

ForSpec

ReferenceManual

June 24, 2001

Rev1.0

| | | |
|----------|---|----------|
| 1 | PREFACE | 5 |
| 1.1 | CONVENTIONS | 5 |
| 2 | LEXICAL CONVENTIONS | 6 |
| 2.1 | IDENTIFIERS | 6 |
| 2.2 | KEYWORDS | 6 |
| 2.3 | CONSTANTS | 7 |
| 2.4 | COMMENTS | 7 |
| 3 | BASIC TYPES | 8 |
| 3.1 | BIT VECTORS | 8 |
| 3.1.1 | <i>Constants</i> | 8 |
| 3.1.2 | <i>Don't Cares</i> | 9 |
| 3.1.3 | <i>Bit-Vector Bit -Wise Operators and Functions</i> | 9 |
| 3.1.4 | <i>Bit-Vector Length Operators and Functions</i> | 11 |
| 3.1.5 | <i>Bit-Vector Arithmetic</i> | 11 |
| 3.1.6 | <i>Bit-Vector Comparisons</i> | 12 |
| 3.1.7 | <i>Arithmetic Operations and Comparison with Integers</i> | 13 |
| 3.1.8 | <i>Conditional Operators</i> | 13 |
| 3.1.9 | <i>Temporal Operators on Bit -Vectors</i> | 14 |
| 3.2 | BOOLEAN | 15 |
| 3.3 | EVENTS | 17 |
| 3.4 | EXTENDED EVENTS | 17 |
| 3.5 | FORMULAS | 18 |
| 3.5.1 | <i>Clocks and Resets</i> | 19 |
| 3.6 | SEMANTICS | 20 |
| 3.6.1 | <i>Introduction</i> | 20 |
| 3.6.2 | <i>Semantics of Boolean Expressions and Boolean Connectives</i> | 21 |
| 3.6.3 | <i>Semantics of Events and Extended Events</i> | 26 |
| 3.6.4 | <i>Semantics of Formulas</i> | 28 |
| 3.6.5 | <i>Temporal Intervals</i> | 36 |

| | | |
|----------|--|-----------|
| 3.7 | INTEGERS | 43 |
| 3.7.1 | <i>IntegerConstants</i> | 43 |
| 3.7.2 | <i>IntegerArithmeticandIntegerExpressions</i> | 43 |
| 3.7.3 | <i>sizeofFunction</i> | 43 |
| 3.8 | ENUMERATEDTYPES | 44 |
| 3.8.1 | <i>Enumeratedtypes:Operators</i> | 44 |
| 4 | DECLARATIONSOFSPECIFICATIONVARIABLES | 46 |
| 5 | TEMPLATESANDDEFINITIONS | 47 |
| 5.1 | DEFINITIONS | 47 |
| 5.2 | TEMPLATES | 48 |
| 5.2.1 | <i>TemplateExamples</i> | 48 |
| 5.2.2 | <i>CommaasParameterDelimitervs.EventsConcatenation</i> | 49 |
| 6 | BLOCKS&SCOPING | 50 |
| 6.1 | BLOCK DEFINITION | 50 |
| 6.2 | ALIASING | 51 |
| 6.3 | BLOCK INSTANTIATION | 52 |
| 6.4 | SCOPING | 53 |
| 6.4.1 | <i>ReferencingtheDUT</i> | 55 |
| 6.5 | BLOCK EXAMPLES | 56 |
| 6.5.1 | <i>Example1</i> | 56 |
| 6.5.2 | <i>Example2</i> | 56 |
| 6.5.3 | <i>Example3</i> | 56 |
| 6.5.4 | <i>Example4</i> | 57 |
| 7 | ASSIGNMENTS | 58 |
| 7.1 | ASSIGNMENTS SYNTAX | 58 |
| 7.2 | SAR | 59 |
| 7.3 | ASSIGNMENTS TIMING (CURRENT-NEXT) | 60 |
| 7.4 | ASSIGN-CYCLES | 61 |
| 8 | ASSUMPTIONS/ASSERTIONS | 62 |

| | | |
|-----------|---|-----------|
| 8.1 | ASSUME, ASSERT, RESTRICT & MODEL..... | 62 |
| 9 | EXTERNAL SIGNALS AND MAPPING | 64 |
| 9.1 | DEFAULT PATH..... | 66 |
| 10 | INCLUDING FILES | 67 |
| 11 | PRECEDENCE AND ASSOCIATIVITY | 68 |

1 Preface

This document is intended as a reference manual for ForSpec. It describes ForSpec's syntax and semantics.

1.1 Conventions

The following syntax is used to describe ForSpec's syntax:

| | Syn. | Meaning | Example |
|--|-----------------|--|---|
| | <code>[]</code> | Optional elements <u>Note</u> that sometimes square brackets are really part of the syntax. In this case, bold square brackets (<code>[]</code>) are used. | <code>assume[[bit[n]] name:=] expression;</code> Includes: <code>assumex_assert:=x;</code> <code>assumebitx_assert:=x;</code> <code>assumebit[5]x_assert:=x;</code> <code>assumebit[5]x;</code> |
| | <i>italic</i> | <u>Emphasize</u> that the element stands for user input. | <i>identifier</i> can be any valid identifier. |
| | bold | <u>Emphasize</u> obligatory command elements. Obligatory elements may also appear in regular font. | case { e1:v1; ... en:vn; default :vd; }; |

2 Lexical Conventions

2.1 Identifiers

Identifiers refer to specification state variables, definitions of parameterized and unparameterized templates and external references (external to ForSpec). An identifier is a sequence of letters, digits, the \$, the #, the period, and the underscore (_) character. It must begin with a letter and is case sensitive.

Refer to (section 9) for more details on external references.

2.2 Keywords

All keywords and reserved functions can be written either in CAPITAL letters, small letters or both. For example: `assert`, `ASSERT` and `SSert` are all valid.

Keywords:

```
accept_on, always, assert, assume, bit, case, change_if,
change_on, clock, default, event, eventually, fair, false,
formula, inf, int, model, next, past, path, prefix, reject_on,
restrict, rigid, seq, triggers, true, trust, until, wnext, wuntil
```

Reserved built-in functions:

```
a_change, a_fall, a_rise, ashr, b_change, b_fall, b_rise, bdec,
bsub, cadd, cinc, dec, inc, mutex, nand, nor, nxor, ox, repeat,
same, shl, shll, shr, shrlshr, sizeof, strong_mutex, sx, zx
```

2.3 Constants

There are several types of constants (examples are given in parenthesis):

- Integer: decimal (0, 29)
- Bit-vector: binary (**0b**0010), hexadecimal (**0x**A6, **0x**f3), octal (**0**75, **0**12)
Bit-vectors may contain wildcards. See 3.1.2 below.
- Boolean: true, false

2.4 Comments

- One-line comments: // ...
- Multi-line or in-line comments: /* ... */

3 Basic types

- Integer
- Bit-vectors
- Enumerated types
- Boolean
- Event
- Extended event
- Formula

3.1 Bit Vectors

The `bit` -vector data type models finite vectors of bits. A bit vector is defined as:

```
bit[n] x;
```

Where `n` is the size of `x`. The elements of `bit` -vector with size `n` are indexed from `0` to `(n-1)`.

```
bit x;
```

Is a shorthand for `bit` -vector of size `1`.

3.1.1 Constants

| Semantics | Syntax |
|--------------------|-------------------------|
| Binary number | 0b [0-1, _]+ |
| Hexadecimal number | 0x [0-9,a-f, _]+ |
| Octal number | 0 [0-7, _]+ |

3.1.2 Don'tCares

Bit-vectors may contain don't care symbols, notated by an underscore (_). A don't care symbol indicates that any digit can replace the underscore, according to the constant's base (binary, octal or hexadecimal). Don't cares can only be used with equality (=) and in definitions.

| Syntax | Semantics |
|--------|----------------|
| 0b0_1 | 0b001 or 0b011 |
| 0xE_ | 0xE0 to 0xEF |
| 07_ | 070 to 077 |

Examples:

```
assert x = 0x4_F;      // OK
x := 0b0__1;         // OK
assert 0x4_F;        // Error - not in a =
```

3.1.3 Bit-Vector Bit-Wise Operators and Functions

Let x and y be bit-vectors of identical size, and let n be an integer expression (see section 3.7.2). The return value of the following operations has the same width as the operands.

| Semantics | Syntax |
|--|--------------|
| Bit-wise 1's complement | $\sim x$ |
| Bit-wise or | $x y$ |
| Bit-wise and | $x \& y$ |
| Bit-wise xor | $x \wedge y$ |
| Logical shift left of x by n bits | $x \ll n$ |
| Logical shift right of x by n bits | $x \gg n$ |

The return value of the following functions has the same width as the operand(s):

| | |
|---------------------------------------|--------------------|
| Bit-wise nor | nor (x,y) |
| Bit-wise nand | nand (x,y) |
| Bit-wise not xor | nxor (x,y) |
| Logical shift left of x by n bits | shl (x,n) |
| Shift left, adding 1's from the right | shl1 (x,n) |
| Logical shift right of x by n bits | shr (x,n) |
| Shift right, adding 1's from the left | shr1 (x,n) |
| Arithmetic shift right | ashr (x,n) |

The following functions return a Boolean:

| | |
|---|-------------------------|
| True if at most one bit of x is true | mutex (x) |
| True if exactly one bit of x is true | strong_mutex (x) |
| True if all bits of x have the same value | same (x) |

3.1.4 Bit-Vector Length Operators and Functions

Let x be a bit $-$ vector of any width, b a bit $-$ vector of width 1 and i and j be integer expressions (see section 3.7.2). The following operators concatenate or truncate their operand:

| Semantics | Syntax | Return Size |
|------------------------|---------------------------|-------------|
| Sub-vector, $i \geq j$ | $x[i, j]$ or $x[i:j]$ | $i-j+1$ |
| Bit extraction | $x[i]$ | 1 |
| Concatenation | $x \% y$ | $ x + y $ |

Functions:

| | | |
|--|--------------------------|-----|
| Binary constant $-$ bit b appears n times | repeat (b, n) | n |
| Extending x to width n , adding 0's on the left | zx (x, n) | n |
| Extending x to width n , adding 1's on the left | ox (x, n) | n |
| Extending x to width n , adding the sign on the left | sx (x, n) | n |

3.1.5 Bit-Vector Arithmetic

Let x and y be bit $-$ vectors of identical size. The size of x is represented by $|x|$.

| Semantics | Syntax | Result Size |
|----------------|---------|-------------|
| Addition | $x + y$ | $ x $ |
| Subtraction | $x - y$ | $ x $ |
| Multiplication | $x * y$ | $2 * x $ |

Functions:

| | | |
|-----------|----------------|---|
| Increment | inc (x) | x |
| Decrement | dec (x) | x |

The following functions return a Boolean (see paragraph 3.2):

| | |
|--------------------------------------|-------------------|
| the carry of the addition of x and y | cadd (x,y) |
| the borrow of the subtraction x - y | bsub (x,y) |
| the carry of incrementing x | cinc (x) |
| the borrow of decrementing x | bdec (y) |

3.1.6 Bit-Vector Comparisons

Let x and y be bit-vectors of identical size. All operators return a Boolean.

| Semantics | Syntax |
|---------------------------|---------------|
| Equality | x =y |
| Nonequality | x !=y |
| x less than y | x <y |
| x less than or equal to y | x <=y |
| x greater than y | x >y |
| x greater or equal to y | x >=y |

3.1.7 Arithmetic Operations and Comparison with Integers

It is possible to perform arithmetic operations and comparisons between bit n -vectors and integer expressions. In this case, the integer operand is converted to a bit n -vector with the same width as the bit n -vector operand. An error is reported if the conversion is impossible.

3.1.8 Conditional Operators

Let b_1, b_2, \dots, b_n be Boolean expressions. Let v_1, v_2, \dots, v_n be bit n -vectors of identical size and e_1, \dots, e_n bit vectors of identical size (v_i and e_i may have different sizes).

| Semantics | Syntax |
|--|---|
| if b then v_1 else v_2 | $(b) ? v_1 : v_2$ |
| if $(e=e_1)$ then v_1 else if $(e=e_2)$ then v_2 ... else if $(e=e_n)$ then v_n else v_d | case { $e_1 : v_1;$... $e_n : v_n;$ default : $v_d;$ }; |
| if b_1 then v_1 else if b_2 then v_2 ... else if b_n then v_n else v_d | case { $b_1 : v_1;$... $b_n : v_n;$ default : $v_d;$ }; |

Notice:

- Case statements final switch must be the default switch. Case statements must end with a semicolon (;).
- If-then-else requires parenthesis on the Boolean condition.

3.1.9 Temporal Operator on Bit -Vectors

Let x be a bit -vector, let n be an integer expression (see section 3.7.2) and let clk be a Boolean expression.

| Semantics | Syntax |
|---|-----------------------------|
| The value of x , n clocks before present time | past (x, n, clk) |

The following are shorthand formats for these operators. The keyword **CLOCK** is the default active clock (in the current scope):

| Semantics | Syntax |
|--------------------------------------|------------------------|
| Past (x, n, CLOCK) | past (x, n) |
| Past ($x, 1, \text{CLOCK}$) | past (x) |

While **past** looks into the past, the prime operator **'** looks into the future. The prime operator has only one argument, implicitly its associated clock is true.

| Semantics | Syntax |
|--|--------|
| The value of x in the next primary phase | x' |

3.2 Boolean

A Boolean data type has 2 values: false(0) and true(1).

3.2.1.1 Boolean Constants

| Semantics | Syntax |
|-------------------|-------------------|
| Boolean constants | false, true, 0, 1 |

3.2.1.2 Boolean operators

Boolean is basically a bit -vector with size one. Therefore a 1-bit -vector operations may be applied to Boolean too. In addition, the operators in the table below can be applied solely to Boolean operands. They all return a Boolean value. Let x and y be Boolean.

| Semantics | Syntax |
|------------------------|-----------------------|
| Negation | $\neg x$ |
| If x then y | $x \rightarrow y$ |
| x if and only if y | $x \leftrightarrow y$ |

3.2.1.3 TemporalOperatorsonBoolean

Let b be Boolean and let v be any bitvector.

| Semantics | Syntax |
|--|-------------------------|
| Afterchange $v \neq \text{past}(v, 1, \text{true})$ | a_change (v) |
| Beforechange $v \neq v'$ | b_change (v) |
| Afterrise $(\neg \text{past}(b, 1, \text{true}) \ \& \ b$ | a_rise (b) |
| Beforerise $!b \ \& \ b'$ | b_rise (b) |
| Afterfall $\text{past}(b, 1, \text{true}) \ \& \ !b$ | a_fall (b) |
| Beforefall $b \ \& \ !b'$ | b_fall (b) |

3.3 Events

Let b be Boolean, let r_1 and r_2 be regular expressions, and let $n < m$ be integer expressions with values at least one. The following are also regular expressions:

| Semantics | Syntax |
|---|---------------------|
| Every Boolean expression | b |
| Concatenation | r_1, r_2 |
| Glue (also called concatenation with overlap) | $r_1 \setminus r_2$ |
| Or (also called union) | $r_1 \mid r_2$ |
| Zero or more times | r^* |
| Zero or one time | $r?$ |
| One or more times | r^+ |
| n times | $r\{n\}$ |
| k times, where $n \leq k \leq m$ | $r\{n, m\}$ |

An **event** is a regular expression r , such that language of r does not contain the empty word.

3.4 Extended Events

Every event is an *extended event*. Applying *change_on* operator to an event results in an extended event. Extended events can be used on the left-hand side of these *seq* and *triggers* operators. Refer to section 3.5 below, for more details.

3.5 Formulas

Let k, m, n be integer expressions such that $0 < k < \text{INF}$ and $0 \leq m \leq n \leq \text{INF}$. Assume also that f, g and e are formulas and e is an extended event. Then the following are also formulas:

| Semantics | Syntax |
|----------------------|----------------------------|
| Every extended event | ev |
| NOT | $!f$ |
| OR | $f \mid g$ |
| AND | $f \& g$ |
| IMPLIES | $f \rightarrow g$ |
| IF AND ONLY IF | $f \leftrightarrow g$ |
| SEQ | $e \text{ seq } f$ |
| TRIGGERS | $e \text{ triggers } f$ |
| NEXT | $\text{next}[k]f$ |
| WNEXT | $\text{wnext}[k]f$ |
| UNTIL | $f \text{ until}[m,n] g$ |
| WUNTIL | $f \text{ wuntil}[m,n] g$ |
| ALWAYS | $\text{always}[m,n] f$ |
| EVENTUALLY | $\text{eventually}[m,n] f$ |

Shortenings (without specifying timeframes):

| Semantics | Syntax |
|--------------------|---------------------|
| next[1]f | next f |
| wnext[1]f | wnext f |
| until[0,inf]f | until f |
| wuntil[0,inf]f | wuntil f |
| always[0,inf]f | always f |
| eventually[0,inf]f | eventually f |

3.5.1 Clocks and Resets

Let b be a Boolean, and let f be a formula. Then the following are also formulas:

| Semantics | Syntax |
|-----------|------------------------|
| change_on | change_on (b) f |
| change_if | change_if (b) f |
| accept_on | accept_on (b) f |
| reject_on | reject_on (b) f |

3.6 Semantics

3.6.1 Introduction

An FPV session is composed of an RTL model and ForSpec files which contain free variables and a set of assumptions, assertions and assignments. The set of computations (traces) that correspond to an FPV session contains all the traces that satisfy the following:

- Each state (point) on the trace is some assignment to the free variables and to the RTL signals, and restricting the trace to the RTL signals forms a legal RTL trace.

An FPV session passes if all the sessions' traces that satisfy the assumptions and the assignments also satisfy all the assertions. If an FPV session does not pass, it means that a counterexample exists. A counterexample is any trace that satisfies the assumptions and assignments but does not satisfy (at least) one of the assertions.

Assumptions and assertions are ForSpec formulas. Therefore, to determine whether or not an FPV session passes it is necessary to define when a trace satisfies a ForSpec formula. In this section we formally define ForSpec semantics, which determine whether or not a trace satisfies a ForSpec formula.

The semantics of ForSpec formulas are defined in relation to a context. The context is composed of five elements:

- a trace
- a point on the trace
- a clock
- an accept condition
- a reject condition

The accept condition and the reject condition will be referred to as reset signals.

Both the clock and the reset signals are Boolean expressions, where the Boolean expression that corresponds to the clock is not allowed to contain the keyword CLOCK and the two reset signals are mutually exclusive.

A context in which the point on the trace is point 0, the clock is the Boolean expression **true**, and both reset signals are **false** is called an *initial*, or *default* context.

The role of the clock is to define tick points (or sampling points) at which the formula is evaluated. The tick points are the points where the Boolean expression that corresponds to the clock is true.

The value of the clock in the initial context is **true**. The clock of the context can be changed using the operators `CHANGE_ON` and `CHANGE_IF`. That is, by means of these operators the sampling points can be changed. Note that unless a formula contains one of these two operators, every point on the trace is a tick point.

The reset signals in ForSpec are two Boolean expressions that define when evaluation of the formula should be halted. Evaluation is stopped at the first future point¹ that satisfies one of the reset signals, regardless of whether or not that point is a tick point. Whenever evaluation of a formula is halted by an accept or reject signal, the formula is accepted or rejected accordingly.

The reset signals can be changed using the operators `ACCEPT_ON` and `REJECT_ON`, which change the accept signal and the reject signal, respectively. The semantics of these two operators ensure that in the context, the accept signal and the reject signal can never be true simultaneously. Therefore, it is impossible to encounter both signals at the same point.

The semantics of the ForSpec language defines when a context π, i, c, a, r (π is a trace, i is a point on the trace, c is the clock, a and r are the accept and reject signals, respectively) satisfies a ForSpec formula f . This is called the *satisfaction relation*, and is denoted by $\pi, i, c, a, r \models f$. If the clock is true and both reset signals are false we simply write $\pi, i \models f$. We say that a trace satisfies a ForSpec formula whenever the initial context satisfies the formula (that is, $\pi, 0, \text{true}, \text{false}, \text{false} \models f$, or for short $\pi, 0 \models f$). The definition of ForSpec semantics (the satisfaction relation) is by induction on the structure of the formula where the induction basis contains the semantics of Boolean expressions.

3.6.2 Semantics of Boolean Expressions and Boolean Connectives

¹In this section, the term “future points” includes the present point.

3.6.2.1 Boolean Expressions

The decision about whether a Boolean expression is satisfied at a given point is made at that point. A Boolean expression, that does not contain the keyword `CLOCK`, holds at a given point on the trace if it is true at that point. When the Boolean expression b is true at point i on the trace, we write $\pi_i(b)=1$, otherwise we write $\pi_i(b)=0$. Boolean expressions that contain the keyword `CLOCK` have special semantics, which is described below.

Without locks and resets

A Boolean expression b holds at a given point on the trace if the Boolean expression that results from replacing every instance of the keyword `CLOCK` in b with `true` is true at that point. We use the term $[c \rightarrow d]$ to denote the Boolean expression that results from substituting each occurrence of `CLOCK` in b by the Boolean expression c . We will make sure that whenever we use the notation $[c \rightarrow d]$ the expression c does not contain any reference to the parameter `CLOCK` (explicitly or implicitly).

Formally:

$$\pi_i, j = b \text{ iff } \pi_i([true \rightarrow b]) = 1$$

With locks and resets

With a lock but without resets

A Boolean expression b holds at a given point if the Boolean expression $[c \rightarrow b]$ holds at that point.

Formally:

$$\pi_i, i, c = b \text{ iff } \pi_i([c \rightarrow b]) = 1$$

From now on, when we say that a Boolean expression holds at a given point, we mean that the Boolean expression that results from substituting each occurrence of `CLOCK` in the Boolean expression b by the clock of the context holds at that point.

With clock and reset signals

A Boolean expression b holds at a given point either if at that point on the trace the accept signal holds or at that point the reject signal does not hold and the Boolean expression b holds. It is easy to see that the accept signal always makes a Boolean expression true, while

therejects signal always makes it false. While it may appear that preference is being given to the accepts signal, in reality this is not true because the accepts signal and therejects signal can never be true simultaneously.

Formally:

$$\pi, i, c, a, r | = b \text{ if either } \pi, i, c | = a \text{ or } \pi, i, c | = b \text{ and } \pi, i, c | = !r$$

3.6.2.2 BitVectors

Let v be a bit vector and n be a constant. The expression $v \text{ equals } n$ holds at a given point on a trace if the value of v at that point is n .

Without clock and reset signals

$$\pi, i | = v \text{ equals } n \text{ if } n \text{ is the value of } v \text{ at point } i \text{ on the trace } \pi.$$

With clock and reset signals

$$\pi, i, c, a, r | = v \text{ equals } n \text{ if either } \pi, i, c | = a \text{ or if } \pi, i, c | = !r \text{ and } n \text{ is the value of } v \text{ at point } i.$$

3.6.2.3 The Prime Operator

Let v be a bit vector and n be a constant. The expression $v' \text{ equals } n$ holds at point i on a trace if n is the value of v at point $i+1$.

Without clock and reset signals

$$\pi, i | = v' \text{ equals } n \text{ if } \pi, i+1 | = v \text{ equals } n.$$

With clock and reset signals

$$\pi, i, c, a, r | = v' \text{ equals } n \text{ if either } \pi, i, c | = a \text{ or if } \pi, i, c | = !r \text{ and } n \text{ is the value of } v \text{ at point } i+1.$$

Note that the reset signals are considered only at point i .

3.6.2.4 The Past Operator

Let v be a bit vector, d be a clock and n, m be two constants. The expression $\text{past}(v, m, d)$ equals n holds at a given point on a trace if either:

- there are no tick points of the clock d in the past and $n=0$.

- n is the value of v at tick points of the clock in the past.

Without clock and reset signals

$\pi, i \models \text{past}(v, m, d)$ equals neither if:

- $n = 0$ and there is no $j < i$ such that $\pi, j \models d$.

or

- there exists $j < i$ such that $\pi, j \models d$ and for the maximal such j , $\pi, j \models \text{past}(v, m - 1, d)$ equals n when $m > 1$ or $\pi, j \models v$ equals n when $m = 1$.

With clock and reset signals

$\pi, i, c, a, r \models \text{past}(v, m, d)$ equals n if:

- $\pi, i, c \models a$

or

- $n = 0$ and $\pi, i, c \models !r$, and there is no $j < i$ such that $\pi, j \models d$.

or

- $\pi, i, c \models !r$, and there exists $j < i$ such that $\pi, j \models d$ and for the maximal such j , $\pi, j \models \text{past}(v, m - 1, d)$ equals n when $m > 1$ or $\pi, j \models v$ equals n when $m = 1$.

Note that the reset signals are considered only at point i .

3.6.2.5 Boolean Connectives

Disjunction: The disjunction of two formulas holds if at least one of the formulas holds.

- Semantics without clocks and resets

Let f and g be formulas:

$$\pi, i \models f \vee g \quad \text{if } \pi, i \models f \quad \text{or} \quad \pi, i \models g$$

- Semantics with clocks and resets

Let f and g be formulas:

$$\pi, i, c, a, r \models f \vee g \quad \text{if } \pi, i, c, a, r \models f \quad \text{or} \quad \pi, i, c, a, r \models g$$

Negation: The negation of a formula holds if the formula does not hold.

- Semantics without clocks and resets

Let f be a formula:

$$\pi, i \models !f \text{ iff } \pi, i \not\models f$$

Note: $|\neq$ denotes “does not satisfy”

- Semantics with clocks and resets

Let f be a formula:

$$\pi, i, c, a, r \models !f \text{ iff } \pi, i, c, r, a \not\models f$$

Note that under negation, the roles of **accept** and **reject** are reversed. If this is not done, the definition leads to inconsistency. Assume for a moment that the definition would not have reversed the roles, that is:

$$\pi, i, c, a, r \models !f \text{ iff } \pi, i, c, a, r \not\models f$$

In this case

- If f is satisfied merely because an accept signal was encountered, then $!f$ would also be satisfied for the same reason. This is a contradiction.
- If f is not satisfied merely because a reject signal was encountered, then $!f$ would also not be satisfied for the same reason.

Now, consider the definition with the reset signals reversed:

$$\pi, i, c, a, r \models !f \text{ iff } \pi, i, c, r, a \not\models f$$

Now

- If f is satisfied merely because an accept signal (r) was encountered, then $!f$ will not be satisfied because for $!f$ the reset signal r has taken on the reject roll.
- If f is not satisfied merely because a reject signal (a) was encountered, then $!f$ will be satisfied because for $!f$ the reset signal a has taken on the accept roll.

3.6.3 SemanticsofEventsandExtendedEvents

3.6.3.1 Events

An event holds between ² point i and point j on a trace if point j is a tick point, and there exists a word in the language of the event w such that the following holds:

- the first symbol of the word holds at point i , the second symbol at the next tick point, the third symbol at the following tick point, and so on until the last symbol, which holds at point j .

We say that an event holds at a given point on the trace if there is some future point such that the event holds between the given point and that future point.

In order to formally define the semantics of events, we use the following notation:

$$\pi, i, j, c, a, r \models e$$

This denotes that event e holds between points i and j on the trace, where i is the point at which we start checking the event, and j is the point where it is determined that the event holds.

If c is true and a and r are false, we simply write $\pi, i, j \models e$

Semantics without clocks and resets

We define that an event is satisfied at point i on a trace if there is some point j (where $j \geq i$) such that the event holds between points i and j .

Formally:

$$\pi, i \models e \text{ iff for some } j \geq i \text{ we have } \pi, i, j \models e$$

We now formally define when an event holds between points i and j on the trace. The definition will express that an event holds between two points i and j on a trace if there exists a word in the language of the event such that the first symbol of the word holds at point i , the second symbol at point $i+1$... and the last symbol at point j . Note that the length of such a word is equal to $j - i + 1$.

Formally:

²Whenever we use the term “between”, we use it in the inclusive sense.

$\pi, i, j \models e$ iff there is a word $B = b_0 b_1 \dots b_{j-i} \in L(e)$ such that $\pi, i+m \models b_m$ for $0 \leq m \leq j-i$

With locks and resets

We define that an event is satisfied at point i on a trace if there is some point j (where $j \geq i$) such that the event holds between points i and j .

Formally:

$\pi, i, c, a, r \models e$ iff for some $j \geq i$ we have $\pi, i, j, c, a, r \models e$

We now formally define when an event holds between points i and j on the trace. The definition says that an event holds between two points i and j if there are no reset signals at any point between i and $j-1$, and there exists a word in the language of the events such that either:

- the first symbol in the word holds at point i , the second symbol at the next tick, the third symbol at the following tick, and so on, as long as the tick point is smaller than j , and at point j the accept signal holds.

or

- the first symbol of the word holds at point i , the second symbol at the next tick point, the third symbol at the following tick point, and so on until the last symbol, which holds at point j , and there are no reset signals at point j .

Note that if the number of tick points between $i+1$ and j is m and the length of the word is n , then in the first case we have $n > m+1$ and in the second case we have $n = m+1$.

Formally:

$\pi, i, j, c, a, r \models e$ iff

- $\pi, k, c \models a$ for $i \leq k < j$
- there are precisely $l \geq 0$ tick points $i_1 < \dots < i_l$ of c such that $i_0 < i_1$ and $i_l \leq j$
- there is a word $B = b_0 b_1 \dots b_n \in L(e)$, $n \geq l$ such that $\pi, i_m, c \models b_m$ for $0 \leq m < l$, and either
 - o $\pi, j, c \models a$ and $i_l = j$, or
 - o $\pi, j, c \models a$ and $\pi, i_l, c \models b_l$, or
 - o $l = n$, and $i_n = j$, and $\pi, j, c \models b_n$, and $\pi, j, c \models !r$.

3.6.3.2 ExtendedEvents

$\text{CHANGE_ON}(d)$ changes the clock of the context by which the event is evaluated and causes the event to be evaluated at the nearest tick point of the new clock.

If there is no such tick point, the extended event is not satisfied.

Semantics with locks and resets signals

As with the events, satisfaction of extended events is defined as follows:

$\pi, i, c, a, r \models \text{CHANGE_ON}(d)$ iff for some $j \geq i$ we have $\pi, i, j, c, a, r \models \text{CHANGE_ON}(d)$

We now define when an extended event holds between points i and j on the trace.

This definition will express the following:

The clock in the context is replaced by $[c \rightarrow d]$. If there is no tick of the new clock between i and j , then the extended event is not satisfied. Otherwise, let k be the first tick point of the new clock.

The extended event is satisfied if the event holds between point k and point j .

In order to express that the event is evaluated at the nearest sampling point (k), we define a new event $((!\text{CLOCK})^*, \text{CLOCK})$ which is abbreviated as TICK . Note that TICK holds between points i and k if k is the nearest tick point or nearest accept signal, whichever comes first, and there are no reject signals between point i and point k .

Formally:

$\pi, i, j, c, a, r \models \text{CHANGE_ON}(d)$ iff there exists some $k, i \leq k \leq j$, such that $\pi, i, k, [c \rightarrow d], a, r \models \text{TICK}$ and $\pi, k, j, [c \rightarrow d], a, r \models e$

3.6.4 Semantic of Formulas

3.6.4.1 ACCEPT_ON/REJECT_ON

$\text{ACCEPT_ON}(b)$ and $\text{REJECT_ON}(b)$ change the reset signals of the context by which the formula is evaluated. $\text{ACCEPT_ON}(b)$ changes the accept signal. $\text{REJECT_ON}(b)$ changes the reject signal.

With locks and resets

Formally:

$$\pi, i, c, a, r | = \text{ACCEPT_ON}(b) \text{ iff } \pi, i, c, a | (b \& !r), r | = f$$

$$\pi, i, c, a, r | = \text{REJECT_ON}(b) \text{ iff } \pi, i, c, a, r | (b \& !a) | = f$$

The new reset signals in both cases assure that the reset signals will be mutually exclusive and that *b* will have lower precedence than *a* and *r*.

3.6.4.2 SEQ

*e*SEQ holds at a given point on the trace if the extended event *e* holds between the nearest tick point and some future point *j* and *f* holds at the tick point that is nearest to point *j*.

Without locks and resets

*e*SEQ holds at point *i* on the trace if the event *e* holds between point *i* and some future point *j*, and *f* holds at point *j*.

Formally:

$$\pi, i | = e \text{SEQ} \text{ iff for some } j \geq i \text{ we have that } \pi, i, j | = e, \text{ and } \pi, j | = f.$$

With locks and resets

*e*SEQ holds either if:

The event *e* holds between the nearest tick point and some future point *k* and *f* holds at the nearest tick point *l* following point *k*, and there are no reset signals (accept or reject) before point *l*.

or

there is an accept signal that is not preceded by a reject signal, and the accept signal is such that either it occurs before or at the nearest tick, or it occurs after the nearest tick point and some prefix of a word *w* in the language of the event holds between the nearest tick point and some point before the accept signal.

Formally:

$$\pi, i, c, a, r | = e \text{SEQ} \text{ iff for some } l \geq k \geq j \geq i \text{ we have that } \pi, i, j, c, a, r | = \text{TICK}, \pi, j, k, c, a, r | = e, \pi, k, l, c, a, r | = \text{TICK}, \text{ and } \pi, l, c, a, r | = f.$$

3.6.4.3 TRIGGERS

e TRIGGERSf holds if, for every word in the language of the event e , if the word holds between the nearest tick point and some future point j , then f holds at the tick point nearest to that point.

Without locks and resets

e TRIGGERSf holds if, for every word in the language of the event e , if the word holds between point i and some future point j , then f holds at point j .

Formally:

$\pi, i | = e$ TRIGGERSf iff for all $j \geq i$ if we have that $\pi, i, j | = e$, then $\pi, j | = f$.

With locks and resets

e TRIGGERSf holds iff for every word in the language of the event e either:

The word holds between the nearest tick point and some future point k and f holds at a tick point l nearest to point k , and there are no reset signals (accept or reject) before point l .

or

the word does not hold between the nearest tick point and any future point

or

there is an accept signal that is not preceded by a reject signal, and the accept signal is such that either it occurs before or at the nearest tick, or it occurs after the nearest tick point and some prefix of the word holds between the nearest tick point and some point before the accept signal.

Note that if the event does not hold, then e TRIGGER Sf holds vacuously.

Formally:

$\pi, i, c, a, r | = e$ TRIGGERSf iff for all $k \geq j \geq i$ such that we have that $\pi, i, j, c, r, a | = \text{TICK}$, and $\pi, j, k, c, r, a | = e$ and $\pi, k, l, c, r, a | = \text{TICK}$, we have that $\pi, j, c, a, r | = f$.

Note that the roles of a and r are reversed in $\pi, i, j, c, r, a | = \text{TICK}$, and $\pi, j, k, c, r, a | = e$ and $\pi, k, l, c, r, a | = \text{TICK}$.

3.6.4.4 NEXT/WNEXT

NEXTf holds at a given point on the trace iff it holds at the next tick point. WNEXTf is satisfied under the same conditions as NEXTf except in the case in which there is no future tick point. In this case WNEXTf is satisfied whereas NEXTf is not.

Without locks and resets

Both NEXTf and WNEXTf hold at point i on the trace iff they hold at the next point on the trace, that is, at point $i+1$. Note that in this case there is no difference between NEXTf and WNEXTf because a future tick point exists.

Formally:

$$\pi, i \models \text{NEXT}f \text{ iff } \pi, i+1 \models f$$

$$\pi, i \models \text{WNEXT}f \text{ iff } \pi, i+1 \models f$$

With locks and resets

NEXT

NEXTf holds either if:

there is an accept signal, not preceded by a reject signal, before the next tick point

or

there are no reset signals (accept or reject) before the next tick point, and f holds at that tick point.

Note that a necessary condition for the satisfaction of this formula is that there has been a clock tick or an accept signal.

Formally:

$$\pi, i, c, a, r \models \text{NEXT}f \text{ iff either } \pi, i, c \models a \text{ or } \pi, i, c \models !r \text{ and for some } j \geq i+1 \text{ we have that } \pi, i+1, j, c, a, r \models \text{TICK} \text{ and } \pi, j, c, a, r \models f.$$

WNEXT

WNEXTf holds at a particular point on a trace if:

the next tick point does not exist and there is no future reject signal

or

the next tick point exists and one of the following holds: there is an accept signal before that tick point with no reject signal preceding it, or there is no reject signal before that clock tick and f holds at that clock tick.

Formally:

$\pi, i, c, a, r \models \text{WNEXT}f$ if either $\pi, i, c \models a$ or $\pi, i, c \models \neg r$ and for all $j \geq i+1$ we have that if $\pi, i+1, j, c, r, a \models \text{TICK}$, then $\pi, j, c, a, r \models f$.

3.6.4.5 EVENTUALLY

EVENTUALLYf holds at a given point on the trace iff f holds at some future tick point.

Without clocks and resets

EVENTUALLYf holds at point i on a trace if at point i or at some point j in the future f holds.

Formally:

$\pi, i \models \text{EVENTUALLY}f$ iff for some $j \geq i$ we have $\pi, j \models f$

With clocks and resets

EVENTUALLYf holds either if:

there are no reset signals (accept or reject) before some future tick point where f holds

or

there is an accept signal, not preceded by a reject signal or a tick point where f holds.

Formally:

$\pi, i, c, a, r \models \text{EVENTUALLY}f$ iff for some $j \geq i$ the following holds:

- $\pi, j \models \text{cor}$ $\pi, j, c \models a$, and
- $\pi, j, c, a, r \models f$, and

- for $i \leq k < j$ we have $\pi, k, c \models r$.

3.6.4.6 ALWAYS

ALWAYS f holds at a given point on a trace iff f holds at every tick point in the future.

Without locks and resets

ALWAYS f is satisfied at point i iff f holds at every point $j \geq i$.

Formally:

$\pi, i \models \text{ALWAYS } f$ iff for every $j \geq i$ we have $\pi, j \models f$

With locks and resets

ALWAYS f holds either if:

there are no reset signals (accept or reject) and f holds at every future tick point

or

there is an accept signal, not preceded by a reject signal at some future point, and f holds at every future tick point until that point.

Note that if there are no reset signals and no future tick points, ALWAYS f holds vacuously.

Formally:

$\pi, i, c, a, r \models \text{ALWAYS } f$ iff for every $j \geq i$, if $\pi, j \models \text{cor}$ or $\pi, j, c \models r$, then either:

- $\pi, j, c, a, r \models f$, or
- for some $i \leq k < j$ we have $\pi, k, c \models a$.

3.6.4.7 UNTIL/WUNTIL

$f \text{ UNTIL } g$ holds at a given point iff f holds at every tick point until some tick point where g holds. $f \text{ WUNTIL } g$ is satisfied under the same conditions as $f \text{ UNTIL } g$ except in the case in which there is no tick point where g holds and f holds at every tick point. In this case $f \text{ WUNTIL } g$ is satisfied whereas $f \text{ UNTIL } g$ is not.

Without locks and resets

f UNTIL g

f UNTILgissatisfied at a given point on a trace if g holds at some future point and f holds at every point until the point before the point where g holds. Note that f UNTILgissatisfied at a given point if g holds at that point.

Formally:

$\pi, i \models f$ UNTILg iff for some $j \geq i$ we have $\pi, j \models g$ and for every $i \leq k < j$ we have $\pi, k \models f$

fWUNTILg

f WUNTILgissatisfied at point i iff f UNTILg holds or ALWAYSf holds.

Formally:

$\pi, i \models f$ WUNTILg iff either $\pi, i \models f$ UNTILg or $\pi, i \models \text{ALWAYS}f$.

With locks and resets

fUNTILg

f UNTILg holds either if:

f holds at every tick point until a tick point where g holds and there are no reset signals (accept or reject) before that point.

or

there is an accept signal not preceded by a reject signal or a tick point where g holds, and f holds at every tick point until the point where the accept signal holds.

Note that a necessary condition for the satisfaction of this formula is that there has been a lock tick where g holds or an accept signal.

Formally:

$\pi, i, c, a, r \models f$ UNTILg iff for some $j \geq i$, we have both

- $\pi, j \models c$ or $\pi, j, c \models a$, and
- $\pi, j, c, a, r \models g$ and for $i \leq k < j$ we have
 - $\pi, k, c \models !r$ and
 - if $\pi, k \models c$, then $\pi, k, c, a, r \models f$.

fWUNTILg

$f \text{WUNTIL}_g$ is satisfied at point i iff UNTIL_g holds or ALWAYS_f holds.

$\pi, i, c, a, r \models f \text{WUNTIL}_g$ if either $\pi, i, c, a, r \models f \text{UNTIL}_g$ or $\pi, i, c, a, r \models \text{ALWAYS}_f$.

3.6.4.8 CHANGE_ON/CHANGE_IF

$\text{CHANGE_ON}(d)$ and $\text{CHANGE_IF}(d)$ change the clock of the context by which the formula f is evaluated and cause the formula to be evaluated at the nearest tick point of the new clock.

$\text{CHANGE_ON}(d)$ is satisfied under the same conditions as $\text{CHANGE_IF}(d)$ except when there is no future tick point. In this case $\text{CHANGE_IF}(d)$ is satisfied whereas $\text{CHANGE_ON}(d)$ is not.

With clocks and reset signals

CHANGE_ON(d)f

$\text{CHANGE_ON}(d)$ changes the clock of the current context (c) to $[c \rightarrow d]$, and evaluates f at the nearest tick point of this new clock.

$\text{CHANGE_ON}(d)$ holds either if:

f holds at the nearest tick point of the new clock, and there are no reject signals (acceptor or reject) preceding that point

or

there is an accept signal not preceded by a reject signal or a tick point of the new clock.

Note that a necessary condition for the satisfaction of this formula is that there has been a tick point of the new clock or an accept signal.

Formally:

$\pi, i, c, a, r \models \text{CHANGE_ON}(d)f$ if there exists some $j \geq i$, such that $\pi, i, j, [c \rightarrow d], a, r \models \text{TICK}$ and $\pi, j, [c \rightarrow d], a, r \models f$

CHANGE_IF(d)f

$\text{CHANGE_IF}(d)$ changes the clock of the current context (c) to $[c \rightarrow d]$, and evaluates f at the nearest tick of this new clock.

$\text{CHANGE_IF}(d)$ holds either if:

f holds at the nearest tick point of the new clock, and there are no reset signals (accept or reject) preceding that point

or

there is an accept signal not preceded by a reject signal or a tick point of the new clock.

or

there are no future tick points and no reset signals.

Note that the only case where CHANGE_ON and CHANGE_IF yield different results is when there are no reset signals and no future tick points.

Formally:

$\pi, i, c, a, r \models \text{CHANGE_IF}(d)$ iff, whenever $j \geq i$ such that $\pi, i, j, [c \rightarrow d], r, a \models \text{TICK}$, then $\pi, j, [c \rightarrow d], a, r \models f$

Note the reversal of a and r in the antecedent. Note also that there can be at most one $j \geq i$ such that $\pi, i, j, [c \rightarrow d], r, a \models \text{TICK}$.

3.6.5 Temporal Intervals

Temporal intervals modify the temporal operators NEXT, WNEXT, EVENTUALLY, UNTIL, WUNTIL and ALWAYS, by specifying intervals in which certain events are expected to occur.

3.6.5.1 NEXT/WNEXT

$\text{NEXT}[m]$ f holds at a given point on the trace iff f holds at them m -th tick in the future.

$\text{WNEXT}[m]$ f is satisfied under the same conditions as $\text{NEXT}[m]$ f except in the case in which there is no m -th future tick point. In this case $\text{WNEXT}[m]$ f is satisfied whereas $\text{NEXT}[m]$ f is not.

Without clocks and resets

Both $\text{NEXT}[m]$ f and $\text{WNEXT}[m]$ f hold at point i on the trace iff f holds at point $i+m$ on the trace. Note that in this case there is no difference between $\text{NEXT}[m]$ f and $\text{WNEXT}[m]$ f because there is a future m -th tick.

Formally:

$\pi, i | = \text{NEXT}[m]f$ if $\pi, i+m | = f$

$\pi, i | = \text{WNEXT}[m]f$ if $\pi, i+m | = f$

With locks and resets

$\text{NEXT}[m]f$ holds at a given point on the trace if the formula $\text{NEXT} \dots \text{NEXT}f$, where NEXT is iterated m times, holds at that point.

Likewise, $\text{WNEXT}[m]f$ holds at a given point on the trace if the formula $\text{WNEXT} \dots \text{WNEXT}f$, where WNEXT is iterated m times, holds at that point.

Formally:

$\pi, i, c, a, r | = \text{NEXT}[m]f$ if $\pi, i, c, a, r | = \text{NEXT} \dots \text{NEXT}f$, where NEXT is iterated m times.

$\pi, i, c, a, r | = \text{WNEXT}[m]f$ if $\pi, i, c, a, r | = \text{WNEXT} \dots \text{WNEXT}f$, where WNEXT is iterated m times.

3.6.5.2 EVENTUALLY

$\text{EVENTUALLY}[m, n]f$ holds at a given point on a trace iff f holds at some tick point between the next m -th and n -th tick points.

Without locks and resets

$\text{EVENTUALLY}[m, n]f$ holds at point i on the trace iff f holds at some point between point $i+m$ and $i+n$.

Formally:

$\pi, i | = \text{EVENTUALLY}[m, n]f$ iff for some j where $i+m \leq j \leq i+n$, $\pi, j | = f$.

With locks and resets

$\text{EVENTUALLY}[m, n]f$ is defined in two steps. The first step defines the semantics of $\text{EVENTUALLY}[m, n]f$ when m equals 0, and the second step defines the semantics when m is greater than 0.

The formula $\text{EVENTUALLY}[0, n]f$ holds at a given point on a trace either if

f holds at some tick point j such that there are at most n tick points between the point following the given point and point j , and there are no future reset signals preceding point j .

or

there is some future point j , not preceded by a reject signal or a tick point where f holds, where the accept signal holds and there are at most n tick points between the point following the given point and point j (inclusive).

The formula $\text{EVENTUALLY}[m,n]f$, for $m > 0$, holds at a given point on a trace if there is at least one future reset point or at least one tick point following the given point, and the formula $\text{EVENTUALLY}[m-1,n-1]f$ holds at the nearest tick point following the given point or the nearest point where a reset signal (accept or reject) holds, whichever comes first. Note that the tick point must be after the given point, whereas the reset signal can occur at or after the given point. Note also that if at that nearest point the accept signal holds, then the formula is accepted, and if at that point the reject signal holds the formula is rejected.

Formally,

$\pi, i, c, a, r \models \text{EVENTUALLY}[0,n]f$ iff for some $j \geq i$ we have that:

- there are at most n tick points between $i+1$ and j , and
- $j=i$, or $\pi, j \models \text{cor}$ $\pi, j, c \models a$, and
- $\pi, j, c, a, r \models f$, and
- for $i \leq k < j$ we have $\pi, k, c \models !r$.

$\pi, i, c, a, r \models \text{EVENTUALLY}[m,n]f$, for $m > 0$, if there is at least one future reset point or at least one tick point following point i and $\pi, j, c, a, r \models \text{EVENTUALLY}[m-1,n-1]f$, where j is the nearest reset point or the first tick point following point i , whichever comes first.

3.6.5.3 ALWAYS

$\text{ALWAYS}[m,n]f$ holds at a given point on a trace iff f holds at every tick point between the next m -th and n -th tick points.

Without clocks and resets

$ALWAYS[m,n]f$ holds at point i on the trace iff f holds at every point between $i+m$ and $i+n$.

Formally:

$\pi, i \models ALWAYS[m,n]f$ iff for every j where $i+m \leq j \leq i+n$, $\pi, j \models f$.

With locks and resets

$ALWAYS[m,n]f$ is defined in two steps. The first step defines the semantics of $ALWAYS[m,n]f$ when m equals 0, and the second step defines the semantics when m is greater than 0.

The formula $ALWAYS[0,n]f$ holds at a given point on a trace either if

f holds at every tick point j such that there are at most n tick points between the point following the given point and point j , and there are no reset signals between the given point and point j .

or

there is some future point j , such that there are at most n tick points between the point following the given point and point j , and the accept signal holds at that point and there are no reset signals between the given point and point j .

The formula $ALWAYS[m,n]f$, for $m > 0$, holds at a given point on a trace either if

there are no future reject signals and no tick points following the given point.

or

there is at least one future reset point or at least one tick point following the given point and the formula $ALWAYS[m-1, n-1]f$ holds at the nearest tick point following the given point or the nearest point where a reset signal (accept or reject) holds, whichever comes first. Note that if at point i the nearest point where the accept signal holds is j , then the formula is accepted, and if at that point j there is a reject signal then the formula is rejected.

Formally,

$\pi, i, c, a, r \models ALWAYS[0,n]f$ iff for all $j \geq i$ such that there are at most n tick points between $i+1$ and j , and either $j=i$, or $\pi, j \models c$ or $\pi, j, c \models r$, we have that:

- $\pi, j, c, a, r \models f$, or

- for some $i \leq k < j$ we have $\pi, k, c \models a$.

$\pi, i, c, a, r \models \text{ALWAYS}[m, n]f$, for $m > 0$, if

- for all $j \geq i$, $\pi, j, c \models !r$ and for all $j > i$, $\pi, j \models !c$

or

- there is at least one future reset point or at least one tick point t after point i and $\pi, j, c, a, r \models \text{ALWAYS}[m-1, n-1]f$, where j is the nearest reset point or the first tick following point i , whichever comes first.

3.6.5.4 UNTIL/WUNTIL

$f\text{UNTIL}[m, n]g$ holds at a given point on a trace if g holds at some tick point j between the m -th tick point and then the n -th tick point, and f holds at every tick point from the m -th tick point until the tick point before the tick point where g holds. $fW\text{UNTIL}[m, n]g$ is satisfied under the same conditions as $f\text{UNTIL}[m, n]g$ except in the case in which there is no tick point between the m -th and n -th tick points where g holds and f holds at every tick point between those tick points. In this case $fW\text{UNTIL}[m, n]g$ is satisfied whereas $f\text{UNTIL}[m, n]g$ is not.

Without clocks and resets

$f\text{UNTIL}[m, n]g$

$f\text{UNTIL}[m, n]g$ is satisfied at point i if g holds at some point between points $i+m$ and $i+n$, and f holds at every point from point $i+m$ to the point before the point where g holds.

Formally:

$\pi, i \models f\text{UNTIL}[m, n]g$ iff for some j where $i+m \leq j \leq i+n$, $\pi, j \models g$, and for every $i+m \leq k < j$ we have $\pi, k \models f$.

$fW\text{UNTIL}[m, n]g$

$fW\text{UNTIL}[m, n]g$ is satisfied at point i iff $f\text{UNTIL}[m, n]g$ holds or $\text{ALWAYS}[m, n]f$ holds.

Formally:

$\pi, i \models fW\text{UNTIL}[m, n]g$ iff either $\pi, i \models f\text{UNTIL}[m, n]g$ or $\pi, i \models \text{ALWAYS}[m, n]f$.

Without clocks and resets

fUNTIL[m,n]g

fUNTIL[m,n]g is defined in two steps. The first step defines the semantics of fUNTIL[m,n]g when m equals 0, and the second step defines the semantics when m is greater than 0.

The formula fUNTIL[0,n] g holds at a given point on a trace if

g holds at point i (even in the case that i is not a tick point).

or

f holds at every tick point until some tick point j where g holds and there are at most n tick points between the point following the given point i and point j, and if g does not hold at point i then f holds at that point, and there are no future reset signals preceding point j.

or

there is some future point j at which there is an accept signal and that point is not preceded by a future reject signal i or future tick points where f does not hold, and if j > i then f holds at point i (even in the case that i is not a tick point), and point j is such that there are at most n tick points between the point after the given point and that point.

The formula fUNTIL[m,n]g, for m > 0, holds at a given point on a trace if there is at least one future reset point or at least one tick point following the given point and the formula EVENTUALLY[m-1,n-1]f holds at the nearest tick point following that given point or the nearest point where a reset signal (accept or reject) holds, whichever comes first. Note that if at point i that nearest point the accept signal holds then the formula is accepted, and if at that point the reject signal holds the formula is rejected.

Formally,

$\pi, i, c, a, r \models \text{fUNTIL}[0, n]g$ iff for some $j \geq i$ we have that

- there are at most n tick points between i+1 and j, and
- $j=i$, or $\pi, j \models \text{cor}$ $\pi, j, c \models a$, and
- $\pi, j, c, a, r \models g$ and for $i \leq k < j$ we have
 - $\pi, k, c \models !r$, and

- if $k=i$ or $\pi, k| = c$, then $\pi, j, c, a, r| = f$.

$\pi, i, c, a, r| = \text{ALWAYS}[m, n]f$, for $m > 0$, if there is at least one future reset point or at least one tick point following π, i and $\pi, j, c, a, r| = \text{fUNTIL}[m-1, n-1]g$, where j is the nearest reset point or the first tick point following i , whichever comes first.

fWUNTIL[m, n]g

fWUNTIL[m, n]g is satisfied at point i iff $\text{fUNTIL}[m, n]g$ holds or $\text{ALWAYS}[m, n]f$ holds.

Formally:

$\pi, i, c, a, r| = \text{fWUNTIL}[m, n]g$ if either $\pi, i, c, a, r| = \text{fUNTIL}[m, n]g$ or $\pi, i, c, a, r| = \text{ALWAYS}[m, n]f$.

3.7 Integers

3.7.1 IntegerConstants

| Semantics | Syntax |
|---------------------------|----------------|
| Integer(decimal)constants | $[1-9][0-9]^*$ |

3.7.2 IntegerArithmeticandIntegerExpressions

Let n be an integer constant and let x, y be integer expressions. Then the following are also integer expressions:

| Semantics | Syntax |
|---------------------|---------|
| An integer constant | n |
| Addition | $x + y$ |
| Subtraction | $x - y$ |
| Multiplication | $x * y$ |
| Quotient | x / y |

3.7.3 sizeofFunction

Let x be a bit n -vector.

| Semantics | Syntax |
|--|--------------------|
| Return an integer which is equal to the size (width) of the bit n -vector. | $\text{sizeof}(x)$ |

3.8 Enumerated types

The syntax to define a new enumerated type is as follows:

TypeName := **type**{ value_1, value_2, ..., value_n };

3.8.1 Enumerated types: Operators

| Semantics | Syntax |
|--|-----------------------|
| Increment to next value in type. Incrementing the last value recycles to the first value. | inc(x) |
| Decrement to previous value in type. Decrementing the first value results in the last value. | dec(x) |
| The value of x in the next phase | x' |
| Comparison operators: All comparison operators, as in section 3.1.6. Comparison is only possible between two elements of the same enumerated type. | $=, !=, <, <=, >, >=$ |

There is no built-in conversion (casting) from an enumerated type to any other type, but one can supply such a conversion using a case statement.

For example, assume we have a FSM in the design `-under-test` that we want to build a reference model for:

```
state:=/dut/FSM/state;//referencetothemodelstate.
StateTP:=type{s1,s2,s3,s4,s5};//declarationofenumeratedtype
//userwrittenconversion fromStateTPvaluestointegersusingcasestatement
toBV(st):=casest{
    s1:0;
    s2:1;
    s3:2;
    s4:3;
    default:4
};
StateTPs;//declarationofvariableoftypeStateTP
Inits=s1;//initializationofvariable
//nextstateassignmenttofs
Assigns'=case{
    s=s1&/dut/a:s3;
    s=s1&/dut/b:s4;
    .
    .
    .
    default:s5;
};
//checkstatefromthemodelcorrespondstotheuser'sreferencemodel.
assertalways(state==toBV(s));
```

4 Declaration of Specification Variables

The declaration of state variables has the following syntax:

```
bit b1,b2;  
bit[n] data1,data2;
```

Where n is an integer expression. State variables can't have a definition (see below). By default, specification state variables are free (can get any value at any time). To restrict the behavior of specification state variables one can use assumptions (or any other type of modeling provided in the language). For example:

```
assume always b'=(!b);
```

Rigid variables are variables which don't change their value after initialization (`always x'=x`).

They are declared by the keyword `rigid`:

```
rigid bit b1,b2;  
rigid bit [n] b3,b4;
```

5 Templates and Definitions

5.1 Definitions

A definition has the following form:

```
[bit|event|formula|int] name := formula;
```

Examples:

```
f := always (req -> eventually gnt); // OK
```

```
f := assert always !vec[2:0]; //Error - not a formula
```

When a type is not set explicitly, the definition type is determined by the formula on the right hand side. For example, if `in1` and `in2` are Booleans and `e` is defined as

```
e := in1 & in2;
```

Then `e` is also Boolean. If `in1` was a formula, then `e` would be a formula too.

Definitions are also used for Block aliasing, see section 6.2 below. For example:

```
b := block1(x, y);
```

In this case `b` is a block definition which needs to be instantiated.

5.2 Templates

A template definition is similar to the regular definition described in the previous section. The difference is that a template receives parameters, which are used in the defined expression.

A template definition has the following form:

```
[type] name ([type] p1, ... , [type] pn) := formula;
```

Where [type] stands for [bit|event|formula|int].

Mapping actual parameters to formal parameters is performed by argument order (not by name).

5.2.1 Template Examples

The following is a simple template definition:

```
eventually_first(x) := ((!x)*,x);
```

The usage of this template may have the following form:

```
tf := ((b1 & b2)*, !b1) triggers eventually_first(i2);
```

One can also use integer parameters:

```
req_ack(n,r,a) := r{3} triggers next[n] a;  
tf := req_ack(4,x,y);
```

One can use expressions as actual arguments.

```
tf := req_ack(4, r1|r2, !f);
```

5.2.2 Comma as Parameter Delimiter vs. Events Concatenation

Notice that since commas are used both for regular expression concatenation and for parameter separation, it is possible that there will be an ambiguity. The expression

```
tf := req_ack(4, x, y, z);
```

can be interpreted as

```
tf := req_ack(4, (x, y), z);
```

or as

```
tf := req_ack(4, x, (y, z));
```

To solve this, it is not allowed to give events that contain commas as parameters without putting them within parenthesis:

```
tf2 := req_ack(4, x, (y, z));
```

6 Blocks&Scoping

ForSpec blocks can group any number of ForSpec statements, including declarations, definitions, assignments, assertions, assumptions, mappings and nested blocks.

Blocks are instantiated using the **new** keyword. A block may have several instances. Optionally, blocks can receive parameters just as ForSpec templates (see section 5 above).

6.1 Block Definition

A block definition has the following syntax:

```
blockName[( param1, ..., paramn )]:=
{
    [ForSpecStatements]*
};
```

| | |
|------------------------------------|--|
| <i>blockName</i> | A valid identifier. |
| <i>param1</i> , ..., <i>paramn</i> | An optional parameters list separated by commas. Each formal parameter is a valid identifier. When the block is instantiated, each actual parameter is a valid formula. |
| [ForSpecStatements]* | Any number of valid statements (separated by semicolons). This includes nested blocks definition and blocks instantiation. |

6.2 Aliasing

Once a block is defined, an alias can be created using the regular ForSpec syntax for definitions:

```
alias_name := blockname ;
```

For example:

```
block1 := { ... };  
b1 := block1;
```

b1 is a block definition and can thus be instantiated.

Aliasing can also be used to set any number of block parameters, for example:

```
block1 (f, g) := { ... };  
b1 := block1 ( always f, true);  
b2 (a) := block1 (a, true);
```

Notice once again that b1 and b2 are block definitions, they are neither regular ForSpec definitions nor templates. Therefore:

```
assert b2 ( /x );      // illegal  
new b2 ( /x );        // legal
```

6.3 BlockInstantiation

Block instantiation has the following syntax:

```
[ instName := ] new blockName [(param1, ..., paramn)];
```

| | |
|---------------------|--|
| <i>instName</i> | The block instance name which is a valid identifier and must be preceded by a colon -equal(:=). |
| new | Keyword used for block instantiation. |
| param1, ..., paramn | The <i>actual parameter</i> list. The list must match the <i>formal parameters</i> of the block definition. Actual parameters are matched to the formal parameters by order of appearance. |

Block statements are inactive until the block is instantiated. For example:

```
block1 := {  
    x := /dut/x;  
    assert x = 0;  
    block2 := {  
        f := !x;  
    }  
}
```

Although `block1` was defined, none of its statements are active at this point. The assertion is not performed, `x` and `block2` are undefined and cannot be referenced. Notice however that the block's statements *are* compiled and error messages are generated when required.

Now let's instantiate `block1`:

```
inst1 := new block1; // inst1 is the instance name
```

At this point the assertion is active and both `x` and `block2` are defined. The following code is therefore legal (See 6.4 below for more information on scoping):

```
g := ! inst1/x; // The instance name is used for  
new inst1/block2; // referencing the block's statements
```

Notice that the instance name is optional. Therefore block1 from the example above can also be instantiated in the following manner:

```
new block1;
```

At this point the assertion is performed and the definitions are reactive, but they can't be referenced since for that an instance name is required.

6.4 Scoping

Every ForSpec identifier is defined within a scope. Identifiers created at the topmost level (not within a block) are created in scope/(the scope delimiter). Identifiers defined within blocks are created in sub -scopes according to the instance name. Block definitions on the other hand, do not create sub -scopes.

For example, the following statement defines a new identifier f.:

```
f := always g;
```

If the statement is placed in the outmost scope (not within a block), then the fully qualified ID of f is f/(the same as in the Unix file system when a file is located at the root). If on the other hand we have the following code:

```
b11 := {  
    f := always g;  
}  
bi1 := new b11;
```

Then f is defined within the block instance bi1. Assuming that bi1 is in the outmost scope, then its ID is bi1 and the ID of f is bi1/f.

Typically, a ForSpec statement refers to identifiers other than the one that it may define. There are two ways in which we can refer to other identifiers:

- (1) By giving their fully qualified ID;
- (2) By giving their relative ID;

If a fully qualified ID is given, then an identifier with that exact ID is referenced. A fully qualified ID is one that begins with the scoped delimiter (/) symbol. If the exact identifier is not found, an error message is reported.

For instance: /topbi/bi1/bi2/f1 refers to an identifier f1 defined in the scope /topbi/bi1/bi2. This again resembles the way fully qualified filenames are interpreted in Unix.

If a relative ID is given (one that does not begin with a scoped delimiter) then an identifier with that ID is referenced relative to the scope in which the statement is placed. If that ID is not defined in that scope, then it is searched for in an outer scope.

For example:

```
b11 := {
    bit a;
    f := always a;
}
b12 := {
    bit g;
    f := always g;
    b11 := new b11;
}
bi2 := new b12;
```

The IDs that were redefined here are: /b11, /b12, /bi2, /bi2/g, /bi2/f, /bi2/bi1/bi2/bi1/a and /bi2/bi1/f. The scopes that were redefined are the outermost scope, /bi2 and /bi2/bi1.

6.4.1 Referencing the DUT

DUT signals are referenced using the same scoping mechanism. For example:

```
f := always/block_inst/g ->/dut/x;
```

However, mappings to the DUT cannot use relative IDs - they must begin with a scoped delimiter (/). Notice that a DUT signal might have the same ID as a spec identifier. For example, suppose there is a DUT signal with ID /top/f1/x and suppose we have the following code:

```
b1 := {  
    b2 := {  
        bit x;  
    }  
    f1 := new b2;  
}  
top := new b1;  
assert /top/f1/x;
```

Does /top/f1/x refer to the DUT signal or the spec variable? The answer is that spec variables are resolved first. Since this might cause confusion, it is recommended to use relative IDs when referencing spec variables (if possible).

One should also notice that the following code (in the outermost scope) is illegal:

```
x := /x;
```

It is illegal since `x` and `/x` are identical (the ID of the LHS is also `/x`).

6.5 BlockExamples

6.5.1 Example1

```
bit x,y;
block1 :=
{
    bit x;
    f := always mutex (x%y);
}
i1 := new block1;
assert !x | i1/f;
```

6.5.2 Example2

```
block2 (w, s):=
{
    bit [w] buff;
    init buff = 0;
    assign buff' = f(s) ;
};

out := new block2 (8, /f1/out[7:0]);
in  := new block2 (8, /f1/in[7:0]);
assert (in/buff=0x0F) -> next[5] (out/buff=0x1A);
```

6.5.3 Example3

```
bit x,y;
block1:= {
    bit y;
    block2 := {
        bit z;
        assert (x=y) | (x=z);
    }
    new block2;
}
new block1;
```

6.5.4 Example4

```
block1(a,b) :=
{
    f := always a until b;
}

g(m) := block1(true, m);
h     := block1(/b1/x, /b1/y);

i1 := new g(/b1/z);
i2 := new h;
assert i1/f & i2/f;
```

7 Assignments

Assignments include **assign**, **assign_on** and **init** keywords and are used for modeling. They have the following advantages over modeling with assumptions:

- Assignments are more efficient
- Assignments are “safer” – they cannot restrict the behavior of the DUT
- Assignments give a clear modeling syntax

On the other hand, assignments are more limited than assumptions. Assignments have restrictions – SAR, no cycles and Timing – are described below.

7.1 Assignments Syntax

```
init x = e;
```

```
assign x = e;
```

```
assign x' = f;
```

```
assign_on(clk) x' = f;
```

| | |
|---------------|--|
| <i>X</i> | Name of a state variable (the state variable needs to be declared using the bit keyword prior to the assignment). |
| <i>e</i> | A current bit -vector. |
| <i>f, clk</i> | A current or next bit -vector. |

`assign_on(clk) x' = f` is equivalent to `assign x' = (clk) : [clk → f] ? x` where `[clk → f]` is the bit vector expression that results from substituting each occurrence of the keyword **CLOCK** iff by `clk`. Note that if `f` does not contain the keyword **CLOCK** then `f = [clk → f]`.

7.2 SAR

Init and assign obey SAR (single assignment rule), i.e. there is at most one assignment per variable.

The following example is illegal:

```
bit x;  
assign x = ... ;  
assign x = ... ;
```

The following example is also illegal:

```
bit x;  
assign x = ... ;  
assign x' = ... ;
```

The first assignment determines the value of x from time 0 onwards. The second assignment determines the value of x from time 1 onwards and thus conflicts with the previous assignment.

On the other hand the following example is legal:

```
bit x;  
init x = ... ;  
assign x' = ... ;
```

It is legal since init only determines the value of x at time 0, and the assign determines the value of x from time 1 onwards.

7.3 Assignments Timing (Current -Next)

The use of prime (') in the RHS of assignments and its use is restricted, as described in the following rules:

- If the LHS is current, then the RHS must be current too.
- If the LHS is primed, the RHS may be a mixture of current and next expressions.

Notice that `past` can be used in the RHS of an assignment.

For example:

```
bit x,y;
assign x = y'; // Illegal - RHS must be current
```

The RHS may also include the `past` operator. For example:

```
assign x' = past(x'); // OK - no cycle and a single prime
```

7.4 Assign-Cycles

Assign-cycles are forbidden. Assign -cycles occur when a state -variable appears in both the LHS and the RHS, in the same time.

For example, the following assignments are illegal:

```
bit x,y;  
assign x = x;  
assign y' = y';
```

While the following is legal:

```
bit x,y;  
assign x' = x;
```

Indirect assign -cycles, as in the example below, are forbidden as well:

```
bit x,y,z;  
assign x = y;  
assign y' = z';  
assign z = x;
```

8 Assumptions/Assertions

8.1 Assume, Assert, Restrict & Model

This section covers `assume`, `assert`, `restrict` and `model` keywords.

These directives are interpreted differently for Formal Verification (FV) and for dynamic verification (DV), as described in the following table:

| | Usage | FV | DV |
|-----------------|--|---------------|---------------|
| Assert | Checkers. | FV checker | DV checker |
| Assume | Set “legal” constraints that should be verified (with a proof obligation). | FV assumption | DV checker |
| Restrict | In FV restrictions are properties with no proof obligation, their purpose is to deal with capacity problems. In DV, where there are no capacity problems to deal with, restrictions are ignored. | FV assumption | Ignored |
| Model | Modeling by assumptions. Assignments are a safer way for modeling and should be used whenever possible. Models should be used with care, since they can restrict the DUT. | FV assumption | DV assumption |

The `assume` and `assert` keywords are used to specify assumptions and assertions:

```
assert[ [bit[n] | event | formula | int] name:=] expression;
assume[[ bit[n] | event | formula | int] name:=] expression;
```

Examples:

```
tf1:=always eventually x;  
assume tf1;
```

Integrating a typed definition into an assume/assert statement:

```
assume formula tf1:=always eventually x;
```

Unnamed assumption/assertion:

```
assume always eventually x;
```

9 External Signals and Mapping

External signals (external to ForSpec, e.g. from the Design Under Test) are referenced via hierarchical names, beginning with "/" or ".". For example: `/x/in`. One can use the `sub` -vector operator to specify the size of external `bit` -vectors. For example, `/a/b/x` of the DUT is of size 16:

```
/a/b/x[15,0]
```

In the similar way, one can extract `sub` -vectors from external vectors:

```
/a/b/x[3,1]
```

External signals can be referenced directly or be given new names. The mapping of new names to external signals is done via the usual definition mechanism:

```
in := /a/b/in;  
y := /a/b/x[8,3];
```

In the last example, `y` is a `bit` -vector of size 6, connected to the corresponding bits of `x`. One can directly specify the size of `y` in the last example:

```
bit[6] y := /a/b/x[8,3];
```

ForSpec allows integer expressions as part of a reference to a signal of the design under test. This way one can build parameterized signal names. The syntax uses the `DOT` as the concatenation operator of strings as in Perl. The syntax definition is as follows:

Let `text_in_quote` stand for any text inside double quotes and let `int_exp` stand for an integer expression. Then a reference `string` is given by

```
reference_string = text_in_quote(DOT(text_in_quote|int_exp))*
```

Since a reference `string` must be a reference to the design under test ForSpec returns an error if the first `text_in_quote` does not start with "/" or ".".

Examples:

```
1.x := "/dut/x".4+5."h";
```

```
isequivalentto x := /dut/x9h.
```

```
2.sig(thread,port,stage):="/dut/sigT".t."P".p." M".st."H";  
  y:=sig(1,20,509);  
  
  isequivalenttoy:=/dut/sigT1P20M509H;
```

9.1 *DefaultPath*

A default path can be set in the following syntax:

```
path default_path;
```

From this point on, the default path can serve as a prefix for any path using “.” (dot). For example:

```
path /a/b/c/d;  
b := ./e;
```

b is defined as /a/b/c/d/e.

Every path definition overrides the default path. The current default path cannot be used to redefine the default path. For example:

```
path /a/b/c;  
x := ./x;           // ⇔ x := /a/b/c/x  
path /d/e;  
y := ./y;           // ⇔ y := /d/e/y;  
path ./f;           // Error - can't use . in path definition
```

10 Including Files

ForSpec files can be included using:

```
include“filename.fs” ;
```

Where filename may also include a full path. For example:

```
include“Forspec/src/spec.fs”;
```

A file may contain several included directives. An included file is inserted exactly where the included directive is located. Including a file in different locations, may have different results.

For example, suppose the file `f_def.fs` contains the following definition:

```
f:=x/f;
```

Then the following ForSpec program:

```
include“f_def.fs”;
```

```
assertf;
```

will compile. But the following program:

```
g:=f;
```

```
include“f_def.fs”;
```

```
assertg;
```

will not compile, since `f` is undefined when trying to define `g`.

11 Precedence and Associativity

| Pr . | Operator s | Meaning | Associativity | Example Input | Example Meaning |
|---------|-------------------|---------------------------------|---------------|------------------|--------------------|
| 15 | t(params) | Template instantiation | | | |
| 14 | !~[]` | Not Extractand Prime | | | |
| 13 | % | Concatenation(bits) | | | |
| 12 | * | Multiple | | | |
| 11 | + - | Plus and Minus | Left to right | a - b - c | (a - b) - c |
| 10 | ^ | Bit-wise XOR | | | |
| 9 | <<>> | Shift Left and Right | Left to right | a << n << m | (a << n) << m |
| 8 | = != <> < => = | Comparisons | Left to right | a = b = 1 | (a = b) = 1 |
| 7 | +*?{ } | Repetition | Left to right | | |
| 6 | & | And (Both bit-wise and logical) | | | |
| 5 | | Bit (Both bit-wise and logical) | | | |
| 4 | , | Concatenation (events) | | | |
| 3 | \ | Glue (events) | | | |
| 2 | ->< -> | Implies and IFF | Left to right | | |
| 1 | always, | | Right to left | e seq e2 seq f | e seq (e2 seq f) |

| | | | | | |
|--|--|--|--|-------------------|---------------------|
| | eventually ,seq, until, triggers, next,... | | | alwaysauntil b | always(auntil b) |
|--|--|--|--|-------------------|---------------------|