, i.e., `01 CK & ?! D`. In this context, it makes sense to specify the non-violating pattern also explicitly, i.e., `01 CK & ?- D`. The pattern `01 CK` by itself is not applicable.
- *Case 3*: Simultaneous events do not occur in correct design. For instance, power analysis of the event `01 CK` needs no specification of `?! D` or `?- D`. In the analysis of an event report with timing violations, the power analysis is less accurate anyway. In the analysis of the event report for the design without timing violation, the only occurring event pattern is `01 CK & ?- D` and this pattern can be reliably detected by specifying `01 CK`.[1]
- *Case 4*: The effects of simultaneous events are not modeled accurately. This is the case in static timing analysis and also to some degree in dynamic timing simulation. For instance, a `NAND` gate can have the inputs `A` and `B` and the output `Z`. The event sequence exercising the timing arc `01 A -> 10 Z` can only happen if `B` is constant `1`. No event on `B` can happen in-between `01 A` and `10 Z`. Likewise, the timing arc `01 B -> 10 Z` can only happen if `A` is constant `1` and no event happens in-between `01 B` and `10 Z`. The timing arc with simultaneously switching inputs is commonly ignored. A tool encountering the scenario `01 A & 01 B -> 10 Z` has no choice other than treating it arbitrarily as `01 A -> 10 Z` or as `01 B -> 10 Z`.
- *Case 5*: The effects of simultaneous events are modeled accurately. Here it makes sense to specify all scenarios explicitly, e.g., `01 A & ?- B -> 10 Z`, `01 A &?! B -> 10 Z`, `?- A`

1. The power analysis tool relates to a timing constraint checker in a similar way as a parasitic extraction tool relates to a DRC tool. If the layout has DRC violations, for instance shorts between nets, the parasitic extraction tool shall report inaccurate wire capacitance for those nets. After final layout, the DRC violations shall be gone and the wire capacitance shall be accurate.

`& 01 B -> 10 Z`, etc., whereas the patterns `01 A -> 10 Z` and `01 B -> 10 Z` by themselves apply only for less accurate analysis (see *Case 4*).

There is also a formal argument why unspecified events on a vector expression need to be allowed rather than disallowed. Consider the following vector expressions within the scope of two variables A and B.

```
01 A              // (i)
01 B              // (ii)
01 A & 01 B       // (iii)
```

The natural interpretation here is `(iii)` === `(i)` & `(ii)`. This interpretation is only possible by allowing simultaneously occurring unspecified events.

Allowing simultaneously occurring unspecified events, the vector expressions `(i)` and `(ii)`, respectively, are interpreted as follows:

```
01 A & ?? B       // (i')
?? A & 01 B       // (ii')
```

Disallowing simultaneously occurring unspecified events, the vector expressions `(i)` and `(ii)`, respectively, are interpreted as follows:

```
01 A & ?- B       // (i'')
?- A & 01 B       // (ii'')
```

The vector expressions `(i')` and `(ii')` are compatible with `(iii)`, whereas `(i'')` and `(ii'')` are not.

5.4.9 Simultaneous event sequences

The semantic meaning of the "simultaneous event operator" can be extended to describe simultaneously occurring *event sequences*, by using the following definition:

```
(01 A#1 .. -> ... 01 A#N) & (01 B#1 .. -> ... 01 B#N)
=== 01 A#1 & 01 B#1 ... -> ... 01 A#N & 01 B#N
```

This definition is analogous to scalar multiplication of vectors with the same number of indices. The number of indices corresponds to the number of `vector_event` expressions separated by `->` operators. If the number of `->` in both vector expressions is not the same, the shorter vector expression can be left-extended with unspecified events, using the `??` operator, in order to align both vector expressions.

Example:

```
(01 A -> 01 B -> 01 C) & (01 D -> 01 E)
=== (01 A -> 01 B -> 01 C) & (?? D -> 01 D -> 01 E)
=== 01 A & ?? D -> 01 B & 01 D -> 01 C & 01 E
=== 01 A -> 01 B & 01 D -> 01 C & 01 E
```

The easiest way to understand the meaning of “simultaneous event sequences” is to consider the event report in test pattern format. If each `vector_event_sequence` expression matches the event report in the same time window, then the event sequences happen simultaneously.

```
time  A     B     C     D     E
0     0     1     1     X     1
109   1     1     1     0     1
258   1     0     1     0     1
573   1     0     0     0     1
586   0     0     0     0     1
643   1     0     0     0     1
788   0     1     1     0     1
915   1     1     1     0     1
1062  1     1     1     0     0
1395  1     0     0     0     0
1640  0     0     0     1     0
```

Example:

```
01 A -> 10 B === 01 A & 11 B -> 11 A & 10 B     // (10a)

// event pattern expressed by (10a):
//    A     B
//    0     1
//    1     1
//    1     0

X0 D -> 00 D                                    // (10b)

// event pattern expressed by (10b):
//    D
//    X
//    0
//    0

(01 A -> 10 B) & (X0 D -> 00 D)                 // (10) === (10a)&(10b)
```

Both `(10a)` and `(10b)` are *True* at time 258. Therefore `(10)` is *True* at time 258.

```
10 C
=== ?? C -> ?? C -> 10 C
=== ?? C -> ?1 C -> 10 C                        // (11a)

// event pattern expressed by (11a):
//    C
//    ?
//    ?
//    1
//    0
```

`(11a)` is left-extended to match the length of the following `(11b)`.

```
01 A -> 00 D -> 11 E ===
   01 A & 00 D & ?? E
-> ?? A & 00 D & ?? E
-> ?? A & ?? D & 11 E
===
   01 A & 00 D & ?? E
```

```
-> 1? A & 00 D & ?1 E
-> ?? A & 0? D & 11 E                              // (11b)

// event pattern expressed by (11b):
//    A     D     E
//    0     0     ?
//    1     0     ?
//    ?     0     1
//    ?     ?     1
```

`(11b)` contains explicitly specified non-events. The non-event `00 D` calls for the unspecified events `?? A` and `?? E`. The non-event `00 E` calls for the unspecified events `?? A` and `?? D`. By propagating well-specified previous and next states to subsequent events, some unspecified events become partly specified.

```
10 C & (01 A -> 00 D -> 11 E)                      // (11) === (11a)&(11b)
```

`(11a)` is *True* at time 573 and time 1395. `(11b)` is *True* at time 573 and time 915. Therefore, `(11)` is *True* at time 573.

5.4.10 Implicit local variables

Until now, vector expressions are evaluated against an event report containing all variables within the scope of a cell. It is practical for the application to work with only one event report per cell or, at most, two event reports if the set of variables for `BEHAVIOR` (`scope=behavior`) and `VECTOR` (`scope=measure`) is different. However, for complex cells and megacells, it is sometimes necessary to change the scope of event observation, depending on operation modes. Different modes can require a different set of variables to be observed in different event reports.

The following definition allows to *extend* the scope of a vector expression locally:

> Edge operators apply not only to variables, but also to boolean expressions involving those variables. Those boolean expressions represent *implicit local variables* that are visible only within the vector expression where they appear.

Suppose the local variables `(A & B)`, `(A | B)` are inserted into the event report:

```
time  A     B     C     D     E     A&B   A|B
0     0     1     1     X     1     0     1
109   1     1     1     0     1     1     1
258   1     0     1     0     1     0     1
573   1     0     0     0     1     0     1
586   0     0     0     0     1     0     0
643   1     0     0     0     1     0     1
788   0     1     1     0     1     0     1
915   1     1     1     0     1     1     1
1062  1     1     1     0     0     1     1
1395  1     0     0     0     0     0     1
1640  0     0     0     1     0     0     0
```

Example:

```
01 (A & B)                                          // (12)

// event pattern expressed by (12):
//    A&B
//    0
//    1
```

`(12)` is *True* at time 109 and time 915.

```
10 (A | B)                                          // (13)

// event pattern expressed by (13):
//    A|B
//    1
//    0
```

`(13)` is *True* at time 586 and time 1640.

```
01 (A & B) -> 10 B                                  // (14)

// event pattern expressed by (14):
//    B     A&B
//    1     0
//    1     1
//    0     1
```

`(14)` is *True* at time 258.

```
10 ( A & B) & 10 B -> 10 C                          // (15)

// event pattern expressed by (15):
//    B     C     A&B
//    1     1     1
//    0     1     0
//    0     0     0
```

`(15)` is *True* at time 573.

```
10 (A & B) -> 10 (A | B)                            // (16)

// event pattern expressed by (16):
//    A&B   A|B
//    1     1
//    0     1
//    0     0
```

`(16)` is *True* at time 1640.

5.4.11 Conditional event sequences

The following definition *restricts* the scope of a vector expression locally:

`vector_boolean_and`, also called *conditional event operator*
This operator is defined between a vector expression and a boolean expression, using the overloaded symbol `&` or `&&`. The scope of the vector expression is restricted to the variables and eventual implicit local variables appearing within that vector expression. The boolean expression shall be *True* during the entire vector expression. The boolean expression is called the *Existence Condition* of the vector expression.[2]

Vector expressions using the `vector_boolean_and` operator are called `vector_conditional_event` expressions. Scope and contents of such expressions are identical, as opposed to non-conditional `vector_complex_event` expressions, where the content is a subset of the scope.

Example:

```
(10 (A & B) -> 10 (A | B)) & !D                          // (17)

// event pattern expressed by (17):
//    A&B   A|B
//    1     1
//    0     1
//    0     0

// event report without C, E:
time  A     B     D     A&B   A|B
0     0     1     X     0     1
109   1     1     0     1     1
258   1     0     0     0     1
586   0     0     0     0     0
643   1     0     0     0     1
788   0     1     0     0     1
915   1     1     0     1     1
1062  1     1     0     1     1
1395  1     0     0     0     1
1640  0     0     1     0     0
```

`(17)` contains the same `vector_complex_event` expression as `(16)`. However, although `(16)` is not *True* at time 586, `(17)` is *True* at time 586, since the scope of observation is narrowed to `A`, `B`, `A&B`, and `A|B` by the existence condition `!D`, which is statically *True* while the specified event sequence is observed.

Within, and only within, the narrowed scope of the `vector_conditional_event` expression, `(17)` can be considered equivalent to the following:

```
(10 (A & B) -> 10 (A | B)) & !D
===
(10 (A & B) -> 10 (A | B)) & (11 (!D) -> 11 (!D))
===
10 (A & B) & 11 (!D) -> 10 (A | B) & 11 (!D)
```

The transformation consists of the following steps:

1. Transform the boolean condition into a non-event.
   For example, `!D` becomes `11 (!D)`.
2. Left-extend the `vector_single_event` expression containing the non-event in order to match the length of the `vector_complex_event` expression.

---

2. An Existence Condition can also appear as annotation to a `VECTOR` object instead of appearing in the vector expression. This enables recognition of existence conditions by application tools which can not evaluate vector expressions (e.g., static timing analysis tools). However, for tools that can evaluate vector expressions, there is no difference between existence condition as a co-factor in the vector expression or as an annotation.

For example, `11 (!D)` becomes `11 (!D) -> 11 (!D)` to match the length of `10 (A & B) -> 10 (A | B)`.

3. Apply scalar multiplication rule for simultaneously occurring event sequences.

Thus, a `vector_conditional_event` expression can be transformed into an equivalent `vector_complex_event` expression, but the change of scope needs to be kept in mind. An operator which can express the change of scope in the vector expression language is defined in Section 5.4.13. This can make the transformation more rigorous.

Regardless of scope, the transformation from `vector_conditional_event` expression to `vector_complex_event` expression also provides the means of detecting ill-specified `vector_conditional_event` expressions.

Example:

```
(10 A -> 01 B -> 01 A) & A
===
10 A & 11 A -> 01 B & 11 A -> 01 A & 11 A
```

The first expression `10 A & 11 A` and the third expression `01 A & 11 A` within the `vector_complex_event` expression are contradictory. Hence, the `vector_conditional_event` expression can never be *True*.

5.4.12 Alternative conditional event sequences

All `vector_binary` operators, in particular the `vector_or` operator, can be applied to `vector_conditional_event` expressions as well as to `vector_complex_event` expressions.

Consider again the event report:

```
time  A     B     C     D     E
0     0     1     1     X     1
109   1     1     1     0     1
258   1     0     1     0     1
573   1     0     0     0     1
586   0     0     0     0     1
643   1     0     0     0     1
788   0     1     1     0     1
915   1     1     1     0     1
1062  1     1     1     0     0
1395  1     0     0     0     0
1640  0     0     0     1     0
```

Concurrent alternative `vector_conditional_event` expressions can be paraphrased in the following way:

```
IF <boolean_expression_1> THEN <vector_expression_1>
OR IF <boolean_expression_2> THEN <vector_expression_2>
... OR IF <boolean_expression_N> THEN <vector_expression_N>
```

The conditions can be *True* within overlapping time windows and yhus the vector expressions are evaluated concurrently. The `vector_boolean_and` operator and `vector_or` operator describe such vector expressions.

Example:

```
C&(01 A -> 10 B) | !D&(10 B -> 10 A) | E&(10 B -> 10 C)     // (18)

// Event pattern expressed by (18):
//    A     B     C
//    0     1     1
//    1     1     1
//    1     0     1
```

`(18)` is *True* at time 258 because of `C & (01 A -> 10 B)`.

```
// Alternative event pattern expressed by (18):
//    A     B     D
//    1     1     0
//    1     0     0
//    0     0     0
```

`(18)` is also *True* at time 586 because of `!D & (10 B -> 10 A)`.

```
// Alternative event pattern expressed by (18):
//    B     C     E
//    1     1     1
//    0     1     1
//    0     0     1
```

`(18)` is also *True* at time 573 because of `E & (10 B -> 10 C)`.

Prioritized alternative `vector_conditional_event` expressions can be paraphrased in the following way:

```
IF <boolean_expression1> THEN <vector_expression1>
ELSE IF <boolean_expression2> THEN <vector_expression2>
... ELSE IF <boolean_expressionN> THEN <vector_expressionN>
(optional) ELSE <vector_expressiondefault>
```

Only the vector expression with the highest priority *True* condition is evaluated. The `vector_boolean_cond` operator and `vector_boolean_else` operator are used in ALF to describe such vector expressions.

Example:

```
C? (01 A -> 10 B): !D? (10 B -> 10 A): E? (10 B -> 10 C)    // (19)
```

The prioritized alternative `vector_conditional_event` expression can be transformed into concurrent alternative `vector_conditional_event` expression as shown:

```
C ? (01 A -> 10 B) : !D ? (10 B -> 10 A) : E ? (10 B -> 10 C)
===
C & (01 A -> 10 B)
| !C & !D & (10 B -> 10 A)
| !C & !(!D) & E & (10 B -> 10 C)
```

`(19)` is *True* at time 258 because of `C & (01 A -> 10 B)`, but not at time 586 because of higher priority `C` while `!D & (10 B -> 10 A)`, nor at time 573 because of higher priority `!D` while `E & (10 B -> 10 C)`.

5.4.13 Change of scope within a vector expression

Conditions on vector expressions redefine the scope of vector expressions locally. The following definition can be used to change the scope even within a part of a vector expression. For this purpose, the symbolic state `*` can be used, which means “don’t care about events”. This is different from the symbolic state `?` which means “don’t care about state”. When the state of a variable is `*`, arbitrary events occurring on that variable are disregarded.

- Edge operator with `*` as next state:
  The variable to which the operator applies is no longer within the scope of the vector expression.
- Edge operator with `*` as previous state:
  The variable to which the edge operator applies is now within the scope of the vector expression.

As opposed to `?`, `*` stands for an infinite variety of possibilities.

Example:

Let `A` be a logic variable with the possible states `1`, `0`, and `X`.

```
*0 A ===
00 A | 10 A | X0 A
| 00 A -> 00 A | 10 A -> 00 A | X0 A -> 00 A
| 01 A -> 10 A | 11 A -> 10 A | X1 A -> 10 A
| 0X A -> X0 A | 1X A -> X0 A | XX A -> X0 A
| 00 A -> 00 A -> 00 A | ...
```

```
0* A ===
00 A | 01 A | 0X A
| 00 A -> 00 A | 00 A -> 01 A | 00 A -> 0X A
| 01 A -> 10 A | 01 A -> 11 A | 01 A -> 1X A
| 0X A -> X0 A | 0X A -> X1 A | 0X A -> XX A
| 00 A -> 00 A -> 00 A | ...
```

A vector expression with an infinite variety of possible event sequences cannot be directly matched with an event report. However, there are feasible ways to implement event sequence detection involving `*`. In principle, there is a “static” and “dynamic” way. The following parts of the vector expression are separated by `*` *sub-sequences* of events.

- “Static” event sequence detection with `*`:
  The event report with all variables can be maintained, but certain variables are masked for the purpose of detection of certain sub-sequences.
- “Dynamic” event sequence detection with `*`:
  The event report shall contain the set of variables necessary for detection of a relevant sub-sequence. When such a sub-sequence is detected, the set of variables in the event report shall change until the next sub-sequence is detected, etc.

Examples:

```
01 A -> 1* B -> 10 C                                    // (20)

// Event pattern expressed by (20):
//    A     B     C
//    0     1     1
//    1     1     1
//    1     *     1
//    1     *     0

// pattern for 1st sub-sequence:
//    A     B     C
//    0     1     1
//    1     1     1
//    1     *     1

// pattern for 2nd sub-sequence:
//    A     B     C
//    1     *     1
//    1     *     0
```

The event report with masking relevant for `(20)`:

```
time  A     B     C     D     E
0     0     1     1     X     1
109   1     1     1     0     1
258   1     *     1     0     1     // detection of 1st sub-sequence
573   1     *     0     0     1     // detection of 2nd sub-sequence
586   0     0     0     0     1
643   1     0     0     0     1
788   0     1     1     0     1
915   1     1     1     0     1
1062  1     *     1     0     0     // detection of 1st sub-sequence
1395  1     *     0     0     0     // detection of 2nd sub-sequence
1640  0     0     0     1     0
```

`(20)` is *True* at time 573 and time 1395. The first sub-sequence `01 A -> 1* B` is detected at time 258, since * maps to any state. From time 258 onwards, `B` is masked. The second sub-sequence `10 C` is detected at time 573. From time 573 onwards, `B` is unmasked. The first sub-sequence is detected again at time 1062. The second sub-sequence is detected again at time 1395.

```
01 A & 1* E -> 10 C                                     // (21)

// Event pattern expressed by (21):
//    A     C     E
//    0     1     1
//    1     1     *
//    1     0     *

// pattern for 1st sub-sequence:
//    A     C     E
//    0     1     1
//    1     1     *
```

```
// pattern for 2nd sub-sequence:
//    A     C     E
//    1     1     *
//    1     0     *
```

The event report with masking relevant for `(21)`:

```
time  A     B     C     D     E
0     0     1     1     X     1
109   1     1     1     0     *     // detection of 1st sub-sequence
258   1     0     1     0     *     // abortion of detection process
573   1     0     0     0     1
586   0     0     0     0     1
643   1     0     0     0     1
788   0     1     1     0     1
915   1     1     1     0     *     // detection of 1st sub-sequence
1062  1     1     1     0     *     // disregard event out of scope
1395  1     0     0     0     0     // detection of 2nd sub-sequence
1640  0     0     0     1     0
```

(21) is *True* at time1395. The first sub-sequence `01 A & 1* E` is detected at time 109. From time 109 onwards, `E` is masked. The event on `B` at time 258 aborts continuation of the detection process and triggers restart from the beginning. The first sub-sequence is detected again at time 915. From time 915 onwards, `E` is masked. The event at time 1062 is therefore out of scope. The second sub-sequence `10 C` is detected at time 1395.

```
01 A -> *1 B -> 10 B & 10 C                          // (22)

// Event pattern expressed by (22):
//    A     B     C
//    0     *     1
//    1     *     1
//    1     1     1
//    1     0     0

// pattern for 1st sub-sequence:
//    A     B     C
//    0     *     1
//    1     *     1

// pattern for 2nd sub-sequence:
//    A     B     C
//    1     *     1
//    1     1     1
//    1     0     0
```

The event report with masking relevant for (22):

```
time  A     B     C     D     E
0     0     1     1     X     1
109   1     1     1     0     1     // detection of 1st sub-sequence
258   1     0     1     0     1     // abort
573   1     *     0     0     1
586   0     *     0     0     1
643   1     *     0     0     1
788   0     *     1     0     1
915   1     *     1     0     1     // detection of 1st sub-sequence
1062  1     1     1     0     0     // continue
1395  1     0     0     0     0     // detection of 2nd sub-sequence
1640  0     0     0     1     0
```

(22) is *True* at time 1395. The first sub-sequence `01 A` is detected at time 109. Therefore, `B` is unmasked. Since `B=0` at time 258, the attempt to detect the second sub-sequence is aborted and the detection process restarts from the beginning. The first sub-sequence `01 A` is detected again at time 109. The second sub-sequence `*1 B -> 10 B & 10 C` is detected at time 1395.

```
01 A -> 1? A & 0* B & 1* E -> 10 C                    // (23)

// Event pattern expressed by (23):
//    A     B     C     E
//    0     0     1     1
//    1     0     1     1
//    1     *     1     *
//    1     *     0     *

// pattern for 1st sub-sequence:
//    A     B     C     E
//    0     0     1     1
//    1     0     1     1
//    ?     *     1     *

// pattern for 2nd sub-sequence:
//    A     B     C     E
//    ?     *     1     *
//    ?     *     0     *
```

The event report with masking relevant for (23):

```
time  A     B     C     D     E
0     0     1     1     X     1
109   1     1     1     0     1
258   1     0     1     0     1
573   1     0     0     0     1
586   0     0     0     0     1
643   1     0     0     0     1
788   0     *     1     0     *     // detection of 1st sub-sequence
915   1     *     1     0     *     // abort
1062  1     1     1     0     0
1395  1     0     0     0     0
1640  0     0     0     1     0
```

(23) is not *True* at any time. The first sub-sequence is detected at time 788. The event at time 915 does not match the expected second sub-sequence.

5.4.14 Sequences of conditional event sequences

The symbol * can be used to describe the scope of a vector expression directly in the vector expression language. This is particularly useful for sequences of vector_conditional_event expressions.

In reusing (17) as example:

```
(10 (A & B) -> 10 (A | B)) & !D
```

the scope of the sample event report contains contain the variables A, B, C, D, and E. The vector_conditional_event expression (17) contains only the variables A, B, and D and the implicit local variables A&B and A|B. Therefore, the global variables C and E are out of scope within (17). The implicit local variables A&B and A|B are in scope within, and only within, (17).

Now consider a *sequence* of vector_conditional_event expressions, where variables move in and out of scope. With the following formalism, it is possible to transform such a sequence into an equivalent vector_complex_event expression, allowing for a change of scope within each vector_conditional_event expression.

```
<vector_conditional_event#1> .. -> .. <vector_conditional_event#N>
```

where

```
<vector_conditional_event#i>
=== <vector_complex_event#i> & <boolean_expression#i>// 1 ≤ i ≤ N
```

The principle is to decompose each vector_conditional_event expression into a sequence of three vector expressions *prefix*, *kernel*, and *postfix* and then to reassemble the decomposed expressions.

```
<vector_conditional_event#i>
=== <prefix#i> -> <kernel#i> -> <postfix#i>// 1 ≤ i ≤ N
```

1. Define the prefix for each vector_conditional_event expression.
   The *prefix* is a vector_event expression defining all implicit local variables.

   Example:

   ```
   *? (A&B) & *? (A|B)
   ```

2. Define the kernel for each vector_conditional_event expression.
   The *kernel* is the vector_complex_event expression equivalent to the vector_conditional_event expression.

   ```
   <vector_complex_event#i> & <boolean_expression#i>
   ===   <vector_complex_event#i>
   &     (11 <boolean_expression#i> ..->.. 11 <boolean_expression#i>)
   ```

   The kernel can consist of one or several alternative vector_event_sequence expressions. Within each vector_event_sequence expression, the same set of global variables are pulled out of scope at the first vector_event expression and pushed back in scope at the last vector_event expression.

Example:

```
?* C & ?* E              // global variables out of scope
& 10 (A & B) & 11 (!D) -> 10 (A | B) & 11 (!D)
& *? C & *? E            // global variables back in scope
```

3. Define the postfix for each `vector_conditional_event` expression.
   The *postfix* is a `vector_event` expression removing all implicit local variables.

Example:

```
?* (A&B) & ?* (A|B)
```

4. Join the subsequent `vector_complex_event` expressions with the `vector_and` operator between prefix#i+1and kernel#i and also between postfix#i and kernel#i+1.

```
   .. <vector_conditional_event#i> -> <vector_conditional_event#i+1>
..
   ===   .. <prefix#i>
         -> <postfix#i-1> & <kernel#i> & <prefix#i+1>
         -> <postfix#i> & <kernel#i+1> & <prefix#i+2>
         -> <postfix#i+1> ..
```

The complete example:

```
(10 (A & B) -> 10 (A | B)) & !D
===
*? (A&B) & *? (A|B)
-> ?* C & ?* E
& 10 (A & B) & 11 (!D) -> 10 (A | B) & 11 (!D)
& *? C & *? E
-> ?* (A&B) & ?* (A|B)
```

Note: The in-and-out-of-scope definitions for global variables are within the kernel, whereas the in-and-out-of-scope definitions for global variables are within the prefix and postfix. In this way, the resulting `vector_complex_event` expression contains the same uninterrupted sequence of events as the original sequence of `vector_conditional_event` expressions.

5.4.15 Incompletely specified event sequences

So far the vector expression language has provided support for *completely specified event sequences* and also the capability to put variables temporarily in and out of scope for event observation. As opposed to changing the scope of event observation, *incompletely specified event sequences* require continuous observation of all variables while allowing the occurrence of intermediate events between the specified events. The following operator can be used for that purpose:

`vector_followed_by`, also called *followed-by operator*, using the symbol `~>`.
The `~>` operator is the separator between consecutively occurring events, with possible unspecified events in-between.

Detection of event sequences involving `~>` requires detection of the sub-sequence before `~>`, setting a flag, detection of the sub-sequence after `~>`, and clearing the flag.

This can be illustrated with a sample event report:

```
time  A     B     C     D     E
0     0     1     1     X     1
109   1     1     1     0     1     // 01 A detected, set flag
258   1     0     1     0     1
573   1     0     0     0     1     // 10 C detected, clear flag
586   0     0     0     0     1
643   1     0     0     0     1     // 01 A detected, set flag
788   0     1     1     0     1
915   1     1     1     0     1     // 01 A detected again
1062  1     1     1     0     0
1395  1     0     0     0     0     // 10 C detected, clear flag
1640  0     0     0     1     0
```

Example:

```
01 A ~> 10 C                                              // (24)
// as opposed to previous example (5):01 A -> 10 C
```

`(24)` is *True* at time 573 because of `01 A` at time 109 and `10 C` at time 573. It is *True* again at time 1395 because of `01 A` at time 643 and `10 C` at 1395. On the other hand, `(5)` is never *True* because there are always events in-between `01 A` and `10 C`.

Vector expressions consisting of `vector_event` expressions separated by `->` or by `~>` are called `vector_event_sequence` expressions, using the same syntax rules for the two different `vector_followed_by` operators. Consequently, all vector expressions involving `vector_event_sequence` expressions and `vector_binary` operators are called `vector_complex_event` expressions.

However, only a subset of the semantic transformation rules can be applied to vector expressions containing `~>`.

Associative rule applies for both `->` and `~>`.

```
(01 A ~> 01 B) ~> 01 C === 01 A ~> (01 C ~> 01 B ~> 01 C)

(01 A -> 01 B) -> 01 C === 01 A -> (01 C -> 01 B -> 01 C)

(01 A ~> 01 B) -> 01 C === 01 A ~> (01 C ~> 01 B -> 01 C)

(01 A -> 01 B) ~> 01 C === 01 A -> (01 C -> 01 B ~> 01 C)
```

Distributive rule applies for both `->` and `~>`.

```
(01 A | 01 B) -> 01 C === 01 A ~> 01 C | 01 B -> 01 C

(01 A | 01 B) ~> 01 C === 01 A ~> 01 C | 01 B ~> 01 C

(01 A | 01 B) -> 01 C === 01 A ~> 01 C | 01 B -> 01 C
```

Scalar multiplication rule applies only for `->`. The transformation involving `~>` is more complicated.

```
(01 A -> 01 B) & (01 C -> 01 D)
=== (01 A & 01 C) -> (01 B & 01 D)

(01 A ~> 01 B) & (01 C -> 01 D)
=== (01 A & 01 C) -> (01 B & 01 D)
|    01 A ~> 01 C -> (01 B & 01 D)
```

```
(01 A ~> 01 B) & (01 C ~> 01 D)
=== (01 A & 01 C) ~> (01 B & 01 D)
|    01 A ~> 01 C ~> (01 B & 01 D)
|    01 C ~> 01 A ~> (01 B & 01 D)
```

Transformation of `vector_conditional_event` expressions into `vector_complex_event` expressions applies only for `->`.

```
(01 A -> 01 B) & C
=== 01 A & 11 C -> 01 B & 11 C
```

```
(01 A ~> 01 B) & C
=== 01 A & 11 C ~> 01 B & 11 C
```

Since the `~>` operator allows intermediate events, there is no way to express the continuously *True* condition `C`.

5.4.16 How to determine well-specified vector expressions

By defining semantics for

alternative `vector_event_sequence` expressions

and establishing calculation rules for

transforming `vector_complex_event` expressions into alternative `vector_event_sequence` expressions

and for

transforming alternative `vector_conditional_event` expressions into alternative `vector_complex_event` expressions,

semantics are now defined for all vector expressions.

The calculation rules also provide means to determine whether a vector expression is well-specified or ill-specified. An ill-specified vector expression is contradictory in itself and can therefore never be *True*.

Once a vector expression is reduced to a set of alternative `vector_event_sequence` expressions, two criteria define whether a vector expression is well-defined or not.

- Compatibility between subsequent events on the same variable:
  The next state of earlier event shall be compatible with previous state of later event. This check applies only if no `~>` operator is found between the events.
- Compatibility between simultaneous events on the same variable:
  Both the previous and next state of both events shall be compatible. Such events commonly occur as intermediate calculation results within vector expression transformation.

The following compatibility rules apply:

- `?` is compatible with any other state. If the other state is `*`, the resulting state is `?`. Otherwise, the resulting state is the other state.
- `*` is compatible with any other state. The resulting state is the other state.
- Any other state is only compatible with itself.

Examples:

```
01 A -> 01 B -> 10 A
```

The next state of `01 A` is compatible with the previous state of `10 A`.

```
0X A -> 01 B -> 10 A
```

The next state of `0X A` is not compatible with the previous state of `10 A`.

```
0X A ~> 01 B -> 10 A
```

Compatibility check does not apply, since intermediate events are allowed.

```
01 A & 10 A
```

Both the previous and next state of **`A`** are contradictory; this results in an impossible event.

```
?1 A & 1? A
```

Both previous and next state of `A` are compatible; this results in the non-event `11 A`.

5.5 Variable declarations

Inside a `CELL` object, the `PIN` objects with the `PINTYPE digital` define variables for `FUNCTION` objects inside the same `CELL`. A *primary input variable* inside a `FUNCTION` shall be declared as a `PIN` with `DIRECTION=input` or `both` (since `DIRECTION=both` is a bidirectional pin). However, it is not required that all declared pins are used in the function. Output variables inside a `FUNCTION` need not be declared pins, since they are implicitly declared when they appear at the left-hand side (LHS) of an assignment.

Example:

```
CELL my_cell {
      PIN A {DIRECTION = input;}
      PIN B {DIRECTION = input;}
      PIN C {DIRECTION = output;}
      FUNCTION {
            BEHAVIOR {
                  D = A && B;
                  C = !D;
            }
      }
}
```

`C` and `D` are output variables that need not be declared prior to use. After implicit declaration, `D` is reused as an input variable. A and B are primary input variables.

5.5.1 BEHAVIOR

Inside `BEHAVIOR`, variables that appear at the LHS of an assignment conditionally controlled by a vector expression, as opposed to an unconditional continuous assignment, hold their values, when the vector expression evaluates *False*. Those variables are considered to have latch-type behavior.

Examples:

```
BEHAVIOR {
      @(G){
            Q = D;  // both Q and QN have latch-type behavior
            QN = !D;
      }
}
BEHAVIOR {
      @(G){
            Q = D;  // only Q has latch-type behavior
      }
      QN = !Q;
}
```

5.5.2 STATETABLE

The functional description can be supplemented by a `STATETABLE`, the first row of which contains the arguments that are object IDs of the declared `PIN`s. The arguments appear in two fields, the first is input and the second is output. The fields are separated by a `:`. The rows are separated by a `;`. The arguments can appear in both fields if the `PIN`s have attribute `direction=output` or `direction=both`. If `direction=output`, then the argument has latch-type behavior. The argument on the input field is considered previous state and the argument on the output field is considered the next state. If `direction=both`, then the argument on the input field applies for input direction and the argument on the output field applies for output direction of the bidirectional `PIN`.

Example:

```
CELL ff_sd {
      PIN  q {DIRECTION=output;}
      PIN  d {DIRECTION=input;}
      PIN cp {DIRECTION=input;
                  SIGNALTYPE=clock;
                  POLARITY=rising_edge;}
      PIN cd {DIRECTION=input; SIGNALTYPE=clear; POLARITY=low;}
      PIN sd {DIRECTION=input; SIGNALTYPE=set; POLARITY=low;}
      FUNCTION {
            BEHAVIOR {
                  @(!cd) {q = 0;} :(!sd) {q = 1;} :(01 cp) {q = d;}
            }
            STATETABLE {
                  cd sd  cp  d   q  : q ;
                  0  ?   ??  ?   ?  : 0 ;
                  1  0   ??  ?   ?  : 1 ;
                  1  1   1?  ?   0  : 0 ;
                  1  1   ?0  ?   1  : 1 ;
                  1  1   1?  ?   0  : 0 ;
                  1  1   ?0  ?   1  : 1 ;
                  1  1   01  ?   ?  :(d);
            }
      }
}
```

If the output variable with latch-type behavior depends only on the previous state of itself, as opposed to the previous state of other output variables with latch-type behavior, it is not necessary to use that output variable in the input field. This allows a more compact form of the STATETABLE.

Example:

```
STATETABLE {
      cd sd  cp  d  : q ;
      0  ?   ??  ?  : 0 ;
      1  0   ??  ?  : 1 ;
      1  1   1?  ?  :(q);
      1  1   ?0  ?  :(q);
      1  1   01  ?  :(d);
}
```

A generic ALF parser shall make the following semantic checks:

- Are all variables of a FUNCTION declared either by declaration as PIN names or through assignment?
- Does the STATETABLE exclusively contain declared PINs?
- Is the format of the STATETABLE, i.e., the number of elements in each field of each row, consistent?
- Are the values consistently either state or transition digits?
- Is the number of digits in each TABLE entry compatible with the signal bus width?

A more sophisticated checker for complete verification of logical consistency of a FUNCTION given in both equation and tabular representation is out of scope for a generic ALF parser, which checks only syntax and compliance to semantic rules. However, formal verification algorithms can be implemented in special-purpose ALF analyzers or model generators/ compilers.

5.5.3 Multi-dimensional variables

A group of pins of a cell can be logically considered together by declaring a PIN with a range. A pin can be declared with one dimension or two dimensions. For example,

```
PIN                    A ;   // declares a scalar pin A
PIN [1:8]              A1 ;  // declares pin A1 with bits numbered 1
                             // through 8
PIN [1:8]              A2[1:4] ;// declares pin A2 with two dimensions
```

When a pin is declared with one dimension, the left number in the range shall specify the most significant bit number and the right number shall specify the least significant bit number. If the pin is declared with two dimensions, the second dimension shall specify the index of the first and the last rows of the two-dimension pin object.

A PIN object can be referenced in one of the four forms:

- Individual bit - the pin name shall be followed by an index of the bit.
- Contiguous group of bits - the pin name shall be followed by the contiguous range of bits. The most significant and least significant bit numbers shall follow the same relationship as given in the declaration.
- Entire PIN object - only the pin name shall be used. It shall be illegal to reference the entire two-dimension pin object in any operation.
- One row of a PIN object - for a two-dimension pin object, the name of the pin shall be followed by the row index of that pin. It shall be illegal to reference the individual bit or a group of bits of a two-dimension pin object directly in an operation.

When a PIN object is referenced on the left-hand side of an assignment, the result of the right-hand side expression is copied from the least significant bit towards the most significant bit. If the right-hand side value has lesser number of bits than the referenced PIN object in an assignment, the right-hand side value shall be zero-extended to fill the remaining bits of the referenced PIN object. If the right-hand side value has more bits than the referenced PIN object in an assignment, the right-hand side value shall be truncated to the size of the referenced PIN object.

Example:

```
pin [1:8] A1;
pin [1:8] A2[1:32] ;

A1[8]   = 'b0 ;
A1[1:6] = 'o75 ;              // is equivalent to A1[1:6] = 'b111_101
A1[1:5] = 'o75 ;              // is equivalent to A1[1:5] = 'b11_101,
                              // left most bit is truncated
A2[18]  = 'h5 ;               // is equivalent to A2[18] = 'b0000_0101
                              // entire row 18 of A2 is assigned a value.
```

Two-dimension PIN objects shall be referenced with the row index. It shall be illegal to directly reference an individual bit or a contiguous group of bits of a two-dimension PIN object. It shall be illegal to reference the entire PIN object as a two-dimension PIN object.

Example:

```
pin [1:8] A2[1:32] ;
pin [1:8] B1 ;
pin C ;

                        // legal references and assignments

A2[10]  = 'h45 ;        // assign 'h45 to row 10 of A2  ('b0100_0101)
B1      = A2[10] ;      // copies whole row A2[10] to B1
C       = B1[3] ;       // c = 'b0

// Illegal references and assignments
// B1[3]   = A2[10][3] ;illegal reference to bit 3 of A2[10]
// A2      = B1 ;       illegal reference to entire A2
```

It shall be legal to use identifiers as an index, but expressions shall not be permitted.

Example:

```
pin [4:1] ADDR;

ADDR       = 'd 10;
A2[ADDR]   = 'h45 ;     // assign 'h45 to row 10 of A2  ('b0100_0101)

// A2[ADDR+1] = 'h45 ;  illegal
```

5.5.4 ROM initialization

The STATETABLE statement can be used to describe the contents of a ROM, as far as this content is fixed in the library.

Example:

```
CELL my_rom {
   CELLTYPE = memory;
   ATTRIBUTE { rom asynchronous }
   PIN[1:2] addr { DIRECTION = input; SIGNALTYPE = address; }
   PIN[7:0] dout { DIRECTION = output; SIGNALTYPE = data; }
   PIN[7:0] mem[1:4] { DIRECTION=none; VIEW=none; SIGNALTYPE=data; }
   FUNCTION {
      BEHAVIOR { dout = mem[addr]; }
      STATETABLE {
         addr: mem ;
         ‘h0:  ‘h5 ;
         ‘h1:  ‘hA ;
         ‘h2:  ‘h5 ;
         ‘h3:  ‘hA ;
      }
   }
}
```

For flexibility, a separate included file can be used:

```
CELL my_rom {
   CELLTYPE = memory;
   ATTRIBUTE { rom asynchronous }
   PIN[1:3] addr { DIRECTION = input; SIGNALTYPE = address; }
   PIN[7:0] dout { DIRECTION = output; SIGNALTYPE = data; }
   PIN[7:0] mem[1:4] { DIRECTION=none; VIEW=none; SIGNALTYPE=data; }
   FUNCTION {
      BEHAVIOR { dout = mem[addr]; }
      INCLUDE “rom_initialization_file.alf”
      }
   }
}
```

The contents of the included file `rom_initialization_file.alf` are:

```
STATETABLE {
   addr: mem ;
   'h0:  'h5 ;
   'h1:  'hA ;
   'h2:  'h5 ;
   'h3:  'hA ;
}
```

5.6 Predefined models

This section defines the use of predefined models in ALF.

5.6.1 Usage of PRIMITIVEs

A `PRIMITIVE` referenced in a `CELL` can replace the complete set of `PIN` and `FUNCTION` definition. `PIN`s can be declared before the reference to the `PRIMITIVE`, in order to provide supplementary annotations that cannot be inherited from the `PRIMITIVE`. However, the `CELL` shall be pin-compatible with the `PRIMITIVE`.

If the `PRIMITIVE` or a `CELL` is referenced in an annotation container such as `SCAN`, only the subset of `PIN`s used in the non-scan cell shall be compatible with the `PIN`s of the cell.

The pin names can be referenced by order or by name. In the latter case, the LHS is the pin name of the referenced `PRIMITIVE` or `CELL` (e.g., the non-scan cell), the RHS is the pin name of the actual cell. A constant logic value can also appear at the LHS or RHS, indicating a pin needs to be tied to a constant value. If this information is already specified in an annotation inside the `PIN` object itself, referencing between a pin name and a constant value is not necessary.

`PRIMITIVE`s can also be instantiated inside `BEHAVIOR`.

5.6.2 Concept of user-defined and predefined primitives

Primitives are described in ALF syntax. Primitives are generic cells containing `PIN` and FUNCTION objects only, i.e., no characterization data. The primitives are used for structural functional modeling.

Example:

```
PRIMITIVE MY_PRIMITIVE {
      PIN x { ... }
      PIN y { ... }
      PIN z { ... }
      FUNCTION { ... }
}
```

```
CELL MY_CELL {
      PIN a { ... }
      PIN b { ... }
      PIN c { ... }
      FUNCTION {
            BEHAVIOR { MY_PRIMITIVE { x=a; y=b; z=c; } }
      }
      ...
}
```

Extensible primitives, i.e., primitives with variable number of pins can be modeled using a `TEMPLATE`.

Example:

```
TEMPLATE EXTENSIBLE_PRIMITIVE{
      PRIMITIVE <primitive_name> {
            PIN [0:<max_index>] pin_name {  ... }
            ...
      }
}

// instantiation of the template creates a primitive
EXTENSIBLE_PRIMITIVE {
      primitive_name = MY_EXTENSIBLE_PRIMITIVE;
      max_index = 2;
}
```

The set of statements above is equivalent to the following statement:

```
PRIMITIVE MY_EXTENSIBLE_PRIMITIVE {
      PIN [0:2] pin_name {  ... }
            ...
}
```

The primitive can be used as shown in the following example:

```
CELL MY_MEGACELL {
      PIN a { ... }
      PIN b { ... }
      PIN c { ... }
      FUNCTION {
            BEHAVIOR {
                  // reference to the primitive
                  MY_EXTENSIBLE_PRIMITIVE {
                        pin_name[0] = a;
                        pin_name[1] = b;
                        pin_name[2] = c;
                  }
            }
      }
      ...
}
```

Primitives can be freely defined by the user. For convenience, ALF provides a set of predefined primitives with the reserved prefix `ALF_` in their name, which cannot be used by user-defined primitives.

For all PINs of predefined primitives, the following annotations are defined by default:

```
VIEW = functional;
SCOPE = behavioral;
```

For predefined extensible primitives, a placeholder can be directly in the PRIMITIVE definition:

```
PRIMITIVE ALF_EXTENSIBLE_PRIMITIVE {
            PIN [0:<max_index>] pin_name {  ... }
      ...
}
```

This is equivalent to the following more verbose set of statements:

```
TEMPLATE EXTENSIBLE_PRIMITIVE{
      PRIMITIVE <primitive_name> {
            PIN [0:<max_index>] pin_name {  ... }
            ...
      }
}

EXTENSIBLE_PRIMITIVE {
      primitive_name = ALF_EXTENSIBLE_PRIMITIVE;
      max_index = <max_index>;
}
```

5.6.3 Predefined combinational primitives

This section defines the use of predefined combinational primitives.

5.6.3.1 One input, multiple output primitives

There are two combinational primitives with one input pin and multiple output pins:

ALF_BUF and ALF_NOT

A GROUP statement is used to define the behavior of all output pins in one statement.

The output pins are indexed starting with 0. If 0 is the only index used, the index can be omitted when referencing the output pin, e.g., out refers to out[0].

```
PRIMITIVE ALF_BUF {
      GROUP index {0:<max_index>}
      PIN[0:<max_index>] out {
            DIRECTION = output ;
      }
      PIN in {
            DIRECTION = input ;
      }
```

```
      FUNCTION {
            BEHAVIOR {
                  out[index] = in;
            }
      }
}
```

Figure 5-7: Primitive model of ALF_BUF

```
PRIMITIVE ALF_NOT {
      GROUP index {0:<max_index>}
      PIN[0:<max_index>] out {
            DIRECTION = output ;
      }
      PIN in {
            DIRECTION = input ;
      }
      FUNCTION {
            BEHAVIOR {
                  out[index] = !in;
            }
      }
}
```

Figure 5-8: Primitive model of ALF_NOT

5.6.3.2 One output, multiple input primitives

There are six combinational primitives with one output pin and multiple input pins:

`ALF_AND`, `ALF_NAND`, `ALF_OR`, `ALF_NOR`, `ALF_XOR`, and `ALF_XNOR`

The input pins are indexed starting with `0`. If `0` is the only index used, the index can be omitted when referencing the input pin, e.g., `in` refers to `in[0]`.

```
PRIMITIVE ALF_AND {
      PIN out {
            DIRECTION = output;
      }
      PIN[0:<max_index>] in {
            DIRECTION = input;
      }
      FUNCTION {
            BEHAVIOR {
                  out = & in;
            }
      }
}
```

Figure 5-9: Primitive model of ALF_AND

```
PRIMITIVE ALF_NAND {
      PIN out {
            DIRECTION = output;
      }
      PIN[0:<max_index>] in {
            DIRECTION = input;
      }
      FUNCTION {
            BEHAVIOR {
                  out = ~& in;
            }
      }
}
```

Figure 5-10: Primitive model of ALF_NAND

```
PRIMITIVE ALF_OR {
      PIN out {
            DIRECTION = output;
      }
      PIN[0:<max_index>] in {
            DIRECTION = input;
      }
      FUNCTION {
            BEHAVIOR {
                  out = | in;
            }
      }
}
```

Figure 5-11: Primitive model of ALF_OR

```
PRIMITIVE ALF_NOR {
      PIN out {
            DIRECTION = output;
      }
      PIN[0:<max_index>] in {
            DIRECTION = input;
      }
      FUNCTION {
            BEHAVIOR {
                  out = ~| in;
            }
      }
}
```

Figure 5-12: Primitive model of ALF_NOR

```
PRIMITIVE ALF_XOR {
      PIN out {
            DIRECTION = output;
      }
      PIN[0:<max_index>] in {
            DIRECTION = input;
      }
      FUNCTION {
            BEHAVIOR {
                  out = ^in;
            }
      }
}
```

Figure 5-13: Primitive model of ALF_XOR

```
PRIMITIVE ALF_XNOR {
      PIN out {
            DIRECTION = output;
      }
      PIN[0:<max_index>] in {
            DIRECTION = input;
      }
      FUNCTION {
            BEHAVIOR {
                  out = ~^in;
            }
      }
}
```

Figure 5-14: Primitive model of ALF_XNOR

5.6.4 Predefined tristate primitives

There are four tristate primitives:

`ALF_BUFIF1`, `ALF_BUFIF0`, `ALF_NOTIF1`, and `ALF_NOTIF0`

```
PRIMITIVE ALF_BUFIF1 {
      PIN out {
            DIRECTION  = output;
            ENABLE_PIN = enable;
            ATTRIBUTE {TRISTATE}
      }
      PIN in  {
            DIRECTION  = input;
      }
      PIN enable {
            DIRECTION  = input;
            SIGNALTYPE = out_enable;
      }
```

```
        FUNCTION {
              BEHAVIOR {
                    out = (enable)? in : 'bZ;
              }
              STATETABLE {
                    enable in : out;
                     0     ?  : Z;
                     1     ?  : (in);
              }
        }
  }
```

Figure 5-15: Primitive model of ALF_BUFIF1

```
  PRIMITIVE ALF_BUFIF0 {
        PIN out {
              DIRECTION  = output;
              ENABLE_PIN = enable;
              ATTRIBUTE {TRISTATE}
        }
        PIN in  {
              DIRECTION  = input;
        }
        PIN enable {
              DIRECTION  = input;
              SIGNALTYPE = out_enable;
        }
        FUNCTION {
              BEHAVIOR {
                    out = (!enable)? in : 'bZ;
              }
              STATETABLE {
                    enable in : out;
                     1     ?  : Z;
                     0     ?  : (in);
              }
        }
  }
```

Figure 5-16: Primitive model of ALF_BUFIF0

```
  PRIMITIVE ALF_NOTIF1 {
        PIN out {
              DIRECTION  = output;
              ENABLE_PIN = enable;
              ATTRIBUTE {TRISTATE}
        }
        PIN in  {
              DIRECTION  = input;
        }
```

```
      PIN enable {
            DIRECTION  = input;
            SIGNALTYPE = out_enable;
      }
      FUNCTION {
            BEHAVIOR {
                  out = (enable)? !in : 'bZ;
            }
            STATETABLE {
                  enable in : out;
                   0     ?  : Z;
                   1     ?  : (!in);
            }
      }
}
```

Figure 5-17: Primitive model of ALF_NOTIF1

```
PRIMITIVE ALF_NOTIF0 {
      PIN out {
            DIRECTION  = output;
            ENABLE_PIN = enable;
            ATTRIBUTE {TRISTATE}
      }
      PIN in  {
            DIRECTION  = input;
      }
      PIN enable {
            DIRECTION  = input;
            SIGNALTYPE = out_enable;
      }
      FUNCTION {
            BEHAVIOR {
                  out = (!enable)? !in : 'bZ;
            }
            STATETABLE {
                  enable in : out;
                   1     ?  : Z;
                   0     ?  : (!in);
            }
      }
}
```

Figure 5-18: Primitive model of ALF_NOTIF0

5.6.5 Predefined multiplexor

The predefined multiplexor has a known output value if either the select signal and the selected data inputs are known or both data inputs have the same known value while the select signal is unknown.

```
PRIMITIVE ALF_MUX {
      PIN Q  {
            DIRECTION  = output;
            SIGNALTYPE = data;
      }
      PIN[1:0] D  {
            DIRECTION  = input;
            SIGNALTYPE = data;
      }
      PIN S  {
            DIRECTION  = input;
            SIGNALTYPE = select;
      }
      FUNCTION {
            BEHAVIOR {
                  Q = (S || (d[0] ~^ d[1]) )? d[1] : d[0];
            }
            STATETABLE {
                  D[0] D[1] S  : Q ;
                  ?    ?    0  : (D[0]);
                  ?    ?    1  : (D[1]);
                  0    0    ?  : 0;
                  1    1    ?  : 1;
            }
      }
}
```

Figure 5-19: Primitive model of ALF_MUX

5.6.6 Predefined flip-flop

A dual-rail output D-flip-flop with asynchronous set and clear pins is a generic edge-sensitive sequential device. Simpler flip-flops can be modeled using this primitive by setting input pins to appropriate constant values. More complex flip-flops can be modeled by adding combinational logic around the primitive.

A particularity of this model is the use of the last two pins `Q_CONFLICT` and `QN_CONFLICT`, which are virtual pins. They specify the state of `Q` and `QN` in the event `CLEAR` and `SET` become active simultaneously.

```
PRIMITIVE ALF_FLIPFLOP {
      PIN Q     {
            DIRECTION  = output;
            SIGNALTYPE = data;
            POLARITY   = non_inverted;
      }
```

```
PIN QN    {
      DIRECTION  = output;
      SIGNALTYPE = data;
      POLARITY   = inverted;
}
PIN D     {
      DIRECTION  = input;
      SIGNALTYPE = data;
}
PIN CLOCK {
      DIRECTION  = input;
      SIGNALTYPE = clock;
      POLARITY   = rising_edge;
}
PIN CLEAR {
      DIRECTION  = input;
      SIGNALTYPE = clear;
      POLARITY   = high;
      ACTION     = asynchronous;
}
PIN SET   {
      DIRECTION  = input;
      SIGNALTYPE = set;
      POLARITY   = high;
      ACTION     = asynchronous;
}
PIN Q_CONFLICT   {
      DIRECTION  = input;
      VIEW       = none;
}
PIN QN_CONFLICT  {
      DIRECTION  = input;
      VIEW       = none;
}
FUNCTION {
      ALIAS QX  = Q_CONFLICT;
      ALIAS QNX = QN_CONFLICT;
      BEHAVIOR {
            @ (CLEAR && SET) {
                  Q  = QX;
                  QN = QNX;
            }
            : (CLEAR) {
                  Q  = 0;
                  QN = 1;
            }
            : (SET) {
                  Q  = 1;
                  QN = 0;
            }
            : (01 CLOCK) {          // edge-sensitive behavior
                  Q  = D;
                  QN = !D;
            }
```

```
            }
            STATETABLE {
                  D CLOCK CLEAR SET QX  QNX :  Q    QN ;
                  ?  ??    1     1  ?   ?   : (QX) (QNX);
                  ?  ??    0     1  ?   ?   :  1    0 ;
                  ?  ??    1     0  ?   ?   :  0    1 ;
                  ?  1?    0     0  ?   ?   : (Q)  (QN) ;
                  ?  ?0    0     0  ?   ?   : (Q)  (QN) ;
                  ?  01    0     0  ?   ?   : (D)  (!D) ;
            }
      }
}
```

Figure 5-20: Primitive model of ALF_FLIPFLOP

5.6.7 Predefined latch

The dual-rail D-latch with set and clear pins has the same functionality as the flip-flop, except the level-sensitive clock (`ENABLE` pin) is used instead of the edge-sensitive clock.

```
PRIMITIVE ALF_LATCH {
      PIN Q     {
            DIRECTION  = output;
            SIGNALTYPE = data;
            POLARITY   = non_inverted;
      }
      PIN QN    {
            DIRECTION  = output;
            SIGNALTYPE = data;
            POLARITY   = inverted;
      }
      PIN D     {
            DIRECTION  = input;
            SIGNALTYPE = data;
      }
      PIN ENABLE {
            DIRECTION  = input;
            SIGNALTYPE = clock;
            POLARITY   = high;
      }
      PIN CLEAR {
            DIRECTION  = input;
            SIGNALTYPE = clear;
            POLARITY   = high;
            ACTION     = asynchronous;
      }
      PIN SET   {
            DIRECTION  = input;
            SIGNALTYPE = set;
            POLARITY   = high;
            ACTION     = asynchronous;
      }
```

```
        PIN Q_CONFLICT   {
              DIRECTION = input;
              VIEW      = none;
        }
        PIN QN_CONFLICT  {
              DIRECTION = input;
              VIEW      = none;
        }
        FUNCTION {
              ALIAS QX  = Q_CONFLICT;
              ALIAS QNX = QN_CONFLICT;
              BEHAVIOR {
                    @ (CLEAR && SET) {
                          Q  = QX;
                          QN = QNX;
                    }
                    : (CLEAR) {
                          Q  = 0;
                          QN = 1;
                    }
                    : (SET) {
                          Q  = 1;
                          QN = 0;
                    }
                    : (ENABLE) {            // level-sensitive behavior
                          Q  = D;
                          QN = !D;
                    }
              }
              STATETABLE {
                    D  ENABLE CLEAR SET QX  QNX :  Q    QN ;
                    ?  ?      1     1   ?   ?   : (QX) (QNX);
                    ?  ?      0     1   ?   ?   :  1    0 ;
                    ?  ?      1     0   ?   ?   :  0    1 ;
                    ?  0      0     0   ?   ?   : (Q)  (QN) ;
                    ?  1      0     0   ?   ?   : (D)  (!D) ;
              }
        }
  }
```

Figure 5-21: Primitive model of ALF_LATCH

5.6.8 Parameterizeable cells

The concept of describing primitives with variable bus size shall be extended to parameterizeable cells. Dynamic template instantiations can be used for that purpose.

Template definitions can incorporate any type of object. Placeholders in the template definition are the equivalent of parameters. Hence, the definition of parameterizeable cells is already supported within the support of general template definitions.

In a *static template instantiation*, which is identified by the name of the template and by the optional value assignment `static`, placeholders are replaced by fixed values or by complex objects containing fixed values. Non-referenced placeholders stay in place and eventually result in semantically unrecognizable objects, which cannot be processed by downstream applications. Such unrecognizable objects shall be disregarded.

In a *dynamic template instantiation*, which is identified by the name of the template and by the mandatory value assignment `dynamic`, some placeholders can not be replaced. Those placeholders are application parameters. The template definition can already contain certain relationships between parameters (e.g., arithmetic model and its arguments in the header). Therefore the template instantiation determines which parameters need application values in order to calculate values for other parameters.

Going one step further, even the relationship between parameters can be defined in the dynamic template instantiation rather than in the template definition. In this case, the identifiers inside the placeholders become variables for arithmetic assignments. This definition of variables shall only be recognized within the context of the dynamic template instantiation.

Arithmetic assignments provide a shorter syntax for equation-based arithmetic models where only placeholder-parameters are involved.

```
param1 = 1.5 + 0.4 * param2 ** 3 - 2.7 / param3
```

is equivalent to

```
param1 {
      HEADER { param2 param3 }
      EQUATION { 1.5 + 0.4 * param2 ** 3 - 2.7 / param3 }
}
```

For table-based models or for models where the arguments have children objects attached to them, the verbose syntax with `HEADER` needs to be used.

Example:

```
TEMPLATE adder {
   CELL <cellname> {
      PIN [ <bitwidth> : 1 ] A { DIRECTION = input; }
      PIN [ <bitwidth> : 1 ] B { DIRECTION = input; }
      PIN Cin { DIRECTION = input; }
      PIN [ <bitwidth> : 1 ] S { DIRECTION = output; }
      PIN Cout { DIRECTION = output; }

      FUNCTION {
         BEHAVIOR {
            S = A + B + Cin;
            Cout = (A + B + Cin >= (‘b1 << (<bitwidth> - 1)));
         }
      }
      AREA = <areavalue>;
      VECTOR (?! Cin -> ?! Cout) {
```

```
            DELAY {
               HEADER {
                  CAPACITANCE {PIN = Cout; }
                  SLEWRATE {PIN = Cin; }
               }
               EQUATION { <D0> + <D1>*CAPACITANCE + <D2>*SLEWRATE }
            }
         }
      }
   }
```

The template is used for instantiation of a hard macro:

```
   adder { /* a hard macro */
      cellname = ripple_carry_adder_16_bit;
      bitwidth = 16;
      areavalue = 500;
      // D0, D1, D2 are undefined. DELAY cannot be calculated.
   }
```

The static instantiation of the hard macro is equivalent to the following static object:

```
   CELL ripple_carry_adder_16_bit {
      PIN [ 16 : 1 ] A { DIRECTION = input; }
      PIN [ 16 : 1 ] B { DIRECTION = input; }
      PIN Cin { DIRECTION = input; }
      PIN [ 16 : 1 ] S { DIRECTION = output; }
      PIN Cout { DIRECTION = output; }

      FUNCTION {
         BEHAVIOR {
            S = A + B + Cin;
            Cout = (A + B + Cin >= 'b1000000000000000);
         }
      }

      AREA = 500 ;

      VECTOR (?! Cin -> ?! Cout) {
   //    DELAY {
   //       HEADER {
   //          CAPACITANCE {PIN = Cout; }
   //          SLEWRATE {PIN = Cin; }
   //       }
   //       EQUATION { <D0> + <D1>*CAPACITANCE + <D2>*SLEWRATE }
   //    }
      }
   }
```

Now the template is used for instantiation of a soft macro:

```
adder = dynamic { /* a soft macro */
   cellname = ripple_carry_adder_N_bit;
   areavalue = 20 + 30 * bitwidth;
   }
   D0 {
      HEADER { AREA { TABLE { 10 20 30 } } }
      TABLE { 15.6 34.3 50.7 }
   }
   D1 = 0.29;
   D2 = 0.08;
}
```

The dynamic instantiation of the soft macro results in an object for which certain data depend on the runtime-values of the placeholder-parameters, as indicated in *italic* below. The calculation method for such data, however, can be compiled statically (e.g., the equation for AREA is a function of bitwidth and the lookup table for D0 is a function of AREA).

```
CELL ripple_carry_adder_N_bit {
   PIN [ bitwidth : 1 ] A { DIRECTION = input; }
   PIN [ bitwidth : 1 ] B { DIRECTION = input; }
   PIN Cin { DIRECTION = input; }
   PIN [ bitwidth : 1 ] S { DIRECTION = output; }
   PIN Cout { DIRECTION = output; }

   FUNCTION {
      BEHAVIOR {
         S = A + B + Cin;
         Cout = (A + B + Cin >= (‘b1 << (bitwidth - 1)));
      }
   }

   AREA = 20 + 30 * bitwidth ;

   VECTOR (?! Cin -> ?! Cout) {
      DELAY {
         HEADER {
            CAPACITANCE {PIN = Cout; }
            SLEWRATE {PIN = Cin; }
            D0 {
               HEADER { AREA { TABLE { 10 20 30 } } }
               TABLE { 15.6 34.3 50.7 }
            }
         }
         EQUATION { D0 + 0.29*CAPACITANCE + 0.08*SLEWRATE }
      }
   }
}
```