Vector expressions in ALF - Proposal for amendment

1.0 Purpose of the proposal

The purpose of the proposal is to amend the semantic specification of vector expressions in ALF for a rigorous usage in simulation and formal verification.

2.0 Summary of Definitions in ALF1.0

The idea is not to change the concepts but to give more watertight definitions, wherever necessary. Therefore the concepts, as defined and approved in ALF 1.0, November 1997, are summarized first. Suggestions for amendments are introduced in square brackets. [This is a suggestion for amendment].

2.1 Definition of Concepts

Section 2, chapter 2.1, describe the basic concepts. Page 2-1, 3rd paragraph, reads:

> "Vectors describe the stimuli for characterization. This encompasses both the concept of timing arcs and logcal conditions. An exhaustive setof vectors can be generatedfrom functional information, although the complexity of the exhaustive set precludes it from practical usage. The characterizer makes a choice of the relevant subset for characterization."

The last paragraph on page 2-1 reads:

> "Abstraction is required for the characterization of megacells: vectors describe events on buses rather than on [scalar] pins; number and range of switching pins within a bus become additional characterization variables. Characterization measurements are expandable and can be extrapolated from [scalar] pin to bus."

Chapter 2.2 introduces functional modeling concepts. Subchapter 2.2.4 introduces the concept of vector expressions. The first two paragraphs on page 2-4 read:

> "In order to model generalized higher order sequential logic, the concept of vector expressions is introduced, 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 occurence of events."

The 3rd paragraph introduces the followed-by operator with an example. The 4th paragraph reads:

> "A vector expression is evaluated by an event sequence detection function.Like a single event or transition, this function evaluates true only at an infinitely short time when the event sequence is detected."

This paragraph is followed by a graphical illustration of the event sequence detection function for the first example. The last paragraph on page 2-4 reads:

> "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."

The 2nd paragraph on page 2-5 introduces the concept of vector expressions with logic conditions in form of a generic example.

> "A sequence of event[s] can also be gated with static logical conditions. For example,
> (01 CP -> 10 CP) && CD
> the pin CD must 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 that 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 will be introduced in section 3.5.4."

Comment to this paragraph: section 3.5.4 introduces the operators to be used in vector expressions. It does not formalize the semantic rule given within this generic example. We feel that a chapter on formal claculation rules with vector expressions is necessary, like formal calculation rules in boolean algebra. We suggest that the place be an additional subchapter to 3.9 "Functional modeling styles and rules".

Chapter 2.3 introduces timing and power modeling and the idea of using the same form of vector expressions as introduced for functional modeling. The first paragraph of 2.3.1 reads:

> "The timing models of cells consist of two types: delay models for combinational and sequential cells, and timing constraint models for sequential cells. Both types can be described by timing arcs. A timing arc is a sequence of two events which can be described by a vector expression "event e1 is followed by event e2"

Some examples are given in the sequel of 2.3.1.

Subchapter 2.3.2 introduces power modeling.

> "The set of vectors causing power consumption within a cell is a superset of [those] vectors causing the cell output to switch. While only the latter [i.e. vectors with switching output] are needed for delay characterization, more vectors are needed for accurate power characterization".

This is followed by examples of vectors with and without switching output. The most interesting example is at bottom of page 2-7, since it confirms the concept that a static logical condition must be true during the whole event sequence.

> "For a 2-input AND gate [with input pins A, B and output pin Z], if the event "01 A" is detected and then the event "10 B" is detected before the input-to-output delay elapses, a glitch is observed. It is possible to describe the glitch by a higher-order vector.

> In dynamic simulation with transport delay mode, the glitch would appear as follows:
> 01 A -> 10 B -> 01 Z -> 10 Z
> Simulation featuring transport delay mode with invalid-value-detection [i.e. glitch-error-detection] would exhibit the glitch as follows:
> 01 A -> 10 B -> 0X Z -> X0 Z
> [01 A -> 10 B -> ‘b0’bX Z -> ‘bX’b0 Z][1]
> Simulation with inertial delay mode would suppress the output transitions:
> (01 A -> 10 B) && !Z
> The last expression can be used for each of the three [simulation] modes, since !Z is always true at the time when the sequence 01 A -> 10 B is detected [stronger statement: ... !Z is always true from begin to end of the sequence 01 A -> 10 B].

The 2nd paragraph on page 2-8 puts static power into the picture.

> “State-dependent static power is also within the scope of vector-based power models. Static power consumption is activated [in a simulation model] in the same way as level-sensitive logic in functional modeling by a vector expression featuring steady states [incorrect wording, “vector expression featuring steady states“ must be corrected to “boolean expression”], whereas transient power consumption is activated similar to edge-sensitive logic by a vector expression featuring transitions. [redundant wording, “vector expression featuring transitions” is simply “vector expression”]“

The last paragraph of 2.3.1 on page 2-8 refers to a “pin-toggle power model”, which is not mentioned elsewhere. In an older draft, 2.3.1 contained a large section with examples of pin-toggle power model which was removed before 1.0. A copy of this early version of chapter 2.3.1 is in the addendum of this document. The reference to “pin-toggle power model” can stay, if we re-integrate the early version or parts of it into the spec. Otherwise we may change the conclusion of chpater 2.3.1 as follows:

> “[More abstract vector expressions are provided for power modeling of complex blocks, where simplification is needed in order to deal with the complexity of characterization vectors.]”

2.2 Lexical rules for vector expressions

After ALF 1.0 it was concluded that edge literals containing digits other than “0”, “1”, “?” must be based in order to avoid parser ambiguity. Therefore I suggest amendment, starting with lexical definitions in chapter 3.2.8

Existing definition:

> bit_literal ::= X | Z | L | H | U | W | ? | 0 | 1 | x | z | l | h | u | w

suggested amendment:

> bit_literal ::= numeric_bit_literal | alphabetic_bit_literal
> numeric_bit_literal ::= ? | 0 | 1

1. use based edge literals to avoid parser ambiguity

alphabetic_bit_literal ::= | Z | L | H | U | W | x | z | l | h | u | w

suggested rules against ambiguity

rule1:
alphabetic_bit_literal must be based outside a statetable. Non-based alphabetic_bit_literal outside a statetable is interpreted as identifier or part of identifier. alphabetic_bit_literal can be optionally based inside a statetable.
rule2:
numeric_bit_literal can be optionally based inside or outside a statetable.
rule 3:
In an edge literal, which is a pair of two based or non-based literals, either both literals must be based or none of them.

2.3 Operators for vector expressions

The operators for vector expressions are defined in chapter 3.5.4 (page 3-25 ff. in ALF 1.0, page 3-27 ff. in ALF 1.0.5). Contents are the same, presentation is better in ALF 1.0.5.

The unary vector operators represented by non-based edge literals are correctly enumerated:

01 10 00 11 0? 1? ?0 ?1 ??

For all other unary vector operators, which are represented by based edge literals, I suggest the following amendment:

If ’bl1’bl2 is an edge operator consisting of two based literals ‘bl1 and ‘bl2 and “w” is an expression which can take the value ‘bl1 or ‘bl2, then the following vector expressions are considered equivalent:

```
’bl1’bl2 w      == 10 (w == ‘bl1) && 01 (w == ‘bl2)
                == 01 (w != ‘bl1) && 01 (w == ‘bl2)
                == 10 (w == ‘bl1) && 10 (w != ‘bl2)
                == 01 (w != ‘bl1) && 10 (w != ‘bl2)
```

This will allow to develop formal calculation rules for vector expressions by using non-based edge literals only.

The following expression (top of page 3-27 in ALF 1.0, top of page 3-29 in ALF 1.0.5) needs to be corrected:

```
(?? a)   ==
         (0? a)||(1? a)||(Z? a)||(X? a)
||       (H? a)||(L? a)||(W? a)
||       (?0 a)||(?1 a)||(?Z a)||(?X a)
||       (?H a)||(?L a)||(?W a)
```

A syntax correction requests that most of the literals be based. However, we need semantic correction, since the symbolic edge operator ?? applies not only to bit literals. The following amendment is proposed:

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

The binary vector operators ->, <->, &>, <&> as well as the | operator between vector expressions, & operator between vector expressions or between vector expression and boolean expressions are defined.

In chapter 3.5.5 “operator priorities” it is stated that the binary vector operators have a stronger priority than the & operator. That means, the following vector expressions are equivalent (A, B, C are boolean expressions):
01 A -> 01 B & C == (01 A -> 01 B) & C

Priority makes makes only sense, if there is a meaning for (01 A -> 01 B) & C, as opposed to 01 A -> (01 B & C).

The meanings are intuitive:
(01 A -> 01 B) & C means, C must be true during the event sequence (01 A -> 01 B)
01 A -> (01 B & C) means, C must be true while the event (01 B) happens.

However, more formalization needs to be provided in an amendment.

Another existing definition is w.r.t. priority is left-to-right evaluation. For vector expressions this means:
01 A -> 01 B -> 01 C == (01 A -> 01 B) -> 01 C

The amendment shall also discuss the meaning of 01 A -> (01 B -> 01 C).

2.4 Semantic rules for vector expressions

Chapter 3.9 describes rules for combinational and sequential funcions.

The rules for combinational functions cn be summarized as follows:

Boolean expressions involving boolean operators on bits can evaluate “1”, “0” or “X”. The arguments of boolean expressions can have the value of a boolean literal. The values “H” and “L’ are mapped to “1” and “0”, respectively. The other values are mapped to “X”.

The rules for sequential functions can be summarized as follows:

Sequential assignments triggered by boolean expressions (i.e. level-sensitive logic) will issue the value “X”, if the boolean expression evaluates to “X”.

Sequential assignments triggered by vector expressions (i.e. edge-sensitive logic) will issue the value “X”, if there is ambiguity in the detection of the triggering event.

3.0 Suggested Amendments

This is only a draft to convey the idea. The wording may be different than in this proposal.

3.1 Data types for boolean and vector expressions

Values of logic expressions can be assigned to variables. Therefore clear data types are needed. We propose to split up the system of bit literals semantically into the following cathegories:

- Pure logic value type: 1 , 0, X
- logic value-strength type: 1, 0, X, H, L, W, Z
  [This data type may need more poulation in order to support accurate modeling of dynamic logic. The current set has 3 drive strengths: 1, 0, X are strong. H, L, W are weak. Z is very weak and does not specify the logic value, hence Z is implicitely unknown. Maybe “Z1” and “Z0” will suffice.]
- logic incertainty type: ?, U

The following rules shall apply:

- Boolean expressions can only take one of the pure logic values: 1 (true), 0 (false) or X (unknown).
- Values of logic value-strength types can only be assigned directly.
- Values of logic value-strength types shall be mapped to pure logic values for the purpose of evaluating boolean expressions.
- Values of logic incertainty type cannot be assigned for combinational logic.
- Values of logic incertainty type can be assigned for sequential logic with the following meaning:
  Assignment of “?” means in fact no assignment, variable keeps its previous value.
  Assignment of “U” means un-initializing the variable. “U” is the also the default value for any variable to which no a value has ever been assigned.
- Values of all types may appear in statetable
- The following boolean operators on bits have the same meaning as the bitwise operators applied to single bits:
  !, ~ : inversion
  &&, & : and
  ||, | : or
- The following boolean operators work on the pure logic values, i.e. value-strength values are mapped to pure logic values first:
  inversion, and, or, exor “^”, exnor “^~”.
- The following boolean operators work directly on the values of any data type:
  == : comparison for equality
  != : comparison for inequality
  Comparison of the same type of data shall always result in 1 (true) or 0 (false). Comparison of different type of data (e.g. pure logic value with logic value-strength value) shall result in X (unknown).

3.2 Calculation rules for vector expressions

Vector expressions carry a different data type which cannot be assigned to a logic variable. We may call this datatype “boolean event type”. It can take three values:
“instantaneously true” : specified event sequence is detected now
“instantaneously unknown” : not sure whether specified event sequence is detected now
“false” : specified event sequence is not detected now

The event queue model is necessary, when the event sequence consists of more than one event. The event queue model also builds a bridge to simulation and formal verification.

The event queue model is defined by its number of entries and by its depth, i.e. the recorded history. It is equivalent to an all-event trace for a restricted number of variables (the number of entries) and a restricted history (the depth). It is also equivalent to an event dump, when the recorded occurence time is disregarded. The event queue stores the initial state of all its entries at the begin of its history and the new state of at least one particular entry (more than one, if simultaneous events occur) per event.

Evaluation of a vector expression can hence be modeled with two components: The event queue, which is a sequential operation of storing events in a memory and the event match function, which performs comparison of the actual contents of the event queue with an event sequence described by the vector expression. The event match function can be actually represented as a boolean function, which evaluates 1 (match), 0 (no match) or X (unknown match). The vector expression itself evaluates “instantaneously true” or “instantaneously unknown” at the rising edge of the event match function and “false” otherwise.

With the event queue / event match model and the following proposed definitions, the calculation rules for vector expressions can actually be objectively proven instead of being subjectively defined.

definition:
For an arbitrary boolean expression “B”, a vector expression “01 B” can be defined, which evaluates “instantaneously true” only in the event that B changes its value from proveable 0 to proveable 1.

definition:
The vector expression “10 B” shall be equivalent to “01 !B”.

definition of event OR:
For two arbitrary vector expressions “01 B1” and “01 B2”, the vector expression
“01 B1 || 01 B2” can be defined, which evaluates “instantaneously true” in the event that
B1 changes its value from proveable 0 to proveable 1 OR
B2 changes its value from proveable 0 to proveable 1.
The symbol “|” or “||” can be used ith same meaning.
Event OR shall be associative and commutative:
01 B1 || 01 B2 == 01 B2 || 01 B1
01 B1 || 01 B2 || 01 B3 == (01 B1 || 01 B2) || 01 B3 == 01 B1 || (01 B2 || 01 B3)

definition of event AND:
For two arbitrary vector expressions “01 B1” and “01 B2”, the vector expression
“01 B1 & 01 B2” can be defined, which evaluates “instantaneously true” in the event that
B1 changes its value from proveable 0 to proveable 1 AND

B2 changes its value from proveable 0 to proveable 1 at the same time.
The symbol “&” or “&&” can be used with same meaning
Event AND shall be associative and commutative:
01 B1 & 01 B2 == 01 B2 & 01 B1
01 B1 & 01 B2 & 01 B3 == (01 B1 & 01 B2) & 01 B3 == 01 B1 & (01 B2 & 01 B3)
The following expressions shall be equivalent:
01 B1 & 01 B2 == 10 (!B1 & !B2) & 01 (B1 & B2)

auxiliar definition:
For an arbitrary vector expression V =: “01 B1 & 01 B2“ , a sequential function of a boolean variable “matchV” can be defined in ALF in such a way that the vector expression “01 matchV” is equivalent to the vector expression V.

```
BEHAVIOR {
        @ (10 B1 || 10 B2) { matchV = 0; }
        @ (01 B1 & 01 B2) { matchV = 1; }
}
```

definition of event-boolean AND:
For an arbitrary boolean expression B1 and an arbirary vector expression “01 B2”, the vector expression “B1 & 01 B2” can be defined, which evaluates “instantaneously true” in the event that B2 changes its value from proveable 0 to proveable 1 WHILE B1 is proveable 1.
The symbol “&” or “&&” can be used with same meaning
Event-boolean AND shall be commutative:
B1 & 01 B2 == 01 B2 & B1

auxiliar definition:
For an arbitrary vector expression V =: “B0 & 01 B1“ , a sequential function of a boolean variable “matchV” can be defined in ALF in such a way that the vector expression
“01 matchV” is equivalent to the vector expression V.

```
BEHAVIOR {
        @ (10 B0 || 10 B1) { matchV = 0; }
        @ (B0 & 01 B1) { matchV = 1; }
}
```

The following description is equivalent and can be used as a formal definition of the vector expression “B0 & 01 B1“:

```
BEHAVIOR { // behavior of “01 matchV” is same as “B0 & 01 B1”
        @ (10 B0 || 10 B1) {matchV = 0; }
        @ (01 B1) { matchV = B0; }
}
```

definition of event sequence:
For two arbitrary vector expressions “01 B1” and “01 B2”, the vector expression
“01 B1 -> 01 B2” can be defined, which evaluates “instantaneously true” in the event that
B1 changes its value from proveable 0 to proveable 1 AND
B2 changes its value from proveable 0 to proveable 1 while B1 is still proveable 1.

auxiliar definition:
For an arbitrary vector expression V =: “01 B1 -> 01 B2“ , a sequential function of a boolean variable “matchV” can be defined in ALF in such a way that the vector expression “01 matchV” is equivalent to the vector expression V.

```
BEHAVIOR {
        @ (10 B1 || 10 B2) { matchV = 0; }
        @ (01 B1 -> 01 B2) { matchV = 1; }
}
```

The following description is equivalent and can be used as a formal definition of the vector expression “01 B1 -> 01 B2“

```
BEHAVIOR { // behavior of “01 matchV” is same as “01 B1 -> 01 B2”
        @ (01 B1) { possible_matchV = 1;}
        @ (10 B1 || 10 B2) { possible_matchV = 0; matchV = 0; }
        @ (01 B2) { matchV = possible_matchV; }
}
```

It can also be used as a recursive definition of the vector expression
“01 B[1] -> 01 B[2] ... -> ... 01 B[n]” where B[i] are boolean expressions, $1 \leq i \leq n$

```
GROUP i { 2 : n }
BEHAVIOR {
        match[1] = B[1];
        matchV = match[n];
        @ (01 match[i-1]) {possible_match[i] = 1;}
        @ (10 match[i-1] || 10 B[i]) { possible_match[i] = 0; match[i] = 0; }
        @ (01 B[i]) { match[i] = possible_match[i]; }
}
```

This recursive definition makes the “->” operator left-associative:
01 B[1] -> 01 B[2] -> 01 B[3] == (01 B[1] -> 01 B[2]) -> 01 B[3]
since there exists a substitution “01 matchV” for “01 B[1] -> 01 B[2]” such that
“01 B[1] -> 01 B[2] -> 01 B[3]” can be substituted by “01 matchV -> 01 B[3]”

It can also be shown, that the “->” is NOT right-associative. Considering the vector expression “01 B[1] -> (01 B[2] -> 01 B[3])“, we can substitute “01 B[2] -> 01 B[3]” by “01 matchV” and obtain “01 B[1] -> 01 matchV”. There is no order imposed between “01 B[1]” and “01 B[2]”. “01 matchV” will happen, if “01 B[2]” is followed by “01 B[3]”, regardless of “01 B[1]”. Hence the following expressions shall be equivalent:

```
01 B[1] -> (01 B[2] -> 01 B[3])
==      01 B[1] -> 01 B[2] -> 01 B[3]
||      (01 B[1] & 01 B[2]) -> 01 B[3]
||      01 B[2] -> 01 B[1] -> 01 B[3]
```

or using the shorthand notation:
01 B[1] -> (01 B[2] -> 01 B[3]) == 01 B[1] <&> 01 B[2] -> 01 B[3]

The substitution technique can also be applied to define the behavior of
“(B[0] & 01 B[1]) -> 01 B[2]” versus “B[0] & (01 B[1] -> 01 B[2])”

```
BEHAVIOR { // behavior of “(B[0] & 01 B[1]) -> 01 B[2]“
        // behavior of “01 match[1]” is same as “B[0] & 01 B[1]”
        @ (10 B[0] || 10 B[1]) { match[1] = 0; }
        @ (01 B[1]) { match[1] = B[0]; }
        // behavior of “01 match[2]“ is same as “01 match[1] -> 01 B[2]“
        @ (01 match[1]) {possible_match[2] = 1;}
        @ (10 match[1] || 10 B[2]) { possible_match[2] = 0; match[2] = 0; }
        @ (01 B[2]) { match[2] = possible_match[2]; }
}
```

```
BEHAVIOR { // behavior of “B[0] & (01 B[1]) -> 01 B[2])“
        // behavior of “01 match[1]” is same as “01 B[1] & 01 B[2]”
        @ (01 B[1]) {possible_match[1] = 1;}
        @ (10 B[1] || 10 B[2]) { possible_match[1] = 0; match[1] = 0; }
        @ (01 B[2]) { match[1] = possible_match[1]; }
        // behavior of “01 match[2]“ is same as “B[0] & 01 match[1]“
        @ (10 B[0] || 10 match[1]) { match[2] = 0; }
        @ (01 match[1]) { match[2] = B[0]; }
}
```

From both descriptions it can be seen that B[0] must stay proveable 1 in order to make “01 match[2]” happen.
Hence the following vector expressions shall be considered equivalent:
(B[0] & 01 B[1]) -> 01 B[2] == B[0] & (01 B[1] -> 01 B2)

This becomes more intutitive by expressing “B[0] & 01 B[1]” in the semantically equivalent way:
B[0] & 01 B[1] == ?1 B[0] -> 01 B[1][1]
since the “?1” operator expresses any transition leading to 1 without specifying the state before the transition. It also includes the pseudo-transition “11”, i.e. constant 1.
Therefore the following equivalence is obvious from the left-associativity of the “->” operator:
(?1 B[0] -> 01 B[1]) -> 01 B[2] == ?1 B[0] -> 01 B[1] -> 01 B[2]
i.e. B[0] must stay 1 until after “01 B[2]” in order to make the event sequence happen.

Conversely, in the vector expression “01 B[1] -> (B[0] & 01 B[2])” B[0] needs only be 1 during “01 B[2]”. The state of B[0] during “01 B[1]” does not matter.
01 B[1] -> (?1 B[0] -> 01 B[2]) == 01 B[1] <&> ?1 B[0] -> 01 B[2]

Discussion item:
The old spec. ALF 1.0 says:
“?1 B” stands for an arbitrary transition on Bwith final state 1.
i.e. ?1 B == 01 B || 11 B || ‘bX’b1 B etc.
Should this be amended to
“?1 B” stands for an arbitrary sequence of transitions on B with final state 1,

1. Note however that “?1 B[0]” is NOT the same as “B[0]”. The former is a vector expression, the latter is a boolean expression.

i.e. ?1 B == 01 B || 11 B || (10 B -> 01 B) || (01 B -> 10 B -> 01 B) etc. etc.

4.0 Addendum

The addendum shows an old version of chapter 2.3.1. The parts essentially removed for ALF 1.0 are marked in red.

4.1 Old version of power modeling chapter (before ALF 0.7)

A straightforward way of power modeling is to simply extend the **delay model** for each timing arc by a third variable:

- **scaled average current**, which is measured by integrating and scaling the total transient current through the power supply of the cell for the specific timing arc or vector. The current measurement can start anytime before the first event of the vector starts and can end anytime after all transients of the vector have stabilized.

Variants of this model are scaled average power and energy, which are obtained by simple scaling of average current measurements:

power = current * Vdd
energy = current * Vdd * integration time

However, it may be preferable to use current, since it is a basic physical entity, such as time and length, which can be measured directly, whereas power and energy are derived entities.

The current measurement technique in simulation is exactly the same as it would be in the lab: by inserting a current meter between the supply voltage source and the supply terminal of the cell. From this standpoint, the total current flowing through Vdd is the basic measurable entity. Other entities such as

short-circuit current / power / energy
load current / power / energy
cell-internal current / power /energy

are derived entities. It may be possible to put current meters between particular transistors of the cell and declare the measurement results as “short-circuit current”, but the viability of these measurements will always be subject to discussion, besides the problem of reproducing those types of measurements in a lab.

However, if people prefer to express power consumption in derived entities, it is always possible to do so by manipulating the characterization data before putting it into the library, or by putting explicitly the equation into the library

avg. cell-internal current = avg. total current - avg. load current

where

avg. total current = raw measurement data
avg. load current = load capacitance * Vdd / integration time for rising output
0 for falling output

A detailed current-versus time model can be obtained by extending the scaled average current into a piece-wise rectangular or piece-wise linear current waveform, which simply means adding more variables to the model.

The set of vectors causing power consumption within a cell is a superset of vectors causing the cell output to switch. While only the latter are needed for delay characterization, more vectors are needed for accurate power characterization.

The most popular example is a flipflop, which consumes power at every edge of the clock, even if the output does not switch. The vectors for delay and power characterization could be described as follows:

```
01 CP -> 01 Q
01 CP -> 10 Q
```

The vectors for power characterization with clock-switching-only could be described as follows:

```
01 CP && Q==D
10 CP && Q==D
```

The D input having the same value as the Q output is a necessary and sufficient condition that the output will not switch at the rising edge of CP and that the value transferred to the master latch at the falling edge of CP will be the same as already stored. Hence those two vectors capture the actual power dissipation within the clock buffers only. Other power vectors could be defined to capture the power dissipation within the data buffers and the master latch etc.

The full set of power vectors is not an exhaustive set of all possible input stimuli to a cell. In the case of single-stage combinational logic cells, the set of power vectors is identical to the set of delay vectors. Any internal cell switching activity triggered by an input vector will be observable at the switching output - since there is only a single stage of DC-connected network. The same holds for logic cells with buffers at the output only: the switching activity is still visible through the buffer.

For example, a 4-input AND cell with following functionality

```
Z = A && B && C && D
```

will have the following full set of delay-and-power vectors, if implemented in a single stage + inverter:

```
01 A -> 01 Z
10 A -> 10 Z
01 B -> 01 Z
10 B -> 10 Z
01 C -> 01 Z
10 C -> 10 Z
01 D -> 01 Z
10 D -> 10 Z
```

Now, if the same cell is implemented within two levels of logic (2-input NAND into 2-input NOR), for instance

```
E = !(A && B)
F = !(C && D)
Z = !(E || F) = !E && !F = A && B && C && D
```

then we need the following additional power vectors:

```
01 A && B && !(C && D)              // E will switch, F will not switch
10 A && B && !(C && D)
01 B && A && !(C && D)              // E will switch, F will not switch
10 B && A && !(C && D)
01 C && D && !(A && B)              // F will switch, E will not switch
10 C && D && !(A && B)
01 D && C && !(A && B)              // F will switch, E will not switch
10 D && C && !(A && B)
```

Although the total number of vectors has doubled, the effective characterization overhead is less, since the additional power vectors have only the input transition time as variable, whereas the delay-power vectors have both input transition time and output load as variable.

Like there is more than one possibility to describe functionality by boolean expressions, there are also different ways of express the same vector. For instance, the previously defined delay vectors for the 4-input AND gate, could also be expressed as follows:

```
01 A && B && C && D
10 A && B && C && D
01 B && A && C && D
10 B && A && C && D
01 C && A && B && D
10 C && A && B && D
01 D && A && B && C
10 D && A && B && C
```

since the conditions on the other input pin are necessary and sufficient to make the output switch.

The first way has benefits in that it highlights the sequence of “power consuming events”, and can reduce the number of vectors required to describe the power behaviour of some cells. It also maps most closely to the traditional timing representation. The second method also has its benefits in that it allows the decoupling of the power due to the input change and that due to the output change, which can be useful for modeling complex cells, and can also help reduce innaccuracies caused by glitching (depending on the power tool implementation). It is up to the modeler to decide the best method for a particular characterization environment or on a cell by cell basis.

For a 2-input AND gate, if `01 A` happens and then `10 B` happens before the input-to-output delay elapses, we observe a **glitch**. It is possible to describe the glitch by a higher-order vector.

In dynamic simulation with **transport delay mode**, the glitch would appear as follows:

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

Simulation featuring **transport delay mode with invalid-value-detectio**n would exhibit the glitch as follows:

```
01 A -> 10 B -> 0X Z -> X0 Z
```

Simulation with **inertial delay mode** would suppress the output transitions:

```
(01 A -> 10 B) && !Z
```

The last expression can be used for each of the three modes, since `!Z` is always true at the time when the sequence `01 A -> 10 B` is detected.

Each way of expressing vectors can be derived from the cell functionality. The different examples for delay vectors (= timing arcs), power vectors, and glitch vectors emphasize the rich potential of modeling capabilities using vector expressions.

State-dependent **static power** is also within the scope of vector-based power models. Static power consumption is activated in the same way as level-sensitive logic in functional modeling by a vector expression featuring steady states, whereas transient power consumption is activated similar to edge-sensitive logic by a vector expression featuring transitions.

The advantages of adding power models *within each delay vector* and providing *extra power vectors* are the following:

- straightforward extension of delay characterization
- capable of yielding the most detailed and accurate model on gate-level
- each vector defines a comprehensive stimulus for power measurements

However, it is also possible to describe simpler power models with vector expressions. For instance, a vector for a simple toggle-count based power model of the NAND cell would be described as follows:

```
01 Z || 10 Z
```

For sake of simplicity, one could only count rising transitions, since a falling transition is always expected to follow:

```
01 Z
```

The power numbers per toggle need to be twice as much in the second model, since only half of the toggle rate is counted.

It is possible to add some sophistication to the model, e.g. for a flipflop, by counting toggle rate on both clock and output pin, in order to account for switching and non-switching power.

```
01 CP || 10 CP
01 Q || 10 Q
```

However, attention must be paid while annotating power numbers to the clock pin and to the output pin in order to avoid double counting: If `CP` toggled 100 times and `Q` toggled 30 times, the vectors

```
01 CP -> 01 Q
01 CP -> 10 Q
```

actually occurred 15 times each, whereas the vectors

```
01 CP && Q==D
10 CP && Q==D
```

occurred 20 times each, and the vectors

```
10 CP && !Q && D
10 CP && Q && !D
```

occurred 15 times each. This makes the total toggle count 15+15+20+20+15+15=100 on `CP`, and 15+15=30 on `Q`.

The simple pin-toggle power model is perfect for statistical average power analysis and very easy to evaluate by a tool. Yet the characterization becomes more of a challenge, since the information, which kind of stimulus needs to be applied, is no longer available.

In the example of the flipflop, the pin-toggle power can be derived from the vector-based power as follows (it would be more complicated, if we had extra vectors for `D` switching):

```
power(01 Q || 10 Q)
        = (     power(01 CP -> 01 Q)
              + power(10 CP && Q && !D)
              + power(01 CP -> 10 Q)
              + power(10 CP && !Q && D)
          ) / 4
```

```
                - (      power(01 CP && Q==D)
                     +   power(10 CP && Q==D)
                  ) / 2
power(01 CP || 10 CP)
                = (      power(01 CP && Q==D)
                     +   power(10 CP && Q==D)
                  ) / 2
```

While it is possible for flipflops to find exact equations in order to calculate pin-toggle power from vector-based power (of course it works only one-way: vector-based power cannot be calculated from pin-toggle power), the NAND cell shows the limitations of the pin-toggle power model.

In order to characterize the model, one must apply vectors. One could make the choice of applying all delay vectors and taking the average or picking just one or a subset. In any case, the model makes implicit *a priory* assumptions about the occurrence frequency of each vector, while these assumptions can only be validated *a posteriori* through analysis.

Let us assume the choice of the average model:

```
power(01 Z || 10 Z)
                = (      power(01 A -> 01 Z)
                     +   power(10 A -> 10 Z)
                     +   power(01 B -> 01 Z)
                     +   power(10 B -> 10 Z)
                     +   power(01 C -> 01 Z)
                     +   power(10 C -> 10 Z)
                     +   power(01 D -> 01 Z)
                     +   power(10 D -> 10 Z)
                  ) / 8
```

In one application, each vector could actually have about the same occurrence frequency, thus justifying this model, whereas in another application, some vectors could occur more often than others, which would be in favor of weighting each vector differently. Introducing pin-toggle power also for the input pins does not always yield better accuracy. The input pin dependent part of the output-toggle power applies only, if the input pin causes the output pin to toggle. Internal cell power triggered by input-toggle applies only, if the other input pins are in a particular state. The pin-toggle power model does not take into account the correlations between states and transitions on different pins, whereas the vector-based power model does.

As a conclusion, the vector-based power model is the more general and more reliable model. However, the pin-toggle power model can be expressed as a special case of the vector-based model, hence nothing precludes the choice of the pin-toggle power model. This model has its justification in tools working at a high level of abstraction, where accurate characterization is not a predominant requirement.