A

Syntax Rule Summary

Below we present the syntax of PSL in Backus-Naur Form (BNF).

A.1 Conventions

The formal syntax described uses the following extended Backus-Naur Form (BNF).

a. The initial character of each word in a nonterminal is capitalized. For example:

   PSL_Statement

   A nonterminal is either a single word or multiple words separated by underscores. When a multiple word nonterminal containing underscores is referenced within the text (e.g., in a statement that describes the semantics of the corresponding syntax), the underscores are replaced with spaces.

b. Boldface words are used to denote reserved keywords, operators, and punctuation marks as a required part of the syntax. For example:

   vunit (;

c. The ::= operator separates the two parts of a BNF syntax definition. The syntax category appears to the left of this operator and the syntax description appears to the right of the operator. For example, item (d) shows three options for a *Vunit_Type*.

d. A vertical bar separates alternative items (use one only) unless it appears in boldface, in which case it stands for itself. For example:

Vunit_Type ::= **vunit** | **vprop** | **vmode**

e. Square brackets enclose optional items unless it appears in boldface, in which case it stands for itself. For example:

Sequence_Declaration ::=
sequence Name [**(** Formal_Parameter_List **)**] DEF_SYM Sequence **;**

indicates that (*Formal_Parameter_List*) is an optional syntax item for *Sequence_Declaration*, whereas

| Sequence [***** [Range]]

indicates that (the outer) square brackets are part of the syntax, while *Range* is optional.

f. Braces enclose a repeated item unless it appears in boldface, in which case it stands for itself. A repeated item may appear zero or more times; the repetitions occur from left to right as with an equivalent left-recursive rule. Thus, the following two rules are equivalent:

Formal_Parameter_List ::= Formal_Parameter { **;** Formal_Parameter }
Formal_Parameter_List ::=
Formal_Parameter | Formal_Parameter_List **;** Formal_Parameter

g. A colon (:) in a production starts a line comment unless it appears in boldface, in which case it stands for itself.

h. If the name of any category starts with an italicized part, it is equivalent to the category name without the italicized part. The italicized part is intended to convey some semantic information. For example, *vunit*_Name is equivalent to Name.

i. Flavor macros, containing embedded underscores, are shown in uppercase. These reflect the various HDLs that can be used within the PSL syntax and show the definition for each HDL. The general format is the term Flavor Macro, then the actual *macro name*, followed by the = operator, and, finally, the definition for each of the HDLs. For example:

Flavor Macro RANGE_SYM =
SystemVerilog: **:** / Verilog: **:** / VHDL: **to** / GDL: **/ ..**

shows the *range symbol* macro (RANGE_SYM). See for further details about *flavor macros*.

The main text uses *italicized* type when a term is being defined, and monospace font for examples and references to constants such as 0, 1, or x values.

A.2 Tokens

PSL syntax is defined in terms of primitive *tokens*, which are character sequences that act as distinct symbols in the language.

Each PSL keyword is a single token. Some keywords end in one or two non-alphabetic characters ('!' or '_' or both). Those characters are part of the keyword, not separate tokens.

Each of the following character sequences is also a token:

[	]	(	)	{	}
,	;	:	..	=	:=
*	+	|->	|=>	<->	->
[*	[+]	[->	[=		
&&	&	||	|	!	
$	@	.	/		

Finally, for a given flavor, the tokens of the corresponding HDL are tokens of PSL.

A.3 HDL dependencies

PSL depends upon the syntax and semantics of an underlying hardware description language. In particular, PSL syntax includes productions that refer to nonterminals in SystemVerilog, Verilog, VHDL, or GDL. PSL syntax also includes Flavor Macros that cause each flavor of PSL to match that of the underlying HDL for that flavor.

For SystemVerilog, the PSL syntax refers to the following nonterminals in the IEEE P1800 syntax:

– module_or_generate_item_declaration
– module_or_generate_item
– list_of_variable_identifiers
– identifier
– expression
– constant_expression

For Verilog, the PSL syntax refers to the following nonterminals in the IEEE Std 1364 syntax:

– module_or_generate_item_declaration
– module_or_generate_item
– list_of_variable_identifiers
– identifier
– expression
– constant_expression
– task_port_type

For VHDL, the PSL syntax refers to the following nonterminals in the IEEE Std 1076 syntax:

– block_declarative_item
– concurrent_statement
– design_unit
– identifier
– expression
– entity_aspect

For SystemC, the PSL syntax refers to the following nonterminals in the IEEE P1666 syntax:

– simple_type_specifier
– expression
– event_expression
– declaration
– statement
– identifier

For GDL, the PSL syntax refers to the following nonterminals in the GDL syntax:

– module_item_declaration
– module_item
– module_declaration
– identifier
– expression

A.3.1 Verilog extensions

For the Verilog flavor, PSL extends the forms of declaration that can be used in the modeling layer by defining two additional forms of type declaration.

Extended_Verilog_Declaration ::=
*Verilog*_module_or_generate_item_declaration
| Extended_Verilog_Type_Declaration

Extended_Verilog_Type_Declaration ::=
Finite_Integer_Type_Declaration
| Structure_Type_Declaration

Finite_Integer_Type_Declaration ::=
integer Integer_Range list_of_variable_identifiers **;**

Structure_Type_Declaration ::=
struct { Declaration_List **}** list_of_variable_identifiers **;**

Integer_Range ::=
(constant_expression **:** constant_expression **)**

Declaration_List ::=
HDL_Variable_or_Net_Declaration { HDL_Variable_or_Net_Declaration }

HDL_Variable_or_Net_Declaration ::=
net_declaration
| reg_declaration
| integer_declaration

A.3.2 Flavor macros

Flavor Macro DEF_SYM =
SystemVerilog: **=** / Verilog: **=** / VHDL: **is** / SystemC: **=** / GDL: **:=**

Flavor Macro RANGE_SYM =
SystemVerilog: **:** / Verilog: **:** / VHDL: **to** / SystemC: **:** / GDL: **..**

Flavor Macro AND_OP =
SystemVerilog: **&&** / Verilog: **&&** / VHDL: **and** / SystemC: **&&** / GDL: **&**

Flavor Macro OR_OP =
SystemVerilog: || / Verilog: || / VHDL: **or** / SystemC: || / GDL: |

Flavor Macro NOT_OP =
SystemVerilog: **!** / Verilog: **!** / VHDL: **not** / SystemC: **!** / GDL: **!**

Flavor Macro MIN_VAL =
SystemVerilog: **0** / Verilog: **0** / VHDL: **0** / SystemC: **0** / GDL: *null*

Flavor Macro MAX_VAL =
SystemVerilog: **$** / Verilog: **inf** / VHDL: **inf** / SystemC: **inf** / GDL: *null*

Flavor Macro HDL_EXPR =
SystemVerilog: *SystemVerilog*_Expression
/ Verilog: *Verilog*_Expression
/ VHDL: *VHDL*_Expression
/ SystemC: *SystemC*_Expression
/ GDL: *GDL*_Expression

Flavor Macro HDL_CLOCK_EXPR =
SystemVerilog: *SystemVerilog*_Event_Expression
/ Verilog: *Verilog*_Event_Expression
/ VHDL: *VHDL*_Expression
/ SystemC: *SystemC*_Event_Expression
/ GDL: *GDL*_Expression

Flavor Macro HDL_UNIT =
SystemVerilog: *SystemVerilog*_module_declaration
/ Verilog: *Verilog*_module_declaration
/ VHDL: *VHDL*_design_unit
/ SystemC: *SystemC_class*_sc_module
/ GDL: *GDL*_module_declaration

Flavor Macro HDL_DECL =
SystemVerilog: *SystemVerilog*_module_or_generate_item_declaration
/ Verilog: Extended_Verilog_Declaration
/ VHDL: *VHDL*_block_declarative_item
/ SystemC: *SystemC*_declaration
/ GDL: *GDL*_module_item_declaration

Flavor Macro HDL_STMT =
SystemVerilog: *SystemVerilog*_module_or_generate_item
/ Verilog: *Verilog*_module_or_generate_item
/ VHDL: *VHDL*_concurrent_statement
/ SystemC: *SystemC*_statement
/ GDL: *GDL*_module_item

Flavor Macro HDL_VARIABLE_TYPE =
SystemVerilog : *SystemVerilog*_data_type
/ Verilog : *Verilog*_Variable_Type
/ VHDL : *VHDL*_subtype_indication
/ SystemC: *SystemC*_simple_type_specifier
/ GDL : *GDL*_variable_type

Flavor Macro HDL_RANGE =
VHDL: range_attribute_name

Flavor Macro LEFT_SYM =
SystemVerilog: [/ Verilog: [/ VHDL: (/ SystemC: (/ GDL: (

Flavor Macro RIGHT_SYM =
SystemVerilog:] / Verilog:] / VHDL:) / SystemC:) / GDL:)

A.4 Syntax productions

The rest of this appendix defines the PSL syntax.

A.4.1 Verification units

PSL_Specification ::=
{ Verification_Item }

Verification_Item ::=
HDL_UNIT | Verification_Unit

Verification_Unit ::=
Vunit_Type *PSL*_Identifier [(Hierarchical_HDL_Name)] **{**
{ Inherit_Spec }
{ Vunit_Item }
}

Vunit_Type ::=
vunit | **vprop** | **vmode**

Hierarchical_HDL_Name ::=
HDL_Module_NAME { Path_Separator *instance*_Name }

*instance*_Name ::=
HDL_or_PSL_Identifier

HDL_Module_Name ::=
*HDL_Module*_Name [(*HDL_Module_Name*)]

Path_Separator ::=
. | /

Inherit_Spec ::=
inherit *vunit*_Name { **,** *vunit*_Name } **;**

Vunit_Item ::=
HDL_DECL
| HDL_STMT
| PSL_Declaration
| PSL_Directive

A.4.2 PSL declarations

PSL_Declaration ::=
Property_Declaration
| Sequence_Declaration
| Clock_Declaration

Property_Declaration ::=
property *PSL*_Identifier [**(** Formal_Parameter_List **)**] DEF_SYM Property **;**

Formal_Parameter_List ::=
Formal_Parameter { ; Formal_Parameter }

Formal_Parameter ::=
Param_Spec *PSL*_Identifier { , *PSL*_Identifier }

Param_Spec ::=
const
| [**const**] Value_Parameter
| **sequence**
| **property**

Value_Parameter ::=
HDL_Type
| PSL_Type_Class

HDL_Type ::=

hdltype HDL_VARIABLE_TYPE

PSL_Type_Class ::= **boolean** | **bit** | **bitvector** | **numeric** | **string**

Sequence_Declaration ::=
sequence *PSL*_Identifier [**(** Formal_Parameter_List **)**] DEF_SYM Sequence **;**

Clock_Declaration ::=
default clock DEF_SYM Clock_Expression **;**

Clock_Expression ::=
*boolean*_Name
| *boolean*_Built_In_Function_Call
| **(** Boolean **)**
| **(** HDL_CLOCK_EXPR **)**

Actual_Parameter_List ::=
Actual_Parameter { **,** Actual_Parameter }

Actual_Parameter ::=
AnyType|Number | Boolean | Property | Sequence

A.4.3 PSL directives

PSL_Directive ::=
[Label **:**] Verification_Directive

Label ::=
*PSL*_Identifier

HDL_or_PSL_Identifier ::=
*SystemVerilog*_Identifier
| *Verilog*_Identifier
| *VHDL*_Identifier
| *SystemC*_Identifier
| *GDL*_Identifier
| *PSL*_Identifier

Verification_Directive ::=
Assert_Directive
| Assume_Directive
| Assume_Guarantee_Directive
| Restrict_Directive

| Restrict_Guarantee_Directive
| Cover_Directive
| Fairness_Statement

Assert_Directive ::=
assert Property [**report** String] **;**

Assume_Directive ::=
assume Property **;**

Assume_Guarantee_Directive ::=
assume_guarantee Property [**report** String] **;**

Restrict_Directive ::=
restrict Sequence **;**

Restrict_Guarantee_Directive ::=
restrict_guarantee Sequence [**report** String] **;**

Cover_Directive ::=
cover Sequence [**report** String] **;**

Fairness_Statement ::=
fairness Boolean **;**
| **strong fairness** Boolean **,** Boolean **;**

A.4.4 PSL properties

Property ::=
Replicator Property
| FL_Property
| OBE_Property

Replicator ::=
forall Parameter_Definition **:**

Index_Range ::=
LEFT_SYM *finite*_Range RIGHT_SYM
| **(** HDL_RANGE **)**

Value_Set ::=
{ Value_Range { **,** Value_Range } **}**
| **boolean**

Value_Range ::=
    Value
  | *finite*_Range

Value ::=
    Boolean
  | Number

FL_Property ::=
    Boolean
  | (FL_Property)
  | Sequence [**!**]
  | *FL_property*_Name [(Actual_Parameter_List)]
  | FL_Property **@** Clock_Expression
  | FL_Property **abort** Boolean
  | FL_Property **async_abort** Boolean
  | FL_Property **sync_abort** Boolean
  | Parameterized_Property
: Logical Operators :
  | NOT_OP FL_Property
  | FL_Property AND_OP FL_Property
  | FL_Property OR_OP FL_Property
  :
  | FL_Property **->** FL_Property
  | FL_Property **<–>** FL_Property
: Primitive LTL Operators :
  | **X** FL_Property
  | **X!** FL_Property
  | **F** FL_Property
  | **G** FL_Property
  | [FL_Property **U** FL_Property]
  | [FL_Property **W** FL_Property]
: Simple Temporal Operators :
  | **always** FL_Property
  | **never** FL_Property
  | **next** FL_Property
  | **next!** FL_Property
  | **eventually!** FL_Property
  :
  | FL_Property **until!** FL_Property
  | FL_Property **until** FL_Property
  | FL_Property **until!_** FL_Property
  | FL_Property **until_** FL_Property
  :

| FL_Property **before!** FL_Property
| FL_Property **before** FL_Property
| FL_Property **before!_** FL_Property
| FL_Property **before_** FL_Property
: Extended Next (Event) Operators :
| **X** [Number] (FL_Property)
| **X!** [Number] (FL_Property)
| **next** [Number] (FL_Property)
| **next!** [Number] (FL_Property)
:
| **next_a** [*finite*_Range] (FL_Property)
| **next_a!** [*finite*_Range] (FL_Property)
| **next_e** [*finite*_Range] (FL_Property)
| **next_e!** [*finite*_Range] (FL_Property)
:
| **next_event!** (Boolean) (FL_Property)
| **next_event** (Boolean) (FL_Property)
| **next_event!** (Boolean) [*positive*_Number] (FL_Property)
| **next_event** (Boolean) [*positive*_Number] (FL_Property)
:
| **next_event_a!** (Boolean) [*finite_positive*_Range] (FL_Property)
| **next_event_a** (Boolean) [*finite_positive*_Range] (FL_Property)
| **next_event_e!** (Boolean) [*finite_positive*_Range] (FL_Property)
| **next_event_e** (Boolean) [*finite_positive*_Range] (FL_Property)
: Operators on SEREs :
| { SERE } (FL_Property)
| Sequence |−> FL_Property
| Sequence |=> FL_Property

A.4.5 Sequential Extended Regular Expressions (SEREs)

SERE ::=
Boolean
| Sequence
| SERE **;** SERE
| SERE **:** SERE
| Compound_SERE

Compound_SERE ::=
Repeated_SERE
| Braced_SERE
| Clocked_SERE
| Compound_SERE | Compound_SERE

| Compound_SERE **&** Compound_SERE
| Compound_SERE **&&** Compound_SERE
| Compound_SERE **within** Compound_SERE
| Parameterized_SERE

A.4.6 Parameterized Properties and SEREs

Parameterized_Property ::=
for Parameters_Definition **:** And_Or_Property_OP **(** FL_Property **)**

Parameterized_SERE ::=
for Parameters_Definition **:** And_Or_SERE_OP **{** SERE **}**

Parameters_Definition ::=
Parameter_Definition { Parameter_Definition }

Parameter_Definition ::=
PSL_Identifier [Index_Range] **in** Value_Set

And_OR_Property_OP ::=
AND_OP
| OR_OP

And_Or_SERE_Op :: =
&& | **&** | |

A.4.7 Sequences

Sequence ::=
Sequence_Instance
| Repeated_SERE
| Braced_SERE
| Clocked_SERE

Repeated_SERE ::=
Boolean [***** [Count]]
| Sequence [***** [Count]]
| [***** [Count]]
| Boolean [+]
| Sequence [+]
| [+]

| Boolean [= Count]
| Boolean [-> [*positive*_Count]]

Braced_SERE ::=
{ SERE **}**

Sequence_Instance ::=
*sequence*_Name [**(** Actual_Parameter_List **)**]

Clocked_SERE ::=
Braced_SERE @ Clock_Expression

Count ::=
Number
| Range

Range ::=
Low_Bound RANGE_SYM High_Bound

Low_Bound ::=
Number
| MIN_VAL

High_Bound ::=
Number
| MAX_VAL

A.4.8 Forms of expression

Any_Type ::=
HDL_or_PSL_Expression

Bit ::=
*bit*_HDL_or_PSL_Expression

Boolean ::=
*boolean*_HDL_or_PSL_Expression

BitVector ::=
*bitvector*_HDL_or_PSL_Expression

Number ::=
*numeric*_HDL_or_PSL_Expression

String ::=
*string*_HDL_or_PSL_Expression

HDL_or_PSL_Expression ::=
HDL_Expression
| PSL_Expression
| Built_In_Function_Call
| Union_Expression

HDL_Expression ::=
HDL_EXPR

PSL_Expression ::=
Boolean -> Boolean
| Boolean <–> Boolean

Built_In_Function_Call ::=
prev (Any_Type [, Number [, Clock_Expression]])
| **next** (Any_Type)
| **stable** (Any_Type [, Clock_Expression])
| **rose** (Bit [, Clock_Expression])
| **fell** (Bit [, Clock_Expression])
| **ended** (Sequence [, Clock_Expression])
| **isunknown** (BitVector)
| **countones** (BitVector)
| **onehot** (BitVector)
| **onehot0** (BitVector)
| **nondet** (Value_List)
| **nondet_vector** (Number, Value_List)

Union_Expression ::=
Any_Type **union** Any_Type

A.4.9 Optional Branching Extension

OBE_Property ::=
Boolean
| (OBE_Property)
| *OBE_property*_Name [(Actual_Parameter_List)]
: Logical Operators :
| NOT_OP OBE_Property
| OBE_Property AND_OP OBE_Property

| OBE_Property OR_OP OBE_Property
| OBE_Property -> OBE_Property
| OBE_Property <−> OBE_Property
: Universal Operators :
| **AX** OBE_Property
| **AG** OBE_Property
| **AF** OBE_Property
| **A** [OBE_Property **U** OBE_Property]
:Existential Operators :
| **EX** OBE_Property
| **EG** OBE_Property
| **EF** OBE_Property
| **E** [OBE_Property **U** OBE_Property]

B

Formal Syntax and Semantics

This appendix formally describes the syntax and semantics of the temporal layer.

B.1 Typed-text representation of symbols

Table B.1 shows the mapping of various symbols used in this definition to the corresponding typed-text PSL representation, in the different flavors.

Table B.1: Typed-text symbols in the SystemVerilog, Verilog, VHDL, SystemC and GDL flavors

	SystemVerilog	Verilog	VHDL	SystemC	GDL
$\mapsto$	`|->`	`|->`	`|->`	`|->`	`|->`
$\Mapsto$	`|=>`	`|=>`	`|=>`	`|=>`	`|=>`
$\rightarrow$	`->`	`->`	`->`	`->`	`->`
$\leftrightarrow$	`<->`	`<->`	`<->`	`<->`	`<->`
$\neg$	`!`	`!`	`not`	`!`	`!`
$\wedge$	`&&`	`&&`	`and`	`&&`	`&`
$\vee$	`||`	`||`	`or`	`||`	`|`
..	`:`	`:`	`to`	`:`	`..`
$\langle\ \rangle$	`[ ]`	`[ ]`	`( )`	`( )`	`( )`

NOTE –
For reasons of simplicity, the syntax given herein is more flexible than the one defined by the extended BNF (given in Appendix A). That is, some of the expressions which are legal here are not legal under the BNF grammar. Users should use the stricter syntax, as defined by the BNF grammar in Appendix A.

B.2 Syntax

The logic PSL is defined with respect to a non-empty set of atomic propositions P and a given set of boolean expressions B over P. We assume two designated boolean expression *true* and *false* belong to B.

Definition 1 (Sequential Extended Regular Expressions (SEREs))

1. *Every boolean expression $b \in B$ is a SERE.*
2. *If r, r_1, and r_2 are SEREs, and c is a boolean expression, then the following are SEREs:*
   - $\{r\}$
   - $r_1 \; ; \; r_2$
   - $r_1 : r_2$
   - $r_1 \mid r_2$
   - $r_1 \;\&\&\; r_2$
   - $[*0]$
   - $r[*]$
   - r **@** c

Definition 2 (FL formulas)

1. *If b is a boolean expression, then both b and $b!$ are FL formulas.*
2. *If φ and ψ are FL formulas, r, r_1, r_2 are SEREs, and b a boolean expression, then the following are FL formulas:*
   - (φ)
   - $\neg\varphi$
   - $\varphi \wedge \psi$
   - $r \mapsto \varphi$
   - $r!$
   - r
   - $X!\ \varphi$
   - $[\varphi \; U \; \psi]$
   - φ **@** b
   - φ *async_abort* b
   - φ *sync_abort* b

NOTE –
We define formal semantics for both strong and weak booleans[20]. However, strong booleans are not accessible to the user.

Definition 3 (OBE Formulas)

1. *Every boolean expression is an OBE formula.*
2. *If f, f_1, and f_2 are OBE formulas, then so are the following:*
   a) (f)
   b) $\neg f$
   c) $f_1 \wedge f_2$
   d) EXf
   e) $E[f_1 \; U \; f_2]$
   f) EGf

   Additional OBE operators are derived from these as follows:

- $f_1 \vee f_2 = \neg(\neg f_1 \wedge \neg f_2)$
- $f_1 \rightarrow f_2 = \neg f_1 \vee f_2$
- $f_1 \leftrightarrow f_2 = (f_1 \rightarrow f_2) \wedge (f_2 \rightarrow f_1)$
- $EFf = E[true \; U \; f]$
- $AXf = \neg EX\neg f$
- $A[f_1 \; U \; f_2] = \neg(E[\neg f_2 \; U \; (\neg f_1 \wedge \neg f_2)] \vee EG\neg f_2)$
- $AGf = \neg E[true \; U \; \neg f]$

- $AFf = A[true\ U\ f]$

Definition 4 (PSL Formulas)

1. *Every FL formula is a PSL formula.*
2. *Every OBE formula is a PSL formula.*

In Section B.4, we show additional operators which provide syntactic sugaring to the ones above.

B.3 Semantics

The semantics of PSL formulas are defined with respect to a *model*. A model is a quintuple (S, S_0, R, P, L), where S is a finite set of states, $S_0 \subseteq S$ is a set of initial states, $R \subseteq S \times S$ is the transition relation, P is a non-empty set of atomic propositions, and L is the valuation, a function $L : S \longrightarrow 2^P$, mapping each state with a set of atomic propositions valid in that state.

A *path* π is a finite (or infinite) sequence of states $\pi = (\pi_0, \pi_1, \pi_2, \cdots, \pi_n)$ (or $\pi = (\pi_0, \pi_1, \pi_2, \cdots)$). A *computation path* π of a model M is a finite (or infinite) path π such that for every $i < n$, $R(\pi_i, \pi_{i+1})$ and for no s, $R(\pi_n, s)$ (or such that for every i, $R(\pi_i, \pi_{i+1})$). Given a finite (or infinite) path π, we define $\hat{L}$, an extension of the valuation function L from states to paths as follows: $\hat{L}(\pi) = L(\pi_0)L(\pi_1)\ldots L(\pi_n)$ (or $\hat{L}(\pi) = L(\pi_0)L(\pi_1)\ldots$). Thus we have a mapping from states in M to letters of 2^P, and from finite (or infinite) sequences of states in M to finite (or infinite) words over 2^P.

B.3.1 Semantics of FL formulas

The semantics of FL formulas is interpreted over finite and infinite words over $\Sigma = 2^P \cup \{\top, \bot\}$. Let φ be an FL formula, w a word over Σ and M a model. The notation $w \models \varphi$ means that the FL formula φ holds over the word w. The notation $M \models \varphi$ means that for all π such that π is computation path of M, $\hat{L}(\pi) \models \varphi$.

We denote a letter from Σ by ℓ and an empty, finite, or infinite word from Σ by u, v, or w (possibly with subscripts). We denote the length of word v as $|v|$. A finite non-empty word $v = (\ell_0\ell_1\ell_2\cdots\ell_n)$ has length $n+1$, the (finite) empty word $v = \epsilon$ has length 0, and an infinite word has length ∞. We use i, j, and k to denote non-negative integers. We denote the i^{th} letter of v by v^{i-1} (since counting of letters starts at zero). We denote by $v^{i\cdot\cdot}$ the suffix of v starting at v^i. That is, for every $i < |v|$, $v^{i\cdot\cdot} = v^i v^{i+1}\cdots v^n$ or $v^{i\cdot\cdot} = v^i v^{i+1}\cdots$. We denote by $v^{i..j}$ the finite sequence of letters starting from v^i and ending in v^j. That is, for $j \geq i$, $v^{i..j} = v^i v^{i+1}\cdots v^j$ and for $j < i$, $v^{i..j} = \epsilon$. We use ℓ^ω to denote an infinite-length word, each letter of which is ℓ.

We use $\overline{v}$ to denote the word obtained by replacing every $\top$ with a $\bot$ and vice versa. We call $\overline{v}$ the *dual* of v.

The semantics of FL *formulas* over *words* is defined inductively, using as the base case the semantics of *boolean expressions* over *letters* in Σ. The semantics of boolean expression is assumed to be given as a relation $\models \subseteq \Sigma \times B$ relating letters in Σ with boolean expressions in B. If $(\ell, b) \in \models$ we say that the letter ℓ *satisfies* the boolean expression b and denote it $\ell \models b$. We assume the two special letters $\top$ and $\bot$ behave as follows: for every boolean expression b, $\top \models b$ and $\bot \not\models b$. We assume that otherwise the boolean relation $\models$ behaves in the usual manner. In particular, that for every letter $\ell \in 2^P$, atomic proposition $p \in P$ and boolean expressions $b, b_1, b_2 \in B$ (i) $\ell \models p$ iff $p \in \ell$, (ii) $\ell \models \neg b$ iff $\ell \not\models b$, and (iii) $\ell \models \mathit{true}$ and $\ell \not\models \mathit{false}$. Finally, we assume that for every letter $\ell \in \Sigma$, $\ell \models b_1 \wedge b_2$ iff $\ell \models b_1$ and $\ell \models b_2$.

Unclocked Semantics

Semantics of unclocked SEREs

Unclocked SEREs are defined over finite words over the alphabet Σ. The notation $v \models r$, where r is a SERE and v a finite word means that v *models tightly* r. The semantics of unclocked SEREs are defined as follows, where b denotes a boolean expression, and r, r_1, and r_2 denote unclocked SEREs:

1. $v \models \{r\} \iff v \models r$
2. $v \models b \iff |v| = 1$ and $v^0 \models b$
3. $v \models r_1 \ ; \ r_2 \iff \exists v_1, v_2$ s.t. $v = v_1 v_2$, $v_1 \models r_1$, and $v_2 \models r_2$
4. $v \models r_1 : r_2 \iff \exists v_1, v_2$, and ℓ s.t. $v = v_1 \ell v_2$, $v_1 \ell \models r_1$, and $\ell v_2 \models r_2$
5. $v \models r_1 \mid r_2 \iff v \models r_1$ or $v \models r_2$
6. $v \models r_1 \ \&\& \ r_2 \iff v \models r_1$ and $v \models r_2$
7. $v \models [*0] \iff v = \epsilon$
8. $v \models r[*] \iff$ either $v \models [*0]$
   or $\exists v_1, v_2$ s.t. $v_1 \neq \epsilon$, $v = v_1 v_2$, $v_1 \models r$ and $v_2 \models r[*]$

Semantics of unclocked FL

We refer to a formula of FL with no `@` operator as an *unclocked formula*. Let v be a finite or infinite word, b be a boolean expression, r, r_1, r_2 unclocked SEREs, and φ, ψ unclocked FL formulas. We use $\models$ to define the semantics of unclocked FL formulas: If $v \models \varphi$ we say that v *models* (or *satisfies*) φ.

1. $v \models (\varphi) \iff v \models \varphi$
2. $v \models \neg\varphi \iff \overline{v} \not\models \varphi$
3. $v \models \varphi \wedge \psi \iff v \models \varphi$ and $v \models \psi$
4. $v \models b! \iff |v| > 0$ and $v^0 \models b$

5. $v \models b \iff |v| = 0$ or $v^0 \Vdash b$
6. $v \models r \mapsto \varphi \iff \forall j < |v|$ s.t. $\overline{v}^{0..j} \models r$, $v^{j..} \models \varphi$
7. $v \models r! \iff \exists j < |v|$ s.t. $v^{0..j} \models r$
8. $v \models r \iff \forall j < |v|$, $v^{0..j}\top^\omega \models r!$
9. $v \models X!\ \varphi \iff |v| > 1$ and $v^{1..} \models \varphi$
10. $v \models [\varphi U \psi] \iff \exists k < |v|$ s.t. $v^{k..} \models \psi$, and $\forall j < k,\ \ v^{j..} \models \varphi$
11. $v \models \varphi$ async_abort $b \iff$ either $v \models \varphi$
    or $\exists j < |v|$ s.t. $v^j \Vdash b$ and $v^{0..j-1}\top^\omega \models \varphi$
12. $v \models \varphi$ sync_abort $b \iff$ either $v \models \varphi$
    or $\exists j < |v|$ s.t. $v^j \Vdash b$ and $v^{0..j-1}\top^\omega \models \varphi$

NOTES –

1. The semantics given here for the LTL operators and the async_abort operator is equivalent to the truncated semantics given in [18] which is interpreted over 2^P rather than over $2^P \cup \{\top, \bot\}$. Using $\models_\bullet$ for the semantics in [18], the following proposition states the equivalence: Let w be a finite word over 2^P, and let φ be a formula of LTL^{trunc}. Then, as shown in [19], the three following equivalences hold:
$$w \models^{-}_\bullet \varphi \iff w\top^\omega \models \varphi$$
$$w \models_\bullet \varphi \iff w \models \varphi$$
$$w \models^{+}_\bullet \varphi \iff w\bot^\omega \models \varphi$$
2. Using $\models_\bullet$ as in the note 1 above, we use *holds strongly* for $\models^{+}_\bullet$, *holds* for $\models_\bullet$, and *holds weakly* for $\models^{-}_\bullet$. The remaining terminology of Section 11.1 is formally defined as follows:
   - φ is *pending* on word w iff $w \models^{-}_\bullet \varphi$ and $w \not\models_\bullet \varphi$
   - φ *fails* on word w iff $w \not\models^{-}_\bullet \varphi$
3. There is a subtle difference between boolean negation and formula negation. For instance, consider the formula $\neg b$. If $\neg$ is boolean negation, then $\neg b$ holds on an empty path. If $\neg$ is formula negation, then $\neg b$ does not hold on an empty path. Rather than introduce distinct operators for boolean and formula negation, we instead adopt the convention that negation applied to a boolean expression is boolean negation. This does not restrict expressivity, as formula negation of b can be expressed as $(\neg b)!$.

Clocked Semantics

We say that finite word v *is a clock tick of* c iff $|v| > 0$ and $v^{|v|-1} \Vdash c$ and for every natural number $i < |v| - 1$, $v^i \Vdash \neg c$.

Semantics of clocked SEREs

Clocked SEREs are defined over finite words from the alphabet Σ and a boolean expression that serves as the clock context. The notation $v \models\!\!\!\models^{c} r$, where r is a SERE, c is a boolean expression and v a finite word, means that v *models tightly* r *in context of clock* c. The semantics of clocked SEREs are defined as follows, where b, c, and c_1 denote boolean expressions, and r, r_1, and r_2 denote clocked SEREs:

1. $v \models\!\!\!\models^{c} \{r\} \iff v \models\!\!\!\models^{c} r$
2. $v \models\!\!\!\models^{c} b \iff v$ is a clock tick of c and $v^{|v|-1} \models b$
3. $v \models\!\!\!\models^{c} r_1 \ ; \ r_2 \iff \exists v_1, v_2$ s.t. $v = v_1 v_2$, $v_1 \models\!\!\!\models^{c} r_1$, and $v_2 \models\!\!\!\models^{c} r_2$
4. $v \models\!\!\!\models^{c} r_1 : r_2 \iff \exists v_1, v_2$, and ℓ s.t. $v = v_1 \ell v_2$, $v_1 \ell \models\!\!\!\models^{c} r_1$, and $\ell v_2 \models\!\!\!\models^{c} r_2$
5. $v \models\!\!\!\models^{c} r_1 \mid r_2 \iff v \models\!\!\!\models^{c} r_1$ or $v \models\!\!\!\models^{c} r_2$
6. $v \models\!\!\!\models^{c} r_1 \ \&\& \ r_2 \iff v \models\!\!\!\models^{c} r_1$ and $v \models\!\!\!\models^{c} r_2$
7. $v \models\!\!\!\models^{c} [*0] \iff v = \epsilon$
8. $v \models\!\!\!\models^{c} r[*] \iff$ either $v \models\!\!\!\models^{c} [*0]$
   or $\exists v_1, v_2$ s.t. $v_1 \neq \epsilon$, $v = v_1 v_2$, $v_1 \models\!\!\!\models^{c} r$ and $v_2 \models\!\!\!\models^{c} r[*]$
9. $v \models\!\!\!\models^{c} r@c_1 \iff v \models\!\!\!\models^{c_1} r$

Semantics of clocked FL

The semantics of (clocked) FL formulas is defined with respect to finite/infinite words over Σ and a boolean expression c which serves as the clock context. Let v be a finite or infinite word, b, c, c_1 boolean expressions, r, r_1, r_2 SEREs, and φ, ψ FL formulas. We use $\models^{c}$ to define the semantics of FL formulas. If $v \models^{c} \varphi$ we say that v *models* (or *satisfies*) φ *in the context of clock* c.

1. $v \models^{c} (\varphi) \iff v \models^{c} \varphi$
2. $v \models^{c} \neg\varphi \iff \overline{v} \not\models^{c} \varphi$
3. $v \models^{c} \varphi \wedge \psi \iff v \models^{c} \varphi$ and $v \models^{c} \psi$
4. $v \models^{c} b! \iff \exists j < |v|$ s.t. $v^{0..j}$ is a clock tick of c and $v^{j} \models b$
5. $v \models^{c} b \iff \forall j < |v|$ s.t. $\overline{v}^{0..j}$ is a clock tick of c, $v^{j} \models b$
6. $v \models^{c} r \mapsto \varphi \iff \forall j < |v|$ s.t. $\overline{v}^{0..j} \models\!\!\!\models^{c} r$, $v^{j..} \models^{c} \varphi$
7. $v \models^{c} r! \iff \exists j < |v|$ s.t. $v^{0..j} \models\!\!\!\models^{c} r$
8. $v \models^{c} r \iff \forall j < |v|$, $v^{0..j}\top^{\omega} \models^{c} r!$
9. $v \models^{c} X!\ f \iff \exists j < k < |v|$ s.t. $v^{0..j}$ and $v^{j+1..k}$ are clock ticks of c and $v^{k..} \models^{c} f$
10. $v \models^{c} [\varphi U \psi] \iff \exists k < |v|$ s.t. $v^{k} \models c$, $v^{k..} \models^{c} \psi$, and $\forall j < k$ s.t. $v^{j} \models c$, $v^{j..} \models^{c} \varphi$

11. $v \models^{c} \varphi @ c_1 \iff v \models^{c_1} \varphi$
12. $v \models^{c} \varphi$ async_abort $b \iff$ either $v \models^{c} \varphi$
    or $\exists j < |v|$ s.t. $v^j \Vdash b$ and $v^{0..j-1}\top^{\omega} \models^{c} \varphi$
13. $v \models^{c} \varphi$ sync_abort $b \iff$ either $v \models^{c} \varphi$ or
    or $\exists j < |v|$ s.t. $v^j \Vdash b \wedge c$ and $v^{0..j-1}\top^{\omega} \models^{c} \varphi$

NOTE –
The clocked semantics for the LTL subset follows the clocks paper [20], with the exception that strength is applied at the boolean level rather than at the propositional level.

B.3.2 Semantics of OBE formulas

The semantics of OBE formulas are defined over states in the *model*, rather than finite or infinite words. Let f be an OBE formula, $M = (S, S_0, R, P, L)$ a model and $s \in S$ a state of the model. The notation $M, s \models f$ means that f holds in state s of model M. The notation $M \models f$ is equivalent to $\forall s \in S_0 : M, s \models f$. In other words, f is valid for every initial state of M.

The semantics of OBE formulas are defined inductively, using as the base case the semantics of *boolean expressions* over *letters* in 2^P. The semantics of boolean expression is assumed to be given as a relation $\Vdash \subseteq 2^P \times B$ relating letters in 2^P with boolean expressions in B. If $(\ell, b) \in \Vdash$ we say that the letter ℓ *satisfies* the boolean expression b and denote it $\ell \Vdash b$. We assume that the boolean relation $\Vdash$ behaves in the usual manner. In particular, that for every letter $\ell \in 2^P$, atomic proposition $p \in P$ and boolean expressions $b, b_1, b_2 \in B$ (i) $\ell \Vdash p$ iff $p \in \ell$, (ii) $\ell \Vdash \neg b$ iff $\ell \nVdash b$, (iii) $\ell \Vdash b_1 \wedge b_2$ iff $\ell \Vdash b_1$ and $\ell \Vdash b_2$, and (iv) $\ell \Vdash$ *true* and $\ell \nVdash$ *false*.

The semantics of an OBE formula are those of standard CTL. The semantics are defined as follows, where b denotes a boolean expression and f, f_1, and f_2 denote OBE formulas:

1. $M, s \models b \iff L(s) \Vdash b$
2. $M, s \models (f) \iff M, s \models f$
3. $M, s \models \neg f \iff M, s \not\models f$
4. $M, s \models f_1 \wedge f_2 \iff M, s \models f_1$ and $M, s \models f_2$
5. $M, s \models EX\ f \iff$ there exists a computation path π of M such that $|\pi| > 1$, $\pi_0 = s$, and $M, \pi_1 \models f$
6. $M, s \models E[f_1\ U\ f_2] \iff$ there exists a computation path π of M such that $\pi_0 = s$ and there exists $k < |\pi|$ such that $M, \pi_k \models f_2$ and for every j such that $j < k$: $M, \pi_j \models f_1$
7. $M, s \models EG\ f \iff$ there exists a computation path π of M such that $\pi_0 = s$ and for every j such that $0 \leq j < |\pi|$: $M, \pi_j \models f$

B.4 Syntactic Sugaring

The remainder of the temporal layer is syntactic sugar. In other words, it does not add expressive power, and every piece of syntactic sugar can be defined in terms of the basic FL operators presented above. The syntactic sugar is defined below.

NOTE –
The definitions given here do not necessarily represent the most efficient implementation. In some cases, there is an equivalent syntactic sugaring, or a direct implementation, that is more efficient.

B.4.1 Additional SERE operators

Let i, j, k, and l be integer constants such that $i \geq 0$, $j \geq i$, $k \geq 1$, $l \geq k$. Then, additional SERE operators can be viewed as abbreviations of the basic SERE operators defined above, as follows, where b denotes a boolean expression, and r denotes a SERE:

- $r[+] \stackrel{\text{def}}{=} r; r[*]$
- $r[*0] \stackrel{\text{def}}{=} [*0]$
- $r[*k] \stackrel{\text{def}}{=} \overbrace{r; r; ...; r}^{k\ times}$
- $r[*i..j] \stackrel{\text{def}}{=} r[*i] \mid ... \mid r[*j]$
- $r[*i..] \stackrel{\text{def}}{=} r[*i]; r[*]$
- $r[*..i] \stackrel{\text{def}}{=} r[*0] \mid ... \mid r[*i]$
- $r[*..] \stackrel{\text{def}}{=} r[*0..]$
- $[+] \stackrel{\text{def}}{=} true[+]$
- $[*] \stackrel{\text{def}}{=} true[*]$
- $[*i] \stackrel{\text{def}}{=} true[*i]$
- $[*i..j] \stackrel{\text{def}}{=} true[*i..j]$
- $[*i..] \stackrel{\text{def}}{=} true[*i..]$
- $[*..i] \stackrel{\text{def}}{=} true[*..i]$
- $[*..] \stackrel{\text{def}}{=} true[*..]$
- $b[= i] \stackrel{\text{def}}{=} \{\neg b[*]; b\}[*i]; \neg b[*]$
- $b[= i..j] \stackrel{\text{def}}{=} b[= i] \mid ... \mid b[= j]$
- $b[= i..] \stackrel{\text{def}}{=} b[= i]; [*]$
- $b[= ..i] \stackrel{\text{def}}{=} b[= 0] \mid ... \mid b[= i]$
- $b[= ..] \stackrel{\text{def}}{=} b[= 0..]$
- $b[\rightarrow] \stackrel{\text{def}}{=} \neg b[*]; b$
- $b[\rightarrow k] \stackrel{\text{def}}{=} \{\neg b[*]; b\}[*k]$

- $b[\rightarrow k..l] \stackrel{\text{def}}{=} b[\rightarrow k] \mid ... \mid b[\rightarrow l]$
- $b[\rightarrow k..] \stackrel{\text{def}}{=} b[\rightarrow k] \mid \{b[\rightarrow k]; [*]; b\}$
- $b[\rightarrow ..k] \stackrel{\text{def}}{=} b[\rightarrow 1] \mid ... \mid b[\rightarrow k]$
- $b[\rightarrow ..] \stackrel{\text{def}}{=} b[\rightarrow 1..]$
- $r_1 \ \& \ r_2 \stackrel{\text{def}}{=} \{\{r_1\} \ \&\& \ \{r_2; true[*]\}\} \mid \{\{r_1; true[*]\} \ \&\& \ \{r_2\}\}$
- $r_1 \ within \ r_2 \stackrel{\text{def}}{=} \{[*];\ r_1;\ [*]\} \ \&\& \ \{r_2\}$

B.4.2 Additional FL operators

Let i, j, k and l be integers such that $i \geq 0$, $j \geq i$, $k > 0$ and $l \geq k$. Then, additional operators can be viewed as abbreviations of the basic operators defined above, as follows, where b denotes a boolean expression, r, r_1, and r_2 denote SEREs, and φ, φ_1, and φ_2 denote FL formulas:

- $\varphi_1 \vee \varphi_2 \stackrel{\text{def}}{=} \neg(\neg\varphi_1 \wedge \neg\varphi_2)$
- $\varphi_1 \rightarrow \varphi_2 \stackrel{\text{def}}{=} \neg\varphi_1 \vee \varphi_2$
- $\varphi_1 \leftrightarrow \varphi_2 \stackrel{\text{def}}{=} (\varphi_1 \rightarrow \varphi_2) \wedge (\varphi_2 \rightarrow \varphi_1)$

- $F\varphi \stackrel{\text{def}}{=} [true \ U \ \varphi]$
- $G\varphi \stackrel{\text{def}}{=} \neg F\neg\varphi$
- $X\varphi \stackrel{\text{def}}{=} \neg X! \ \neg\varphi$
- $[\varphi_1 \ W \ \varphi_2] \stackrel{\text{def}}{=} [\varphi_1 \ U \ \varphi_2] \vee G\varphi_1$

- $always \ \varphi \stackrel{\text{def}}{=} G \ \varphi$
- $never \ \varphi \stackrel{\text{def}}{=} G \ \neg\varphi$
- $next! \ \varphi \stackrel{\text{def}}{=} X! \ \varphi$
- $next \ \varphi \stackrel{\text{def}}{=} X \ \varphi$
- $eventually! \ \varphi \stackrel{\text{def}}{=} F\varphi$

- $\varphi_1 \ until! \ \varphi_2 \stackrel{\text{def}}{=} [\varphi_1 \ U \ \varphi_2]$
- $\varphi_1 \ until \ \varphi_2 \stackrel{\text{def}}{=} [\varphi_1 \ W \ \varphi_2]$
- $\varphi_1 \ until!\_ \ \varphi_2 \stackrel{\text{def}}{=} [\varphi_1 \ U \ \varphi_1 \wedge \varphi_2]$
- $\varphi_1 \ until\_ \ \varphi_2 \stackrel{\text{def}}{=} [\varphi_1 \ W \ \varphi_1 \wedge \varphi_2]$

- $\varphi_1 \ before! \ \varphi_2 \stackrel{\text{def}}{=} [\neg\varphi_2 \ U \ \varphi_1 \wedge \neg\varphi_2]$
- $\varphi_1 \ before \ \varphi_2 \stackrel{\text{def}}{=} [\neg\varphi_2 \ W \ \varphi_1 \wedge \neg\varphi_2]$
- $\varphi_1 \ before!\_ \ \varphi_2 \stackrel{\text{def}}{=} [\neg\varphi_2 \ U \ \varphi_1]$
- $\varphi_1 \ before\_ \ \varphi_2 \stackrel{\text{def}}{=} [\neg\varphi_2 \ W \ \varphi_1]$

- $X!\ [i]\varphi \stackrel{\text{def}}{=} \overbrace{X!\ X!\ ...X!}^{i\ times}\ \varphi$
- $X[i]\varphi \stackrel{\text{def}}{=} \overbrace{XX...X}^{i\ times}\ \varphi$
- $next![i]\ \varphi \stackrel{\text{def}}{=} X!\ [i]\ \varphi$
- $next[i]\ \varphi \stackrel{\text{def}}{=} X[i]\ \varphi$
- $next\_a![i..j]\varphi \stackrel{\text{def}}{=} (X![i]\varphi) \wedge \ldots \wedge (X![j]\varphi)$
- $next\_a[i..j]\varphi \stackrel{\text{def}}{=} (X[i]\varphi) \wedge \ldots \wedge (X[j]\varphi)$
- $next\_e![i..j]\varphi \stackrel{\text{def}}{=} (X![i]\varphi) \vee \ldots \vee (X![j]\varphi)$
- $next\_e[i..j]\varphi \stackrel{\text{def}}{=} (X[i]\varphi) \vee \ldots \vee (X[j]\varphi)$

- $next\_event!(b)(\varphi) \stackrel{\text{def}}{=} [\neg b\ U\ b \wedge \varphi]$
- $next\_event(b)(\varphi) \stackrel{\text{def}}{=} [\neg b\ W\ b \wedge \varphi]$
- $next\_event!(b)[k](\varphi) \stackrel{\text{def}}{=}$
  $next\_event!(b)\ \overbrace{(X!\ next\_event!(b)...(X!\ next\_event!(b)}^{k-1\ times}(\varphi))...)$
- $next\_event(b)[k](\varphi) \stackrel{\text{def}}{=}$
  $next\_event(b)\overbrace{(Xnext\_event(b)...(Xnext\_event(b)}^{k-1\ times}(\varphi))...)$
- $next\_event\_a!(b)[k..l](\varphi) \stackrel{\text{def}}{=} next\_event!(b)[k](\varphi) \wedge ... \wedge next\_event!(b)[l](\varphi)$
- $next\_event\_a(b)[k..l](\varphi) \stackrel{\text{def}}{=} next\_event(b)[k](\varphi) \wedge ... \wedge next\_event(b)[l](\varphi)$
- $next\_event\_e!(b)[k..l](\varphi) \stackrel{\text{def}}{=} next\_event!(b)[k](\varphi) \vee ... \vee next\_event!(b)[l](\varphi)$
- $next\_event\_e(b)[k..l](\varphi) \stackrel{\text{def}}{=} next\_event(b)[k](\varphi) \vee ... \vee next\_event(b)[l](\varphi)$

- $r(\varphi) \stackrel{\text{def}}{=} r \mapsto \varphi$
- $r \Rightarrow \varphi \stackrel{\text{def}}{=} \{r; true\} \mapsto \varphi$

- φ abort $b \stackrel{\text{def}}{=} \varphi$ async_abort b

B.4.3 Parameterized SEREs and formulas

Let r be a SERE, and l, m be integers. Let S be a set of constants, integers or boolean values and p an identifier. The left-hand side of the following are SEREs, equivalent to the SEREs on the right-hand side:

- for p *in* S : $| \ r \stackrel{\text{def}}{=} \ \mathop{|}_{s \in S} \{r[p \leftarrow s]\}$.
- for $p\langle l..m\rangle$ *in* S : $| \ r \stackrel{\text{def}}{=} \ \mathop{|}_{s_l \in S} \ldots \mathop{|}_{s_m \in S} \{r[p\langle l..m\rangle \leftarrow \langle s_l..s_m\rangle]\}$
- for p *in* S : && $r \stackrel{\text{def}}{=} \ \mathop{\&\&}_{s \in S} \{r[p \leftarrow s]\}$.

- for $p\langle l..m\rangle$ *in* S : && $r \stackrel{\text{def}}{=} \underset{s_l\in S}{\&\&} \ldots \underset{s_m\in S}{\&\&} \{r[p\langle l..m\rangle \leftarrow \langle s_l..s_m\rangle]\}$
- for p *in* S : & $r \stackrel{\text{def}}{=} \underset{s\in S}{\&} \{r[p \leftarrow s]\}$.
- for & $p\langle l..m\rangle$ *in* S : & $r \stackrel{\text{def}}{=} \underset{s_l\in S}{\&} \ldots \underset{s_m\in S}{\&} \{r[p\langle l..m\rangle \leftarrow \langle s_l..s_m\rangle]\}$

where $r[p \leftarrow s]$ is the SERE obtained from r by replacing every occurrence of p by s and $r[p\langle l..m\rangle \leftarrow \langle s_l..s_m\rangle]$ is the SERE obtained from r by replacing every occurrence of p_j with s_j for all j such that $l \le j \le m$.

Let f be a PSL formula, and l, m integers. Let S be a set of constants, integers or boolean values and p an identifier. The left-hand side of the following are PSL formulas equivalent to the PSL formulas on the right-hand side:

- for p *in* S : $\vee$ $f \stackrel{\text{def}}{=} \bigvee_{s\in S} f[p \leftarrow s]$
- for $p\langle l..m\rangle$ *in* S : $\vee$ $f \stackrel{\text{def}}{=} \bigvee_{s_l\in S} \ldots \bigvee_{s_m\in S} f[p\langle l..m\rangle \leftarrow \langle s_l..s_m\rangle]$
- for p *in* S : $\wedge$ $f \stackrel{\text{def}}{=} \bigwedge_{s\in S} f[p \leftarrow s]$
- for $p\langle l..m\rangle$ *in* S : $\wedge$ $f \stackrel{\text{def}}{=} \bigwedge_{s_l\in S} \ldots \bigwedge_{s_m\in S} f[p\langle l..m\rangle \leftarrow \langle s_l..s_m\rangle]$
- forall p *in* S : $f \stackrel{\text{def}}{=}$ for p *in* S : $\wedge$ f
- forall $p\langle l..m\rangle$ *in* S : $f \stackrel{\text{def}}{=}$ for $p\langle l..m\rangle$ *in* S : $\wedge$ f

where $f[p \leftarrow s]$ is the formula obtained from f by replacing every occurrence of p by s and $f[p\langle l..m\rangle \leftarrow \langle s_l..s_m\rangle]$ is the formula obtained from f by replacing every occurrence of p_j with s_j for all j such that $l \le j \le m$.

B.5 Rewriting rules for clocks

In Section B.3.1, we gave the semantics of clocked FL formulas directly. There is an equivalent definition in terms of unclocked FL formulas, as follows: Starting from the outermost clock, use the following rules to translate clocked SEREs into unclocked SEREs, and clocked FL formulas into unclocked FL formulas.

The rewrite rules for SEREs are:

1. $\mathcal{R}^c(\{r\}) = \mathcal{R}^c(r)$
2. $\mathcal{R}^c(b) = \neg c[*]; c \wedge b$
3. $\mathcal{R}^c(r_1 \,;\, r_2) = \mathcal{R}^c(r_1) \,;\mathcal{R}^c(r_2)$

4. $\mathcal{R}^c(r_1 : r_2) = \{\mathcal{R}^c(r_1)\} : \{\mathcal{R}^c(r_2)\}$
5. $\mathcal{R}^c(r_1 \mid r_2) = \{\mathcal{R}^c(r_1)\} \mid \{\mathcal{R}^c(r_2)\}$
6. $\mathcal{R}^c(r_1 \;\&\&\; r_2) = \{\mathcal{R}^c(r_1)\} \;\&\&\; \{\mathcal{R}^c(r_2)\}$
7. $\mathcal{R}^c([*0]) = [*0]$
8. $\mathcal{R}^c(r[*]) = \{\mathcal{R}^c(r)\}[*]$
9. $\mathcal{R}^c(r@c_1) = \mathcal{R}^{c_1}(r)$

The rewrite rules for FL formulas are:

1. $\mathcal{F}^c((\varphi)) = (\mathcal{F}^c(\varphi))$
2. $\mathcal{F}^c(b!) = [\neg c \; U \; (c \wedge b)]$
3. $\mathcal{F}^c(b) = [\neg c \; W \; (c \wedge b)]$
4. $\mathcal{F}^c(\neg\varphi) = \neg\mathcal{F}^c(\varphi)$
5. $\mathcal{F}^c(\varphi \wedge \psi) = (\mathcal{F}^c(\varphi) \wedge \mathcal{F}^c(\psi))$
6. $\mathcal{F}^c(X!\varphi) = [\neg c \; U \; (c \wedge X! \; [\neg c \; U \; (c \wedge \mathcal{F}^c(\varphi))])]$
7. $\mathcal{F}^c(\varphi \; U \; \psi) = [(c \rightarrow \mathcal{F}^c(\varphi)) \; U \; (c \wedge \mathcal{F}^c(\psi))]$
8. $\mathcal{F}^c(r \mapsto \varphi) = \mathcal{R}^c(r) \mapsto \mathcal{F}^c(\varphi)$
9. $\mathcal{F}^c(r!) = \mathcal{R}^c(r)!$
10. $\mathcal{F}^c(r) = \mathcal{R}^c(r)$
11. $\mathcal{F}^c(\varphi@c_1) = \mathcal{F}^{c_1}(\varphi)$
12. $\mathcal{F}^c(\varphi \;\mathsf{async\_abort}\; b) = \mathcal{F}^c(\varphi) \;\mathsf{async\_abort}\; b$
13. $\mathcal{F}^c(\varphi \;\mathsf{sync\_abort}\; b) = \mathcal{F}^c(\varphi) \;\mathsf{sync\_abort}\; (b \wedge c)$

C

Operator Precedence

The table below gives the order of precedence of the operators as well as their associativity. Here `next*` and `next_event*` stand for all the variations of the `next` and `next_event` operators, and `until*` and `before*` stand for all the variations of the `until` and `before` operators.

	Operator	Associativity
High	HDL operators of the base flavor (e.g. `&&` and `||`) according to their precedence in the base flavor	
	`union`	left
	`@`	left
	`[*]` `[+]` `[=]` `[->]`	left
	`within`	left
	`&` `&&` (SERE and's)	left
	`|` (SERE or)	left
	`:`	left
	`;`	left
	`abort` `async_abort` `sync_abort`	left
	`next*` `next_event*` `eventually!`	right
	`until*` `before*`	right
	`|->` `|=>`	right
	`->` `<->`	right
Low	`always` `never`	right

D

Quick Reference

D.1 Logical operators

D.1.1 Verilog, SystemVerilog and SystemC flavors

Here `b` is a Boolean expression, `p`, `q` properties, `L` a list of values, `j`, `k` integers, and `x` an identifier; `p(x)` indicates a property `p` that uses identifier `x`.

Property	Intuitive Meaning
`b`	`b` holds at the current cycle
`!p`	`p` does not hold at the current cycle
`p && q`	both `p` and `q` hold at the current cycle
`p || q`	either `p` or `q` holds at the current cycle
`p -> q`	if `p` holds at the current cycle then `q` holds at the current cycle as well
`p <-> q`	shortcut for `(p -> q) && (q -> p)`
`for x in boolean: && p(x)`	shortcut for `p(‘true) && p(‘false)`
`for x in {L}: && p(x)`	shortcut for $\texttt{p(l}_1\texttt{) \&\& p(l}_2\texttt{) \&\& ... \&\& p(l}_n\texttt{)}$ where $\texttt{l}_1$, $\texttt{l}_2$, etc. are items from list `L`
`for x in {j:k}: && p(x)`	shortcut for `p(j) && p(j+1) && ... && p(k)`
`for x in boolean: || p(x)`	shortcut for `p(‘true) || p(‘false)`
`for x in {L}: || p(x)`	shortcut for $\texttt{p(l}_1\texttt{) || p(l}_2\texttt{) ... || p(l}_n\texttt{)}$ where $\texttt{l}_1$, $\texttt{l}_2$, etc. are items from list `L`
`for x in {j:k}: || p(x)`	shortcut for `p(j) || p(j+1) ... || p(k)`

D.1.2 Logical operators in the VHDL flavor

Here `b` is a Boolean expression, `p`, `q` properties, `L` a list of values, `j`, `k` integers, and `x` an identifier; `p(x)` indicates a property `p` that uses identifier `x`.

Property	Intuitive Meaning
`b`	`b` holds at the current cycle
`not p`	`p` does not hold at the current cycle
`p and q`	both `p` and `q` hold at the current cycle
`p or q`	either `p` or `q` holds at the current cycle
`p -> q`	if `p` holds at the current cycle then `q` holds at the current cycle as well
`p <-> q`	shortcut for `(p -> q) and (q -> p)`
`for x in boolean: and p(x)`	shortcut for `p('true) and p('false)`
`for x in {L}: and p(x)`	shortcut for `p(`l_1`) and p(`l_2`) and ... and p(`l_n`)` where `l`$_1$, `l`$_2$, etc. are items from list `L`
`for x in {j:k}: and p(x)`	shortcut for `p(j) and p(j+1) and ... and p(k)`
`for x in boolean: or p(x)`	shortcut for `p('true) or p('false)`
`for x in {L}: or p(x)`	shortcut for `p(`l_1`) or p(`l_2`) ... or p(`l_n`)` where `l`$_1$, `l`$_2$, etc. are items from list `L`
`for x in {j:k}: or p(x)`	shortcut for `p(j) or p(j+1) ... or p(k)`

D.1.3 Logical operators in the GDL flavor

Here `b` is a Boolean expression, `p`, `q` properties, `L` a list of values, `j`, `k` integers, and `x` an identifier; `p(x)` indicates a property `p` that uses identifier `x`.

Property	Intuitive Meaning
`b`	`b` holds at the current cycle
`!p`	`p` does not hold at the current cycle
`p & q`	both `p` and `q` hold at the current cycle
`p | q`	either `p` or `q` holds at the current cycle
`p -> q`	if `p` holds at the current cycle then `q` holds at the current cycle as well
`p <-> q`	shortcut for `(p -> q) & (q -> p)`
`for x in boolean: & p(x)`	shortcut for `p(‘true) & p(‘false)`
`for x in {L}: & p(x)`	shortcut for $p(l_1)$ & $p(l_2)$ & ... & $p(l_n)$ where l_1, l_2, etc. are items from list `L`
`for x in {j:k}: & p(x)`	shortcut for `p(j) & p(j+1) & ... & p(k)`
`for x in boolean: | p(x)`	shortcut for `p(‘true) | p(‘false)`
`for x in {L}: | p(x)`	shortcut for $p(l_1)$ | $p(l_2)$ ... | $p(l_n)$ where l_1, l_2, etc. are items from list `L`
`for x in {j:k}: | p(x)`	shortcut for `p(j) | p(j+1) ... | p(k)`

D.2 LTL style

D.2.1 `always`, `never` and `eventually!`

Here `p` and `q` are properties.

Property	Intuitive Meaning
`always p`	`p` holds at the current cycle and at all future cycles
`never p`	`p` does not hold at the current cycle, nor does it hold at some future cycle
`eventually! p`	`p` holds at the current cycle or at some future cycle

D.2.2 The `next*` operators

Here `p` is a property and `m` and `n` are integers such that `m`$\geq$`1` and `n`$\geq$`m`.

Property	Intuitive Meaning
`next! p`	`p` holds in the next cycle, and there must be such a cycle
`next p`	`p` holds in the next cycle, if there is such a cycle
`next![n] p`	`p` holds on the `n`th next cycle, and there must be such a cycle
`next[n] p`	`p` holds on the `n`th next cycle, if there is such a cycle
`next_e![m:n](p)`	`p` holds on one of the next `m`th through `n`th cycles, and there must be such a cycle
`next_e[m:n](p)`	`p` holds on one of the next `m`th through `n`th cycles, if there are at least `n` next cycles
`next_a![m:n](p)`	`p` holds on all of the next `m`th through `n`th cycles, and there must be at least `n` next cycles
`next_a[m:n](p)`	`p` holds on all of the next `m`th through `n`th cycles, however many exist

D.2.3 The `next_event*` operators

Here `b` is a Boolean expression, `p` is a property, and `m` and `n` are integers such that `m`$\geq$`1` and `n`$\geq$`m`.

Property	Intuitive Meaning
`next_event!(b)(p)`	`p` holds at the next cycle `b` holds, and there must be such a cycle
`next_event(b)(p)`	`p` holds at the next cycle `b` holds, if there is such a cycle
`next_event(b)![n](p)`	`p` holds at the `n`th next cycle in which `b` holds, and there must be such a cycle
`next_event(b)[n](p)`	`p` holds at the `n`th next cycle in which `b` holds, if there is such a cycle
`next_event_e!(b)[m:n](p)`	`p` holds at one of the next `m`th through `n`th cycles in which `b` holds, and there must be such a cycle
`next_event_e(b)[m:n](p)`	`p` holds at one of the next `m`th through `n`th cycles in which `b` holds, if there are at least `n` such cycles
`next_event_a!(b)[m:n](p)`	`p` holds at all of the next `m`th through `n`th cycles in which `b` holds, and there must be at least `n` such cycles
`next_event_a(b)[m:n](p)`	`p` holds at all of the next `m`th through `n`th cycles in which `b` holds, however many exist

D.2.4 The `until*` and `before*` operators

Here `p` and `q` are properties.

Property	Intuitive Meaning
`p until! q`	`p` holds until the cycle where `q` holds, and `q` eventually holds
`p until q`	`p` holds until the cycle where `q` holds; if `q` never holds, `p` holds forever (until the end of the trace)
`p until!_ q`	`p` holds until the cycle where `q` holds, inclusive, and `q` eventually holds
`p until_ q`	`p` holds until the cycle where `q` holds, inclusive; if `q` never holds, `p` holds forever (until the end of the trace)
`p before! q`	`p` holds strictly before the cycle where `q` holds, and `p` eventually holds
`p before q`	`p` holds strictly before the cycle where `q` holds; if `p` never holds, then neither does `q`
`p before!_ q`	`p` holds before or at the same cycle where `q` holds, and `p` eventually holds
`p before_ q`	`p` holds before or at the same cycle where `q` holds; if `p` never holds, then neither does `q`

D.2.5 Abort operators

Here `b` is a Boolean expression and `p` is a property.

Property	Intuitive Meaning
`p async_abort b`	either `p` holds or up until `b` holds, `p` does not fail; `b` recognized asynchronously
`p sync_abort b`	either `p` holds or up until `b` holds, `p` does not fail; `b` recognized with respect to current clock context
`p abort b`	equivalent to `p async_abort b`

D.2.6 LTL operators

The Foundation Language is based on the temporal logic LTL. PSL supports the LTL operators shown in the table below. Here `p` and `q` are properties.

LTL operator	Synonym for
`G p`	`always p`
`F p`	`eventually! p`
`X! p`	`next! p`
`X p`	`next p`
`p U q`	`p until! q`
`p W q`	`p until q`

D.3 SERE style

D.3.1 Consecutive repetition operators

Here `b` is a Boolean expression, `s` is a SERE, and `i,j` are integers such that `i` ≥ 0 and `j` $\geq$ `i`.

SERE	Intuitive Meaning
`b[*i]`	`i` consecutive repetitions of `b`
`b[*i:j]`	between `i` to `j` consecutive repetitions of `b`
`b[*i:inf]`	at least `i` consecutive repetitions of `b`
`b[*]`	zero or more consecutive repetitions of `b`
`b[+]`	one or more consecutive repetitions of `b`
`s[*i]`	`i` consecutive repetitions of `s`
`s[*i:j]`	between `i` to `j` consecutive repetitions of `s`
`s[*i:inf]`	at least `i` consecutive repetitions of `s`
`s[*]`	zero or more consecutive repetitions of `s`
`s[+]`	one or more consecutive repetitions of `s`
`‘true`	skip one cycle
`[*i]`	skip exactly `i` cycles
`[*i:j]`	skip between `i` to `j` cycles
`[*i:inf]`	skip at least `i` cycles
`[*]`	skip zero or more cycles
`[+]`	skip one or more cycles

D.3.2 Non-consecutive and goto repetition operators.

Here `b` is a Boolean expression and `i,j,m,n` are integers such that `i` ≥ 0, `j` $\geq$ `i`, `m` ≥ 1 and `n` $\geq$ `m`.

SERE	Intuitive Meaning
`b[=i]`	`i` not necessarily consecutive repetitions of `b` equivalent to `{!b[*]; b}[*i];!b[*]`
`b[=i:j]`	at least `i` and no more than `j` not necessarily consecutive repetitions of `b` equivalent to `{!b[*]; b}[*i:j];!b[*]`
`b[=i:inf]`	at least `i` not necessarily consecutive repetitions of `b` equivalent to `{!b[*]; b}[*i:inf];!b[*]`
`b[->m]`	`m` not necessarily consecutive repetitions of `b`, and `b` holds at the last cycle equivalent to `{!b[*]; b}[*m]`
`b[->m:n]`	at least `m` and no more than `n` not necessarily consecutive repetitions of `b`, and `b` holds at the last cycle equivalent to `{!b[*]; b}[*m:n]`
`b[->m:inf]`	at least `m` not necessarily consecutive repetitions of `b`, and `b` holds at the last cycle equivalent to `{!b[*]; b}[*m:inf]`
`b[->]`	shortcut for `b[->1]` equivalent to `{!b[*]; b}`

D.3.3 Other SERE operators

Here `s` and `t` are SEREs, `L` is a list of values, `j` and `k` are integers, and `x` is an identifier; `s(x)` indicates a SERE `s` that uses the identifier `x`.

SERE	Intuitive Meaning
`s ; t`	match of `s` followed by match of `t`, `t` starts the *cycle after* `s` ends
`s : t`	match of `s` followed by match of `t`, `t` starts the *same cycle* that `s` ends
`s | t`	match of `s` *or* match of `t`
`s && t`	match of `s` *and* match of `t`, lengths are the same
`s & t`	match of `s` *and* match of `t`, lengths may be different
`s within t`	match of `s` within sequence of cycles matching `t`, shortcut for {`[*] ; s ; [*]`} `&&` {`t`}
`for x in boolean: | s(x)`	shortcut for `s(‘true) | s(‘false)`
`for x in {L}: | s(x)`	shortcut for $s(l_1)$ `|` $s(l_2)$ `|` ... `|` $s(l_n)$ where l_1, l_2, etc. are items from list `L`
`for x in {j:k}: | s(x)`	shortcut for `s(j) | s(j+1) | ... | s(k)`
`for x in boolean: && s(x)`	shortcut for `s(‘true) && s(‘false)`
`for x in {L}: && s(x)`	shortcut for $s(l_1)$ `&&` $s(l_2)$ `&&` ... `&&` $s(l_n)$ where l_1, l_2, etc. are items from list `L`
`for x in {j:k}: && s(x)`	shortcut for `s(j) && s(j+1) && ... && s(k)`
`for x in boolean: & s(x)`	shortcut for `s(‘true) & s(‘false)`
`for x in {L}: & s(x)`	shortcut for $s(l_1)$ `&` $s(l_2)$ `&` ... `&` $s(l_n)$ where l_1, l_2, etc. are items from list `L`
`for x in {j:k}: & s(x)`	shortcut for `s(j) & s(j+1) & ... & s(k)`

D.3.4 Common SERE style properties

Here `s` and `t` are SEREs, and `p` is a property.

SERE	Intuitive Meaning
`never t`	there is never a match of `t`
`s |=> t!`	*if* there is a match of `s`, *then* there is a match of `t` on the suffix of the trace • `t` starts the *cycle after* match of `s` ends • *every* match of `s` must see `t` • the match of `t` must reach its end
`s |=> t`	*if* there is a match of `s`, *then* there is a match of `t` on the suffix of the trace • `t` starts the *cycle after* match of `s` ends • *every* match of `s` must see `t` • the match of `t` may "get stuck" in the middle, for instance in a starred subsequence
`s |-> t!`	*if* there is a match of `s`, *then* there is a match of `t` on the suffix of the trace • `t` starts the *same cycle* that match of `s` ends • *every* match of `s` must see `t` • the match of `t` must reach its end
`s |-> t`	*if* there is a match of `s`, *then* there is a match of `t` on the suffix of the trace • `t` starts the *same cycle* that match of `s` ends • *every* match of `s` must see `t` • the match of `t` may "get stuck" in the middle, for instance in a starred subsequence
`s |=> p`	*if* there is a match of `s`, *then* `p` holds on the suffix of the trace • suffix starts the *cycle after* match of `s` ends • *every* match of `s` must see `p`
`s |-> p`	*if* there is a match of `s`, *then* `p` holds on the suffix of the trace • suffix starts the *same cycle* that match of `s` ends • *every* match of `s` must see `p`

D.4 Clocking

D.4.1 Clocking properties

Here `p` is a property and `c` is a Boolean expression.

Clock operator	Intuitive Meaning
`p@rose(c)`	filters out all but the cycles on which `rose(c)` holds
`p@(posedge c)`	same as `p@rose(c)`
`p@fell(c)`	filters out all but the cycles on which `fell(c)` holds
`p@(negedge c)`	same as `p@fell(c)`
`p@c`	filters out all but the cycles on which `c` holds

D.4.2 Clocking SEREs

Here `s` is a SERE and `c` is a Boolean expression.

Clock operator	Intuitive Meaning
`s@rose(c)`	filters out all but the cycles on which `rose(c)` holds
`s@(posedge c)`	same as `s@rose(c)`
`s@fell(c)`	filters out all but the cycles on which `fell(c)` holds
`s@(negedge c)`	same as `s@fell(c)`
`s@c`	filters out all but the cycles on which `c` holds

D.5 Boolean, modeling and verification layers

D.5.1 Built-in functions concerning time

Here `A` is of any type, `n` is a number, `c` is a clock expression, `b` is a bit vector and `s` is a SERE.

Built-in Function	Intuitive Meaning
`prev(A)`	value of `A` at the previous cycle with respect to its clock context
`prev(A,n)`	value of `A` at the $\mathtt{n}^{th}$ previous cycle with respect to its clock context
`prev(A,n,c)`	value of `A` at the $\mathtt{n}^{th}$ previous cycle with respect to clock context `c`
`next(A)`	value of `A` at the next cycle regardless of its clock context
`stable(A)`	true iff value of `A` is the same as it was at previous cycle with respect to its clock context
`stable(A,c)`	true iff value of `A` is the same as it was at previous cycle with respect to clock context `c`
`rose(b)`	true iff value of `b` is `1` and was `0` at the previous cycle with respect to its clock context
`rose(b,c)`	true iff value of `b` is `1` and was `0` at the previous cycle with respect to clock context `c`
`fell(b)`	true iff value of `b` is `0` and was `1` at the previous cycle with respect to its clock context
`fell(b,c)`	true iff value of `b` is `0` and was `1` at the previous cycle with respect to clock context `c`
`ended(s)`	true iff `s` completes at the current cycle with respect to its clock context
`ended(s,c)`	true iff `s` completes at the current cycle with respect to clock context `c`

D.5.2 Other built-in functions and the union operator

Here `A` and `B` are of any type, `n` is a number, `c` is a clock expression, `V` is a bit vector, and `L` is a list of values.

Built-in Function	Intuitive Meaning
`isunknown(V)`	true iff any bit of `V` has an unknown value
`countones(V)`	number of bits in `V` that have the value `1`
`onehot(V)`	true iff `V` contains exactly one bit with the value `1`
`onehot0(V)`	true iff `V` contains at most one bit with the value `1`
`A union B`	nondeterministic choice between `A` and `B`
`nondet({L})`	nondeterministic choice of a value from list `L`
`nondet_vector(n,{L})`	an array of length `n` whose elements are chosen nondeterministically from list `L`

D.5.3 Verification directives

Here `p` is a property, `s` is a SERE, `b` and `c` are Boolean expressions, `msg` is a string, `lname` is an identifier and `D` is a directive.

Directive	Brief Description
`assert p`	verify that `p` holds
`assert p report msg`	verify that `p` holds, report `msg` if it does not
`assume p`	constrain verification so that `p` holds
`assume_guarantee p`	constrain verification so that `p` holds, verify that `p` holds in the driving block(s)
`assume_guarantee p` `report msg`	constrain verification so that `p` holds, verify that `p` holds in the driving block(s), report `msg` if it does not
`restrict s`	constrain verification so that entire trace matches `s`
`restrict_guarantee s`	constrain verification so that entire trace matches `s`, verify that `s` holds in the driving block(s)
`restrict_guarantee s` `report msg`	constrain verification so that entire trace matches `s`, verify that `s` holds in the driving block(s), report `msg` if it does not
`cover s`	check that `s` was covered by the verification suite
`cover s report msg`	check that `s` was covered by the verification suite, report `msg` if it was
`fairness b`	constrain verification so that `b` holds infinitely many times
`strong fairness b,c`	constrain verification so that either `b` holds finitely many times or `c` holds infinitely many times
`lname: D`	identify label `lname` with `D`, and verify, constrain, etc. as per `D`

D.5.4 Verification units

Here `name` is an identifier and `mod` is a module or module instance.

Construct	Brief Description
`vunit name {...}`	group directives and modeling code
`vunit name(mod) {...}`	group directives and modeling code, bind to module `mod`
`vmode name {...}`	same as `vunit name{...}`, but cannot contain assert directives
`vmode name(mode) {...}`	same as `vunit name(mod){...}`, but cannot contain assert directives
`vprop name {...}`	same as `vunit name{...}`, but may contain only assert directives
`vprop name(mode) {...}`	same as `vunit name(mod){...}`, but may contain only assert directives

D.6 Some convenient constructs

D.6.1 Comments and Macros

Here `x` is an identifier, `L` is a statically computable list and `|L|` is the size of the list `L`.

Macro	Brief Description
`// ... <eol>`	trailing comment SystemC, SystemVerilog and Verilog flavors
`-- ... <eol>`	trailing comment VHDL and GDL flavors
`/* ... */`	block comment SystemC, SystemVerilog, Verilog and GDL flavors
`‘define, ‘ifdef`	compiler directives Verilog and SystemVerilog flavors
`#define, #ifdef`	compiler directives VHDL, SystemC and GDL flavors

`%for x in {L} do`
`...`
`%end`
	replicate the text |L| times, each time replace
	the occurrence of `x` with an item from `L`
	all flavors

`%if expr %then`
`...`
`%else`
`...`
`%end`
	similar to the `#if` construct of the cpp preprocessor
	used when encapsulated by `%for`
	all flavors

D.6.2 Named properties and SEREs

Here `name` is an identifier, `type_x`,`type_y` are formal parameter types, `param1,...,paramN` are formal parameters and `actual1,...,actualN` are actual parameters.

Syntax	Brief Description

`property name(type_x param1, param2;`
`type_y param3, ..., paramN) =`
`property_text;`
	property declaration

`name(actual1, actual2, actual3, ..., actualN);`	property instantiation

`sequence name(type_x param1, param2;`
`type_y param3, ..., paramN) =`
`sequence_text;`
	sequence declaration

`name(actual1, actual2, actual3, ..., actualN);`	sequence instantiation

Formal parameter types

The table below gives a description of the parameter types that can be used in the declaration of a property or SERE.

Syntax	Brief Description
`boolean`	a Boolean Expression
`bit`	a single bit
`bitvector`	a vector composed of bits
`numeric`	any expression interpretable as an integer in the underlying flavor
`string`	a string
`sequence`	a braced SERE, a clocked SERE, a repeated SERE or an instantiation of a named SERE
`property`	a PSL property

D.6.3 The `forall` operator

Here `x` is an identifier, `j` and `k` are integers, and `L` a list of values; `p(x)` indicates a property `p` that uses the identifier `x`.

Syntax	Brief Description
`forall x in boolean: p(x)`	shortcut for `p(‘true) && p(‘false)`
`forall x in {L}: p(x)`	shortcut for `p(l1) && p(l2) && ... && p(ln)` where `l1`, `l2`, etc. are items from list `L`
`forall x in {j:k}: p(x)`	shortcut for `p(j) && p(j+1) && ... && p(k)`

NOTE: In replicated properties using `forall`, `x` can be a vector. In such a case, each element of `x` is treated independently. For example, the property

```
forall x[0:7] in boolean:
       always ((read && data[0:7]==x[0:7]) ->
              next_event(write)(data[0:7]==x[0:7]))
```

is equivalent to the “and” of 256 properties, one for each possible value of `x[0:7]`. Similarly `x` can be a vector in parametrized properties and SEREs as well.

Bibliographic Notes

Below we give a brief history of PSL. Our aim is not to give a complete chronicle of the history of temporal logic, nor a full accounting of the history of assertions in hardware design. Furthermore, we will not list each of the many people who participated in one or more of the Accellera and IEEE committees involved in the development of PSL – their names appear in the Accellera and IEEE standards. Rather, our aim is to touch on the major milestones in the development of the language, and the main personalities and ideas that have influenced PSL from its beginnings as syntactic sugaring of the temporal logic CTL, through the move to an LTL-based paradigm, and concluding with the IEEE standardization in October 2005. For background, we include a few words about the temporal logics CTL and LTL as well.

We have made every effort to refer to all the main relevant works, however we may have missed something. If so, we apologize in advance for the omission and would welcome any corrections and/or comments.

The temporal logics LTL and CTL

The linear time logic LTL was introduced as propositional temporal logic, or PTL, by Amir Pnueli in 1977 [41], and the computation tree logic CTL was first presented by Ed Clarke and Allen Emerson in 1981 [14]. For many years, a debate as to the relative merits of each was conducted in the literature. Moshe Vardi was one of the main players in that debate – see for instance [45]. One of the main arguments is that LTL is easier to use, while CTL is easier to model check.

In 1983, Pierre Wolper argued in [46] that LTL is not expressive enough: the requirement "p holds on every even cycle" is not expressible in LTL (nor is it expressible in CTL). In fact, LTL has the expressive power of star-free regular expressions – see [21].

Development of Sugar at IBM

PSL began its life as Sugar at the IBM Haifa Research Laboratory in the early 1990's. Ilan Beer, Shoham Ben-David and Avner Landver developed Sugar as a syntactic sugaring of CTL, with the intention of making the specification process easier for users of IBM's RuleBase model checker. For instance, the `next_event` operator dates to the early days of Sugar, and `next_event(b)(f)` was at that time defined as $A[\neg b\, W\, b \wedge f]$. The concept of vacuity, about which much has been written since [8, 37, 9, 16, 42, 6, 28, 29, 44, 13], dates to these early days.

Circa 1995, regular expressions were added to the logic [10] using the syntax `{r}(p)`, where `r` is a regular expression and `p` a Sugar property, in a manner reminiscent of PDL [22]. Shortly thereafter, suffix implication – in which both the left- and right-hand sides are regular expressions – was added [7], including both weak and strong regular expressions [17]. Although the motivation was usability and not expressive power, Armoni et al. [5] showed that the addition of regular expressions has the side effect of increasing the expressive power to that of omega-regular expressions. As noted in [12], their proof, for the temporal logic ForSpec, holds for PSL as well.

Originally conceived as a language for formal verification [15, 39], 1997 saw the first use of Sugar in simulation [1].

From Accellera onwards

The Accellera FVTC (Formal Verification Technical Committee) started life in 1998 as the VFV (Verilog Formal Verification) committee of OVI (Open Verilog International). When OVI and VI (VHDL International) merged into Accellera in 2000, the charter of the committee was expanded to include VHDL in addition to Verilog. Although the name includes the term "formal verification", a single specification language for both dynamic (simulation) and static (formal) verification soon became the goal of the committee. The two of us participated in the committee from close to its inception as representatives of the candidate language Sugar.

Very important roles were played by Harry Foster and Erich Marschner, chairman and co-chairman of the FVTC. Both Harry and Erich put in an enormous amount of work behind the scenes driving the standardization process – without them it would not have happened. In addition, Erich's endless patience in hearing out the more vocal members of the committee, his care to solicit the input of the more reticent members, and his documentation of everyone's opinion was greatly appreciated by all.

Leading figures from the academic roots of PSL, Ed Clarke, Allen Emerson and Moshe Vardi, took part in the process, as did over 30 industrial representatives, including both potential users of the language as well as EDA vendors. From very early on, it was decided to choose one of a number of

candidate languages as the base language to be modified and enhanced according to requirements identified by the committee. In addition to Sugar, three candidate languages were donated to Accellera for consideration: CBV from Motorola [31], represented by John Havlicek and Hillel Miller, ForSpec from Intel [5], represented by Roy Armoni, and Temporal e from Verisity [40], represented by David Van Campenhout. The committee judged the candidate languages on the basis of an extensive list of 70 requirements, and on the basis of an example document containing 74 example industry properties, expressed in each of the four languages.

The exact selection process was as follows: two candidate languages out of the four were selected by vote, after which the committee identified desired changes. The donors of the two selected languages (CBV and Sugar) then modified their original proposal as per the requested changes. The final vote, taken in April of 2002, chose Sugar (with 71.4% of the votes) to be the Accellera specification language, renamed PSL.

In between the donation of Sugar in November of 2000 and its selection by the FVTC in April of 2002, a huge amount of time was invested in conducting the technical debate in the committee. The IBM team conducting the debate consisted of the two of us as well as Shoham Ben-David. As a result of the debate, and of the changes requested by the committee during the selection process, the language underwent an evolutionary process during this time.

The most visible of the changes was the move from the branching-time semantics of CTL to the linear-time semantics of LTL, as a result of the very persuasive arguments of Moshe Vardi in favor of linear-time semantics. The work of Monika Maidl [38] was instrumental in allowing the move, as it showed that the vast majority of Sugar properties used in practice could be syntactically transformed from CTL into LTL and vice versa. This meant that while the move was deeply significant from a theoretical point of view, there was little or no impact to the user from a practical point of view, for two reasons. First, because the user's view of the language did not change – the fact that `next_event` was now defined in LTL rather than CTL was transparent to the user in the vast majority of cases (which could be ascertained on the basis of a simple syntactic test). And second, because the same tools could be used to check LTL-based Sugar as CTL-based Sugar, providing they passed the same simple syntactic test. The simple subset of PSL, described in Chapter 9, has its roots in Maidl's common fragment (see also [11]).

Two other very visible additions to the language – support for multiple clocks and the `abort` operator – are the result of requests by Intel, recalling features of its ForSpec temporal logic [5]. Some other important additions dating to this period include the flavor concept, the layered definition of the language (the original definition of Sugar did not include the modeling and verification layers), and the formal definition of finite semantics, augmenting the infinite semantics previously defined. During some of this time, the IBM team was supported by Mike Gordon, whose work on incorporating the formal

semantics of PSL into HOL [25, 26, 27] uncovered some subtle bugs in the formal semantics as originally written.

The first Accellera version of PSL (PSL 1.01) [2] was released in June 2003. Accellera version 1.1 [3], released in June 2004, added a SystemVerilog flavor to the original three flavors (Verilog, VHDL, and GDL). In addition, operator precedence was overhauled and labels and report clauses for directives were added. Accellera version 1.1 also corrected three anomalies present in version 1.01. While these anomalies had minimal influence on users of the language (because they involved corner cases that tools could choose to ignore with little or no impact on the user), it was important that they ultimately be solved, because adherence to the standard is determined by adherence to the formal semantics.

The first anomaly was that originally two kinds of clocks, strong and weak, were defined, but the strength of the clock had only a minimal effect. A solution that eliminated the need for two kinds of clocks was presented in [20], and incorporated into the Accellera version 1.1 formal semantics.

The second anomaly was identified in [4], which showed that the complexity of the `abort` operator as defined in Accellera version 1.01 was problematical. A solution, based on the theory of truncated paths developed in [18], incorporated the semantics suggested by [4] but used a simpler and more elegant notation. This solution was later modified to include SEREs [19], and basic results on the resulting semantics (which were incorporated into the Accellera version 1.1 formal semantics) were documented in [30].

The third anomaly concerned weak SEREs such as {`a ; b[*] ; 'false`} (where `'false` is an expression that does not hold at any cycle), that do not match any sequence of cycles. In the formal semantics of Accellera version 1.01, such a SERE, when used as a property, would not hold on any trace, whereas the intuition and intention was that {`a ; b[*] ; 'false`}, being weak, hold on a sequence of cycles in which `a` is asserted on the first cycle and `b` on all the rest. The solution was based on the framework developed in [18, 19], and was incorporated into the Accellera version 1.1 formal semantics. However, the solution creates a new anomaly, in that it treats the *logical contradiction* `'false` differently from the *structural contradiction* {`a`} `&&` {`a;a`}. A possible solution to this was proposed in [17], which also examines in depth the issue of weak vs. strong temporal operators.

The first IEEE version (IEEE Std 1850-2005) [33] was released in October 2005. In addition to a number of clarifications on various topics, the main changes for IEEE Std 1850-2005 were the addition of a fifth flavor (SystemC), replacement of *endpoints* with the built-in function `ended()`, the addition of variations on the `abort` operator, parameterized properties and SEREs, and the introduction of the keyword `hdltype` to ease interaction with the underlying HDL.

Current status

Any attempt to list tools supporting PSL would quickly become out of date. See http://www.haifa.il.ibm.com/projects/verification/sugar/tools.html for such a list.

References

1. Y. Abarbanel, I. Beer, L. Gluhovsky, S. Keidar, and Y. Wolfsthal. FoCs: Automatic generation of simulation checkers from formal specifications. In *Proc. 12th International Conference on Computer Aided Verification (CAV 2000)*, volume 1855 of *LNCS*, pages 538–542. Springer, 2000.
2. Accellera Property Specification Language Reference Manual (version 1.01). http://www.eda.org/vfv/docs/psl_lrm-1.01.pdf.
3. Accellera Property Specification Language Reference Manual (version 1.1). http://www.eda.org/vfv/docs/PSL-v1.1.pdf.
4. R. Armoni, D. Bustan, O. Kupferman, and M.Y. Vardi. Resets vs. aborts in linear temporal logic. In *Proc. 9th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS 2003)*, volume 2619 of *LNCS*, pages 65–80. Springer, 2003.
5. R. Armoni, L. Fix, A. Flaisher, R. Gerth, B. Ginsburg, T. Kanza, A. Landver, S. Mador-Haim, E. Singerman, A. Tiemeyer, M.Y. Vardi, and Y. Zbar. The ForSpec temporal logic: A new temporal property-specification language. In *Proc. 8th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS 2002)*, volume 2280 of *LNCS*, pages 296–311. Springer, 2002.
6. R. Armoni, L. Fix, A. Flaisher, O. Grumberg, N. Piterman, A. Tiemeyer, and M.Y. Vardi. Enhanced vacuity detection in linear temporal logic. In *Proc. 15th International Conference on Computer Aided Verification (CAV 2003)*, volume 2725 of *LNCS*, pages 368–380. Springer, 2003.
7. I. Beer, S. Ben-David, C. Eisner, D. Fisman, A. Gringauze, and Y. Rodeh. The temporal logic Sugar. In *Proc. 13th International Conference on Computer Aided Verification (CAV 2001)*, volume 2102 of *LNCS*, pages 363–367. Springer, 2001.
8. I. Beer, S. Ben-David, C. Eisner, and Y. Rodeh. Detection of vacuity in ACTL formulas. In *Proc. 9th International Conference on Computer Aided Verification (CAV 1997)*, volume 1254 of *LNCS*, pages 279–290. Springer, 1997.
9. I. Beer, S. Ben-David, C. Eisner, and Y. Rodeh. Efficient detection of vacuity in temporal model checking. *Formal Methods in System Design*, 18(2):141–163, 2001.

10. I. Beer, S. Ben-David, and A. Landver. On-the-fly model checking of RCTL formulas. In *Proc. 10th International Conference on Computer Aided Verification (CAV 1998)*, volume 1427 of *LNCS*, pages 184–194. Springer, 1998.
11. S. Ben-David, D. Fisman, and S. Ruah. The safety simple subset. In *Proc. 1st International Haifa Verification Conference*, volume 3875 of *LNCS*, pages 14–29. Springer, 2005.
12. D. Bustan, D. Fisman, and J. Havlicek. Automata construction for PSL. Technical Report MCS05-04, The Weizmann Institute of Science, May 2005.
13. D. Bustan, A. Flaisher, O. Grumberg, O. Kupferman, and M.Y. Vardi. Regular vacuity. In *Proc. 13th Advanced Research Working Conference on Correct Hardware Design and Verification Methods (CHARME 2005)*, volume 3725 of *LNCS*, pages 191–206. Springer, 2005.
14. E.M. Clarke and E.A. Emerson. Design and synthesis of synchronization skeletons using branching-time temporal logic. In *Logic of Programs Workshop*, volume 131 of *LNCS*, pages 52–71. Springer, 1981.
15. E.M. Clarke, O. Grumberg, and D.A. Peled. *Model Checking*. MIT Press, 1999.
16. Y. Dong, B. Sarna-Starosta, C.R. Ramakrishnan, and S.A. Smolka. Vacuity checking in the modal mu-calculus. In *Proc. 9th International Conference on Algebraic Methodology and Software Technology (AMAST 2002)*, pages 147–162. Springer, 2002.
17. C. Eisner, D. Fisman, and J. Havlicek. A topological characterization of weakness. In *Proc. 24th Annual ACM SIGACT-SIGOPS Symposium on Principles Of Distributed Computing (PODC 2005)*, pages 1–8. ACM, 2005.
18. C. Eisner, D. Fisman, J. Havlicek, Y. Lustig, A. McIsaac, and D. Van Campenhout. Reasoning with temporal logic on truncated paths. In *Proc. 15th International Conference on Computer Aided Verification (CAV 2003)*, volume 2725 of *LNCS*, pages 27–39. Springer, 2003.
19. C. Eisner, D. Fisman, J. Havlicek, and J. Mårtensson. The $\top, \bot$ approach for truncated semantics. Technical Report 2006.01, Accellera, May 2006.
20. C. Eisner, D. Fisman, J. Havlicek, A. McIsaac, and D. Van Campenhout. The definition of a temporal clock operator. In *Proc. 30th International Colloquium on Automata, Languages and Programming (ICALP 2003)*, volume 2719 of *LNCS*, pages 857–870. Springer, 2003.
21. E.A. Emerson. Temporal and modal logic. In *Handbook of Theoretical Computer Science, Volume B*, chapter 16, pages 995–1072. Elsevier Science Publishers and The MIT Press, 1994.
22. M.J. Fischer and R.E. Ladner. Propositional dynamic logic of regular programs. *Journal of Computer and Systems Sciences*, 18:194–211, 1979.
23. H.D. Foster, A.C. Krolnik, and D.J. Lacey. *Assertion Based Design, 2nd Edition*. Kluwer Academic Publishers, 2004.
24. GDL – General Description Language. Available at http://standards.ieee.org/downloads/1850/1850-2005/gdl.pdf.
25. M.J.C. Gordon. Using HOL to study Sugar 2.0 semantics. In *Proc. 15th International Conference on Theorem Proving in Higher Order Logics (TPHOLs 2002, NASA/CP-2002-211736)*, pages 87–100. National Aeronautics and Space Administration, 2002.
26. M.J.C. Gordon. Validating the PSL/Sugar semantics using automated reasoning. *Formal Asp. Comput.*, 15(4):406–421, 2003.

27. M.J.C. Gordon, J. Hurd, and K. Slind. Executing the formal semantics of the Accellera Property Specification Language by mechanised theorem proving. In *Proc. 12th Advanced Research Working Conference on Correct Hardware Design and Verification Methods (CHARME 2003)*, volume 2860 of *LNCS*, pages 200–215. Springer, 2003.
28. A. Gurfinkel and M. Chechik. Extending extended vacuity. In *Proc. 5th International Conference on Formal Methods in Computer-Aided Design (FMCAD 2004)*, volume 3312 of *LNCS*, pages 306–321. Springer, 2004.
29. A. Gurfinkel and M. Chechik. How vacuous is vacuous? In *Proc. 10th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS 2004)*, volume 2988 of *LNCS*, pages 451–466. Springer, 2004.
30. J. Havlicek, D. Fisman, and C. Eisner. Basic results on the semantics of Accellera PSL 1.1. Technical Report 2004.02, Accellera, May 2004.
31. J. Havlicek, N. Levi, H. Miller, and K. Shultz. Extended CBV statement semantics, partial proposal presented to the Accellera Formal Verification Technical Committee, April 2002. At http://www.eda.org/vfv/hm/att-0772/01-ecbv_statement_semantics.ps.gz.
32. IEC/IEEE Standard for Verilog Register Transfer Level Synthesis. IEC/IEEE 62142 (IEEE 1364.1$^{\mathrm{TM}}$).
33. IEEE Standard for Property Specification Language (PSL). IEEE Std 1850$^{\mathrm{TM}}$-2005.
34. IEEE Standard for SystemVerilog. IEEE Std 1800$^{\mathrm{TM}}$-2005.
35. IEEE Standard for VHDL Register Transfer Level (RTL) Synthesis. IEEE Std 1076.6$^{\mathrm{TM}}$.
36. IEEE Standard SystemC Language Reference Manual. IEEE Std 1666$^{\mathrm{TM}}$-2005.
37. O. Kupferman and M.Y. Vardi. Vacuity detection in temporal model checking. In *Proc. 10th Advanced Research Working Conference on Correct Hardware Design and Verification Methods (CHARME 1999)*, volume 1703 of *LNCS*, pages 82–96. Springer, 1999.
38. M. Maidl. The common fragment of CTL and LTL. In *Proc. 41th Annual Symposium on Foundations of Computer Science (FOCS 2000)*, pages 643–652. IEEE Computer Society, 2000.
39. K.L. McMillan. *Symbolic Model Checking*. Kluwer Academic Publishers, 1993.
40. M.J. Morley. Semantics of temporal e. In *Proc. Banff'99 Higher Order Workshop (Formal Methods in Computation)*, 1999. University of Glasgow, Dept. of Computing Science Technical Report.
41. A. Pnueli. The temporal logics of programs. In *Proc. of the Annual IEEE Symposium on Foundations of Computer Science (FOCS 1977)*, pages 46–57. IEEE Computer Society, 1977.
42. M. Purandare and F. Somenzi. Vacuum cleaning CTL formulae. In *Proc. 14th International Conference on Computer Aided Verification (CAV 2002)*, volume 2404 of *LNCS*, pages 485–499. Springer, 2002.
43. RuleBase User's Manual. IBM Haifa Research Laboratory.
44. M. Samer and H. Veith. Parameterized vacuity. In *Proc. 5th International Conference on Formal Methods in Computer-Aided Design (FMCAD 2004)*, volume 3312 of *LNCS*, pages 322–336. Springer, 2004.
45. M.Y. Vardi. Branching vs. linear time: Final showdown. In *Proc. 7th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS 2001)*, volume 2031 of *LNCS*. Springer, 2001.

46. P. Wolper. Temporal logic can be more expressive. *Information and Control*, 56(1/2):72–99, 1983.

Index