Notices

PSL/Sugar Version 1.28 User’s Guide
For information regarding PSL, contact: Gil Shapir (shapir @il.ibm.com), Tel: 972-4-8296258

The RuleBase PE Software and any accompanying support are provided by IBM on an "as-is" basis without warranty of any kind. IBM makes no warranty as to the accuracy, sufficiency or suitability for any purpose of the Software or any support provided with it. IBM does not make any warranties for the Software or any accompanying support, whether express or implied, and disclaims the implied warranties of merchantability or fitness for a particular purpose. IBM does not warrant that designs of chips which users verify with the Software shall be free from defects.

International Business Machines Corporation provides this publication “as is” without warranty of any kind, either express or implied. Some states do not allow disclaimer of express or implied warranties in certain transactions; therefore this statement may not apply to you.

This publication may contain technical inaccuracies or typographical errors. While every precaution has been taken in the preparation of this document, the publisher and author assume no responsibility for errors or omissions. Nor is any liability assumed for damages resulting from the use of the information contained herein. Changes are periodically made to the information herein; these changes will be incorporated in new editions of the publication. IBM may make improvements and/or changes in the product(s) and/or the program(s) described in this publication at any time.

It is possible that this publication may contain references to, or information about, IBM products (machines and programs), programming, or services that are not announced in your country. Such references or information must not be construed to mean that IBM intends to announce such IBM products, programming, or services in your country.

All trademarks and service marks are trademarks of their respective owners.

© Copyright IBM Research Labs in Haifa 2005. All rights reserved.

This product and portions thereof is manufactured under license from Carnegie Mellon University.
# Table of Contents

Describing the Environment ........................................................................................................ 5

1.1 Overview ......................................................................................................................... 5
1.2 GDL Modeling Layer ........................................................................................................ 7
1.3 Arrays ............................................................................................................................... 25
1.4 Sequential Processes ....................................................................................................... 32
1.5 Environment Constraints ............................................................................................... 38
1.6 Linking the Environment to the Design ......................................................................... 47
1.7 Overriding Design Behavior ............................................................................................ 47
1.8 Using Non-determinism and Fairness ............................................................................. 50
1.9 Using Counter Files ........................................................................................................ 57
1.10 Modeling Clocks ............................................................................................................ 57
1.11 Modeling Reset ............................................................................................................. 60

PSL/Sugar – The RuleBase Specification Language ................................................................. 63

2.1 Overview ......................................................................................................................... 63
2.2 Semantic Model .............................................................................................................. 64
2.3 LTL Operators ............................................................................................................... 66
2.4 PSL Operators ............................................................................................................... 74
2.5 Forall operator ............................................................................................................... 93
2.6 The Optional Branching Extension (OBE) .................................................................... 94
2.7 Named Properties, Sequences, and Endpoints ............................................................... 105
2.8 Satellites – More Expressiveness .................................................................................. 113

PSL – Property Examples ...................................................................................................... 115

3.1 Overview ......................................................................................................................... 115
3.2 Basic Properties ............................................................................................................. 115
3.3 Arrays ............................................................................................................................. 117
3.4 Before ............................................................................................................................. 117
3.5 Until ................................................................................................................................. 118
3.6 Forall .......................................................................................................................... 119
3.7 Eventuality .................................................................................................................. 119
3.8 More Sequences ....................................................................................................... 119

Managing VUnits, VModes, VProps and Environments .................................................. 121

4.1 Overview .................................................................................................................. 121
4.2 Defining VUnits, VModes and VProps ................................................................. 122
4.3 Using Vmodes to Limit the Environment .............................................................. 126
4.4 Verification Project Management .......................................................................... 129

Current PSL Support in RuleBase PE ............................................................................ 131
Flavors ............................................................................................................................. 131
Boolean Layer ............................................................................................................... 131
Temporal Layer ............................................................................................................. 132
Verification Layer ........................................................................................................... 132
Modeling Layer ............................................................................................................. 133

PSL 1.0 to 2005 Converter: User Guide ........................................................................ 135
Overview ....................................................................................................................... 135
Preprocessor Directives ............................................................................................... 136
Usage ............................................................................................................................... 137
Conversion ...................................................................................................................... 137
1.1 Overview

This chapter describes the syntax and semantics of GDL constructs, suggests modeling techniques and demonstrates them with some examples. Before starting to write your environments, we recommend you read CHAPTER 4: Managing VUnits, VModes, VProps and Environments.

1.1.1 Describing Environment Models

RuleBase checks the properties specified for every possible input sequence. However, most chips are not designed to accept every possible input sequence. Designers often assume a correct behavior of the target environment and simplify the design by ignoring illegal behaviors.

RuleBase must be made aware of the environment’s legal behavior, otherwise it might produce “false negatives”, which are counter-examples that result from illegal input sequences. The way to specify environment behavior is to write environment models, which are the logic that drives the inputs of the design to be verified.
Every input of the design must be assigned some behavior. Some inputs are kept constant (e.g., configuration inputs), others remain completely free (non-deterministic), while the control signals of interest are usually assigned detailed and exact behavior.

Environment models are written in the RuleBase General Description Language (GDL), a dialect of the SMV language. GDL is somewhat similar to common hardware description languages (HDLs), but it also supports non-determinism and multiple environments.

Environments are linked to the design and to other environments by signal names. Signals produced by the environment will match and drive design signals that have the same name even if they are internal to the design, which is a way to abstract by overriding, described later. Signals (both output and internal signals) produced by the design will match and drive environment models that require these signals. In some translation paths, design signals are converted to upper-case.

Writing good environment models is an art. Good environments should be small and simple, while allowing all and only the legal behaviors. Environments should be small to avoid overloading the model-checker, and simple in order to be easily written, read, and maintained. Good environment models should not produce illegal behavior, or else false-negative results will be produced. On the other hand, they should model all the legal behaviors because an unmodelled behavior is a good place for bugs to hide. An attempt should be made to hide as much detail as possible using abstraction techniques (as explained later.

The following are the stages of environment modeling:

- Study the block interfaces in detail. The behavior of every input and every relevant output must be understood. This information can be gathered from standard bus protocols, design documents, and communication with the designers.
Plan the hierarchical structure of the environment models, grouping related signals and reusing components where possible.

- Decide how to model each input. Some inputs are held constant, at least during the initial stages of verification. Usually there is a set of interesting control inputs that need detailed modeling. We have to design and implement logic to drive these signals.
- Code the logic in GDL.

### 1.2 GDL Modeling Layer

An environment is made up of a few type of statements. These statements are described in the following sections. The order of the statements is unimportant and keywords are not case-sensitive; they can be in either upper or lower case.

All the GDL statements (except for `module` statement) can be written inside any verification mode (vmode), verification unit (vunit) or module, and then its scope duration will be limited by this vunit/vmode/module. If the GDL Modeling Layer statement is written in vmode `default`, its scope duration includes the whole verification project.

### 1.2.1 Expressions

#### 1.2.1.1 Variables and Constants

The basic expressions are numbers, enumerated constants, or variable references.

A number is a decimal if it has only decimal digits and no suffix (e.g., 1276). A binary number consists of binary digits and ends with ‘B’ (e.g., 1011B). A hexadecimal number begins with a decimal digit, has hexadecimal digits, and ends with ‘H’ (e.g., 7FFFH, 0FFH). RuleBase infers the width of constants from the context in which they are used and not from their format. For example, 0010B can be assigned to any bit vector that has at least two bits.
One of the symbolic values a variable can take on is an enumerated constant. For instance, if we declare the following:

```
var state: {idle, st1, st2, st3, waiting};
```

then each of the five symbolic values “idle”, “st1”, “st2”, “st3”, and “waiting” is enumerated constants.

A variable reference has one of the following formats:

- name -- simple variable
- name(number) -- one bit of array
- name(number..number) -- a range of bits

For more information on variables see Section 1.2.2.
For more information on arrays, see in Section 1.3.

### 1.2.1.2 Operators

An expression can be a combination of sub-expressions, connected by operators:

**Boolean connectives:**

```
! expr                   not
expr & expr        and
expr | expr           or
expr ^ expr    (or: expr xor expr)   xor
expr -> expr       implies
expr <-> expr     iff (xnor)
```

(Boolean operations can be applied only to boolean expressions.)

**Relational operators:**

```
expr = expr     equals
```
expr != expr        not equals
expr > expr         greater than
expr >= expr        greater than or equals
expr < expr         less than
expr <= expr        less than or equals
(>, >=, < and <= can be applied only to integer or Boolean expressions.)

Arithmetic operators:
expr - exprmin
expr + exprplus
expr * exprmultiplication
expr / expr division (since / is also legal in a signal name, make sure to surround it with spaces)
expr mod exprmodulo
(Arithmetic operators can be applied only to integer and boolean expressions.)

1.2.1.3 Operator Precedence and Associativity

The following operators are listed in decreasing order of precedence and strength:
++       (concatenation)
!        (not)
prev next
+  -
*  /  mod
= != < <= > >=
&        (and)
|        (or)
xor ^
<>       (iff)
All the operators, except ->, have left to right associativity.

Use parentheses in any case that you don’t know or don’t remember the precedence. Even if you know, others may find explicit parenthesizing easier to read and understand.

1.2.1.4 Case and If Expressions

GDL provides two constructs that express a choice between two or more expressions. They are the case and if expressions, described below.

The case expression has the following format:

```
case
  condition1 : expr1 ;
  condition2 : expr2 ;
  ...
  else : exprn ;
esac
```

A case expression is evaluated as follows: condition\_1 is evaluated first. If it is true, expr\_1 is returned. Otherwise, condition\_2 is evaluated. If it is true, expr\_2 is returned, and so forth. Although the else part is not essential, we recommend you use it as the default entry if you are not certain that the other conditions cover all the cases. Falling through the end of a case statement may have unpredictable results. Notice that from the description of the case expression above, it follows that an earlier condition takes precedence over a later one. That is, if two conditions are true, the first takes precedence.

The if expression is shorthand for a case with two entries. If has the following format:
if condition then exprA else exprB endif

In the above if expression, exprA is returned if condition is true, and exprB is returned if condition is false.

Note: This section deals with if/case expressions rather than statements (if/ case statements are allowed only inside sequential processes. See Section 1.4). You cannot write, for example:
if c then assign a := x; b := y; else assign a := z; b := w; endif;
Instead, you should write:
assign a := if c then x else z endif; b := if c then y else w endif;

1.2.1.5 Non-deterministic Choice

RuleBase uses non-determinism to describe many possible behaviors at once. Section 1.8 describes non-determinism in detail. In this section, the non-deterministic constructs are briefly mentioned for completeness. The non-deterministic constructs of RuleBase have the following format:

{ expr1, expr2, ... , exprn }a non-deterministic choice

expr1 union expr2 another way to express {expr1, expr2}

n1 .. n2 another way to express {n1, n1+1,..., n2}

1.2.1.6 Other Expressions

The following are also expressions:

( expr ) a parenthesized expression
expr in {v1, v2, ... , vn} shorthand for ((expr = v1) | (expr = v2) | ... (expr = vn))
1.2.1.7 Built-in Functions

The built-in functions `fell()` and `rose()` have the following functionality:

- `fell(expr)` is true if `expr` is 0, and was 1 on the previous cycle
- `rose(expr)` is true if `expr` is 1, and was 0 on the previous cycle

*Note: `fell()` and `rose()` can only be applied to boolean expressions.*

The built-in function `prev()` has the following functionality:

- `prev(expr)` - refers to the value of the expression on the previous cycle
- `prev(expr,n)` - refers to the value of the expression on the n\(^{th}\) previous cycle

*Note: `prev()` can only be applied to boolean expressions and boolean vectors.*

On the first cycle of a computation path, `prev(expr)` returns an arbitrary value. Similarly, `prev(expr,n)` returns an arbitrary value if there is no n\(^{th}\) previous cycle. The values of `fell()` and `rose()` on the first cycle are derived from the value of `prev()`, as follows:

- `fell(expr) = prev(expr) & !expr`
- `rose(expr) = !prev(expr) & expr`

The use of `fell, rose, and prev` results in additional state variables, one for each expression to which they refer. However, multiple references to the same variable will only add one extra variable.

The built-in function `next()` has the following functionality:

- `next(expr)` - refers to the value of the expression on the next cycle

*Note: `next(expr)` can only be used within assign next and trans statements.*

*Note: next cannot be nested. next(next(expr)) is illegal.*
Note: next is also a PSL operator, and may appear as such in PSL properties. However, the built-in function next cannot be used inside a PSL property.

1.2.2 Var Statement

A var statement declares variables. It has the following format:

```
var name, name, ... : type;      name, name, ... : type;       ...
```

Note: The variables are always state variables as long as the declaration is not within a sequential process (see Section 4.3).

```
var name, name, ... : type;      name, name, ... : type;       ...
```

The type can be one of the following:
- Boolean
- \{ enum1, enum2, ... \}
- number1 .. number2

(For information on arrays, see Section 1.3.)

For instance, the following are legal var statements:

- `var request, acknowledge: boolean;`
- `var state: {idle, reading, writing, hold};`
- `var counter: {0, 1, 2, 3};`
- `var length: 3 .. 15;`

The first statement declares two variables, “request” and “acknowledge”, to be of type Boolean. The second statement declares a variable called “state” which can take on one of four enumerated values: “idle”, “reading”, “writing”, or
“hold”. The third statement declares a variable called “counter” which can take on the values 0, 1, 2, and 3. The fourth statement declares a variable called “length” which can take on any of the values between 3 and 15, inclusive.

A `var` statement only declares state variables. The `assign` statement, described below, defines the behavior of these variables.

### 1.2.3 Assign Statement

An `assign` statement assigns a value to a variable declared with a `var` statement. It has one of the following formats:

- `assign init(name) = expression;` assigns an initial value to a variable (combinational or state).
- `assign name = expression;` assigns a value to a variable (combinational or state).
- `assign next(name) = expression;` defines the next-state function of a state variable.

A state variable is simply a memory element, or register (flip-flop or latch).

**Note:** Using `assign next` within a sequential process causes the variable to be a state variable (see section 4.3). Variables outside a process are already state variables by definition.

The following are examples of legal `assign` statements:

```plaintext
assign init(state) := idle;
assign next(state) :=
   case
      reset : idle;
      state=idle : { idle, busy };
      state=busy & done : { idle };
```

---

IBM Haifa Research Laboratory, Israel
Provided by special agreement with IBM
else : state;
esac

The keyword assign may be omitted for the second and following consecutive assign statements. Thus, the following:

assign var1 := xyz;
    init(var2) := abc;
    next(var2) := qrs;

is equivalent to:

assign var1 := xyz;
assign init(var2) := abc;
assign next(var2) := qrs;

1.2.4 Define Statement

A define statement is used to give a name to a frequently-used expression, much like a macro in other programming or hardware description languages. The define statement has the following format:

define name := expression;

For instance, the following are legal define statements:

define adef := (q | r) & (t | v);
define bb(0) := q & t;  cc := 3;

As with the assign statement, the keyword define may be omitted in the second, and following, consecutive define statements.
1.2.5 The Difference Between Assign and Define

A state variable (flip-flop or latch) must always be declared with the var statement. If assigned an explicit value, the assign init() and assign next() statements are used (if either is omitted, the initial and next values, respectively, are considered to be completely non-deterministic).

For a combinational variable (output of combinational logic), you may use either assign or define. Users of SMV or of previous versions of RuleBase may recall that there were subtle differences between the assign and define statements which made it more efficient to use one or the other in certain situations. These differences are no longer present in RuleBase, which will convert from one to the other as needed in order to make the model checking more efficient.

Only the following semantic distinctions exist between assign and define:

- **assign** must refer to a variable defined with var.
- **define** must NOT refer to a variable defined with var.
- An assign statement can be thought of as a variable assignment, while a define statement should be thought of as macro text substitution. Thus, in the following:

  ```
  VAR v,v1,v2,d1,d2: boolean;
  assign v := {0,1};
  assign v1 := v;
  assign v2 := v;
  define d := {0,1};
  assign d1 := d;
  assign d2 := d;
  ```

  It is true that v1=v2, because both are equal to the value of the variable v. However, it is not true that d1=d2, because the macro text substitution has made the assignments to d1 and d2 equivalent to:

  ```
  assign d1 := {0,1};
  ```
assign d2 := {0,1};

so that each non-deterministic assignment is completely independent of the other. If you code something similar to the above, RuleBase will issue a warning that a non-deterministic define expression is used multiple times.

1.2.6 Module Statement

An environment file can be totally flat, with no hierarchy at all. In this case, all the statements should be written inside the default verification mode:

```
vmode default
{
    statement;
    statement;
    ...
}
```

VMode default is considered as "global" module - it is implicitly inherited by all the vunits and vmodes in the verification project. However, it is usually more appropriate to write a modular and hierarchical environment. The module and instance statements are used for this purpose.

A module statement is used to define a module that can be instantiated a number of times, as in hardware description languages. It has the following format:

```
module module_name ( inputs ) ( outputs )
{
    statement;
    statement;
    ...
}
```
where \textit{inputs} is a list of formal parameters passed to the module, \textit{outputs} is a list of formal parameters produced by the module, and \textit{statements} is any sequence of \texttt{var}, \texttt{assign}, \texttt{define} and \texttt{instance} statements. The input/output parameters can be thought of as input/output signals. Input parameters are produced elsewhere, and they drive the module, while output parameters are produced by the module itself and can be used elsewhere. A signal that appears as an output parameter of a module must be defined and assigned a value in that module (\texttt{var} or \texttt{define} or \texttt{instance} output). If a signal that appears as an input parameter of a module is not used in that module, RuleBase will issue a warning.

For instance, the following is a legal \texttt{module} statement:

```
module delayed_and (s1, s2) (out) {
  var out : boolean;
  assign
    init(out) := 0;
    next(out) := s1 & s2;
}
```

Modules cannot be \texttt{declared} inside other modules but they can be \texttt{used} (instantiated) by other modules.

\subsection*{1.2.7 \texttt{Instance Statement}}

A \texttt{module} statement is only a definition—it has no effect until it is instantiated (called). The \texttt{instance} statement instantiates a module using the following format:

```
instance instance_name : module_name ( inputs ) ( outputs );
```

where \textit{instance\_name} is the name of the specific instance (one module can be multiply instantiated), \textit{module\_name} is the name of the module being instantiated, \textit{inputs} is a list of expressions passed as inputs to this instance, and \textit{outputs}
is a list of output parameters that connect the instance outputs to real signals of the design or the environment. An instance name is optional.

For example, the following is a legal instance statement, instantiating the two-input and-gate defined in Section 1.2.6:

```
instance da : delayed_and(q,r)(t);
```

### 1.2.8 Scope Rules

Statements inside a module cannot reference variables outside that module (no global symbols for modules). External signals and variables needed by the module must be passed as parameters to the instance. A module can only assign values to external signals and variables by passing them as output parameters.

On the other hand, it is possible to reference internal signals of an instance from outside that instance. For example, if module M has an internal signal Sig, and Ins is an instance of module M, one can refer to signal Sig as Ins/Sig (‘/’ is the hierarchy character). This allows assertions to refer to the internal state of instances without the burden of exporting state variables. It also allows you to easily override parts of existing modules without changing the module definition. For further detail on overriding, see Section 1.6.

### 1.2.9 Comments, Macros, and Preprocessing

There are two types of comments in environment description files:

1. Text beginning with “--” and ending at the end of line.
2. Text beginning with “/*” and ending with “*/”.

RuleBase ignores comment text. You can insert a comment anywhere a space is legal (except in text strings).
Before processing the environment description files, RuleBase calls a standard preprocessor, *cpp*, to filter these files. The mechanisms provided by *cpp* can be used to facilitate the development of environment models. The most useful mechanisms are macros, conditional compilation (*#ifdef*, *#if*, *#endif*, ...) and *#include*. See “man *cpp*” on your unix system for more details.

The *cpp* preprocessor may issue errors when the ‘ character appears inside commented text; therefore, we recommend you avoid the use of this character within comments.

RuleBase provides additional preprocessing abilities in addition to *cpp*. These are the *%for* and *%if* constructs described below.

### 1.2.9.1 %for

The *%for* construct replicates a piece of text a number of times, with the possibility of each replication receiving a parameter. The syntax of the *%for* construct is as follows:

```
%for <var> %in <expr1> .. <expr2> do
...
%end
```

or:

```
%for <var> in <expr1> .. <expr2> step <expr3> do
...
%end
```

-- step can be negative

or:

```
%for <var> in { <item> , <item> , ... , <item> } do
...
%end
```
• <item> is either a number, an identifier, or a string in double-quotes.
• When the value of an item is substituted into the loop body (see below), the double quotes will be stripped.

In the first case, the text inside the %for-%end pair will be replicated expr2-expr1+1 times (assuming that expr2>=expr1). In the second case, the text will be replicated (|expr2-expr1|+1)/expr3 times (if both expr2-expr1 and expr3 are positive or both are negative). In the third case, the text will be replicated according to the number of items in the list.

During each replication of the text, the loop variable value can be substituted into the text as follows. Suppose the loop variable is called “ii”. Then, the current value of the loop variable can be accessed from the loop body using the following three methods:

• The current value of the loop variable can be accessed simply using “ii” if “ii” is a separate token in the text. For instance:

```plaintext
%for ii in 0..3 do
    define aa(ii) := ii > 2;
%end
```

is equivalent to:
```plaintext
define aa(0) := 0 > 2;
define aa(1) := 1 > 2;
define aa(2) := 2 > 2;
define aa(3) := 3 > 2;
```

• If “ii” is part of an identifier, it can be accessed using %{ii} as follows:

```plaintext
%for ii in 0..3 do
    define aa%%{ii} := ii > 2;
%end
```

is equivalent to:
```plaintext
define aa0 := 0 > 2;
define aa1 := 1 > 2;
```
If “\( ii \)” needs to be used as part of an expression, it can be accessed using \%\{<expr>\} as follows:

\[
%\text{for } ii \text{ in } 1..4 \text{ do}
\]
\[
\text{define } aa%\{ii-1\} := %\{ii-1\} > 2;
%\text{end}
\]

is equivalent to:
\[
\text{define } aa0 := 0 > 2;
\text{define } aa1 := 1 > 2;
\text{define } aa2 := 2 > 2;
\text{define } aa3 := 3 > 2;
\]

**Attention:** %for just duplicates the code which appears within it. People not always think about the volume of code, which may be generated by using %for. Having many %for iterations may result in duplicating huge code, which RuleBase PE might not be able to handle.

### 1.2.9.2 %i

The syntax of the %if construct is as follows:

\[
%\text{if } <\text{expr}> %\text{then}
\]
\[
... 
\]
\[
%\text{end}
\]

or:

\[
%\text{if } <\text{expr}> %\text{then}
\]
\[
... 
\]
\[
%\text{else}
\]
The %if construct is similar to the #if construct of the cpp preprocessor. However, %if must be used when <expr> refers to variables defined in an encapsulating %for.

### 1.2.9.3 Operators in Preprocessor Expressions

The following operators can be used in pre-processor expressions:

- `=`  
- `!=`  
- `<`  
- `>`  
- `<=`  
- `>=`  
- `-`  
- `+`  
- `*`  
- `/`  
- `%`

In the current version, operators work only on numeric values, for example, it is acceptable to write:

```plaintext
%for i in 0..3 do  
  %if i != 3 %then  
  +...  
%end
```

But it is not possible to write:

```plaintext
%for command in {read, write} do  
  %if command = read %then-- doesn’t work!
%end
```

### 1.2.10 Smart Comments

RuleBase has additional features that are not part of the PSL 2.0 language. Smart comments enables you to use these features in your verification project.
Syntax:

```--+ <RuleBase feature statement>`
```

Since there is a "--" in the beginning of this line, it will be ignored by all the verification tools other than RuleBase (verification tools that support PSL 2.0 language). We will describe these features in the following chapters.

### 1.2.11 Reserved Words

The following words are keywords and should not be used as identifiers:

**PSL 1.0 version:**

```^ ~ < <=> <= <> = >>= >> | |=> || -> - -> -- , ; : := ! /= /= / .. ( ) [ ] { } * & && + ++ vunit vmode vprop assert define module instance fairness strong var boolean assign case esac endcase if endif then else init next itobv itobvn bvtoi EX AX EF AF EG AG always A E U u X x X! W w G G F f false true next_event next_event! next_event_a! next_event_e! ones within before before! before!_ until! until until!_ xor mod union in fell rose inherit assume assume_guarantee restrict restrict_guarantee cover property sequence abort struct clock const default endpoint eventually! forall never next! next_a next_a! next_e next_e! nodets original override prev process rep zereos strong
```

**PSL IEEE version:**

```^ ~ < <=> <= <> = >>= >> | |=> || -> - -> -- , ; : := ! /= /= / .. ( ) [ ] { } * & && + ++ vunit vmode vprop assert define module instance fairness strong var boolean assign case esac endcase if endif then else init next itobv itobvn bvtoi EX AX EF AF EG AG always A E U X X! W w G G F f false true next_event next_event! next_event_a! next_event_e! ones within before before! before!_ until! until until!_ xor mod union in fell rose inherit assume assume_guarantee restrict restrict_guarantee cover property sequence abort async_abort sync_abort struct clock const default ended end-```
If a keyword is prefixed with the ‘\’ character, it becomes a regular identifier.

1.3 Arrays

It is often convenient to define arrays of state variables and to apply operations to entire arrays or to ranges of indices. Boolean arrays (buses, bundles) are the most common, but other types of arrays (integer sub-range, enumerated constants) are also useful. Hence, RuleBase is primarily oriented toward Boolean arrays, but also supports other types of arrays.

1.3.1 Defining Arrays

An array of state variables is defined as follows:

\begin{verbatim}
var name ( index1 .. index2 ) : type ;
\end{verbatim}

It actually defines (|index2-index1|+1) state variables named name(index1), ..., name(index2), where index1 can be either greater or less than index2.

Examples:

\begin{verbatim}
var
    addr(0..7) : boolean;    -- 8 boolean variables, addr(0), addr(1), ..., addr(7)
    counter(4..5) : 0..3;      -- 2 integer variables, each can have the values 0,1,2,3
    status(3..0) : {empty, notempty, full };   
    -- 4 variables, each can have the values empty, notempty, full
\end{verbatim}

An array can also be defined with a define statement:
define name( index1 .. index2 ) := <expr>;

Example:

define masked_sig(0..3) := sig(0..3) & mask(0..3);

Note that the following line

var x(0..3) : { 5, 7, 13 };

defines an array of four integer variables, each of them can have the values 5, 7, or 13. This is not a non-deterministic bit vector. To define a bit vector and assign to it the three values non-deterministically, do the following:

var x(0..3) : boolean; assign x(0..3) := { 5, 7, 13 };

1.3.2 Operations on Arrays

Reference:

The simplest operation on an array is a reference to a bit or a bit range. One bit of an array is referenced as array_name(N) where N is a constant. A range of bits is referenced as array_name(M..N). It is always necessary to specify the bit range when referencing an array.

It is possible to access an array element using variable index:
array_name(V: index1..index2) where V is an integer variable, and index1..index2 are constants that indicate its range. Example:

var source(0..7): boolean; V: 0..7;
define destination := source(V:0..7); -- assuming that the behavior of V is defined elsewhere

Other operations that can be used with any type of arrays are:
:= = != if case next

Examples:

\[
\text{aa}(0..7) := \text{if } \text{bb}(0..2) = \text{cc}(0..2) \text{ then } \text{dd}(0..7) \text{ else } \text{ee}(1..8) \text{ endif};
\]
\[
\text{next(aa}(0..7)) := \text{bb}(2..9);
\]

The rest of the operators can only be applied to Boolean arrays (bit vectors).

**Boolean connectives (bitwise):** & | ^ ! -> <->

Both operands must be of the same width (unless one of them is constant). The result will have the same width as the vector operands.

Example:

\[
\text{v}(0..7) := \text{x}(0..7) & \text{y}(0..7) | !\text{z}(0..7);
\]

**Relational:** = < > <= >=

Both operands must be of the same width (unless one of them is constant). The result will be a scalar Boolean value.

Examples:

\[
\text{c} := \text{v}(0..7) > \text{x}(0..7); \quad \text{d} := \text{v}(0..7) <= 16;
\]

**Arithmetic (unsigned):** + - *

Both operands must be of the same width (unless one of them is constant). The result will have the same width as the vector operands.

Examples:
define \( cc1(0..7) := aa(0..7) + bb(0..7); \)
\[ cc2(0..7) := aa(0..7) + 1; \]
\[ cc3(0..7) := 10 \times aa(0..7); \]

In order not to lose the most significant bits of the result, pad the operands with zeroes on the left.

Examples:
\[ \text{define } aa(0..7) := \text{zeroes}(4) \circledast bb(0..3) \circledast \text{zeroes}(4) \circledast cc(0..3); \]
\[ \text{co\texttt{++sum}(0..7) := 0\texttt{++a}(0..7) \circledast 0\texttt{++b}(0..7);} \]

(+ is the concatenation operator, described below. zeroes(4) is a vector of four zeroes)

Shift: \(\gg\quad \ll\)

The first operand must be a Boolean vector and the second operand must be an integer constant or variable. The result is a Boolean vector of the same width as the first operand. These operations perform the logical shift, (i.e., vacated bit positions are filled with zeroes).

Examples:
\[ \text{define } cc(0..7) := aa(0..7) \ll 2; \]
\[ \text{var } shift\_amount: 0..5; \]
\[ \text{define } dd(0..7) := bb(0..7) \gg shift\_amount; \]
\[ ee(0..8) := 0\texttt{++ff}(0..7) \ll 1; \]

1.3.3 Converting Bit Vectors to Integers and Vice Versa

Bit vector to integer:
\[ \text{bvtoi( a\_vector )} \]
Integer to bit vector:

\texttt{itobv( an\_integer )}

Example:

\texttt{assign next( counter(0..7) ) := itobv( bvtoi( counter(0..7) ) + 1 );}

Constant integers are converted to bit vectors implicitly, you do not need to apply \texttt{itobv}. We recommended that you use bit vectors instead of big integer variables, if possible.

### 1.3.4 Constructing Arbitrary-Lenght Constant Bit Vectors

You can easily construct a bit vector using the following syntax:

\texttt{L#BNNNN...}

where \( L \) is the length of the bit vector (in bits) and \( B \) is the base of the number that \( NNNN... \) stands for - 'b' for binary, 'h' for hexadecimal, 'o' for octal and 'd' for decimal.

Examples:

\begin{verbatim}
define oct_wide(0..63) := 64#o01234567;
define dec_wide(0..63) := 64#d0123456789;
define bin_wide(0..63) := 64#b01010101;
define hex_wide(0..63) := 64#h0123456789ABCDEF;
\end{verbatim}

When \( NNN... \) is too short to fill a vector of length \( L \), as it is in the first three examples, it is left-padded with zeros to fit in.

### 1.3.5 Constructing Bit Vectors from Bits or Sub-vectors

The concatenation operator (++) is used to make bit vectors out of bits or smaller vectors:
expr ++ expr

Example:

    define wide(0..5) := narrow(2..3) ++ bit1 ++ bit2 ++ another_narrow(0..1);

If expr is a constant, it should be either 0 or 1. Wider constant vectors should be split into separate bits.

    define x(0..5) := y(0..2)+1++0++z;  -- allowed
    define x(0..5) := y(0..2)+10B++z;  -- not allowed

The concatenation operator can also appear on the left-hand-side of an assign or define statement. For instance, the following statement:

    define a ++ b ++ c(0..2) := d ++ 1 ++ 0 ++ c(0..1);

is equivalent to the following four statements:

    define a := d;  b := 1;  c(0) := 0;  c(1..2) := c(0..1);

The built-in construct rep() can help construct arrays of repeated elements:

rep (expr, N) is equivalent to expr concatenated with itself N times. For example, you can use the following assignment to make each bit of array 'arr' nondeterministic:

    assign arr(0..3) := rep({0,1},4);  -- {0,1}++{0,1}++{0,1}++{0,1}

Shorthands:

zeros(N) is equivalent to rep(0,N)
ones(N) is equivalent to rep(1,N)
nondets(N) is equivalent to rep({0..1},N)
1.3.6 Array Notes

- **The exact range must be specified in the operation.**
  “a = b” is not equivalent to “a(0..3) = b(0..3)”. b(0..3) represents variables b(0) through b(3) while b represents one variable with no index.

- Operands can take any ranges, provided that their widths are compatible. For example, “a(0..3) & b(1..4)” is legal, but “a(0..3) & b(0..4)” is not.

- If one of the operands is a Boolean vector and the other is a numeric constant, the constant is considered an array of bits. For example, “a(0..1) = 10B” is equivalent to “a(0)=1 & a(1)=0” and “a(1..0) = 10B” is equivalent to “a(1)=1 & a(0)=0”.

- “var v(0..3): { 5, 7, 13 }” defines four state variables, each of them can take the values 5, 7, or 13. This is sometimes confused with “var v(0..3): boolean; assign v(0..3) := { 5, 7, 13 }” that defines a vector of 4 bits, and the whole vector can take the values 5, 7, or 13.

- Arrays can be used as formal parameters of modules and as actual parameters of instances. The actual parameter width must match the width of the formal parameter.

- If you write “#define N 7” and later “a(0..N)”, leave a space around the two dots: a(0 .. N). Otherwise, the standard preprocessor (cpp) used by RuleBase will identify ..N as a token and will not replace N by 7.

1.3.7 More Array Examples

```plaintext
var a(0..3), b(0..8), c(0..2) : Boolean;
define d(0..3) := b(5..8);-- different sub-ranges
define e(0..2) := b(2..0) & c(0..2);-- different directions

var x_state(0..2), y_state(0..2): {s1, s2, s3 };
define same_state := x_state(0..2) = y_state(0..2);

var nda(0..2): boolean;
```

---

PSL/Sugar GDL Flavor User's Guide
Provided by special agreement with IBM
assign nda(0..2) := {001b, 010b, 111b}; -- non-deterministic assign-
ment to a vector

assign next( a(0..2) ) :=
case
    reset : 0;
    a(0..2) = b(0..2) : c(1..3);
    a(0..1) = 10B : d(0..2);
    else : a(0..2);
esac;

var counter(0..7) : boolean;
assign
    init( counter(0..7) ) := 0;
    next( counter(0..7) ) := counter(0..7) + 1;

module and_or ( a(0..7), b(0..7), c(0..7) )( d(0..7) )
{ define d(0..7) := a(0..7) & b(0..7) | c(0..7); }

instance a1 : and_or( x(0..7), y(7..0), z(0..7) )( w(7..0) );

1.4 Sequential Processes

Process constructs of the Modeling Layer are similar to “process statements” of VHDL and to “always blocks” of Verilog. They can be useful in situations when it is awkward to write explicit concurrent definitions for signals. Using process constructs, you can write your code in the form of sequences of state-
ments, which are “executed” in each cycle to compute the needed values of signals. The only statements allowed in a process are variable declarations, variable assignments, IF statements, and CASE statements.
As a simple example,

```plaintext
process {
    var foo: boolean;
    foo := d1;
    if c then foo := d2 endif;
}
```

is equivalent to the concurrent assignment

```plaintext
assign foo := if c then d2 else d1 endif;
```

(Of course, in this example the concurrent form is simpler than the process construct).

As a slightly more realistic example, suppose that we need to model a ripple-carry adder in GDL, but for some reason cannot use the `+` operator:

```plaintext
process {
    var sum(0..7): boolean;
    var carry: boolean;
    carry := 0;
    for i in 7..0 step -1 do
        sum(i) := x(i) ^ y(i) ^ carry;
        carry := (x(i) & y(i)) | (x(i) & carry) | (y(i) & carry);
    end
}
```

The carry signal is assigned several times in the process, and each stanza of the loop refers to the value of carry valid for this specific stanza. If some code outside this process refers to the carry signal, it will refer to the “final” value of carry, which in this case is the overflow bit of the adder.

It is convenient to think about processes as sequential code which is “executed” each cycle, but what happens technically is that RuleBase analyzes the
process construct, keeping track of interim assignments, and generates concurrent definitions for signals driven by the process. This means, for example, that in the wave browser you will only be able to see the “final” values of signals.

If you are familiar with VHDL or Verilog, you will notice that GDL processes are not explicitly associated with some clock signal or a sensitivity list. Instead, they are implicitly clocked on the “system clock”, just like the concurrent “assign next” construct.

The following provides a closer look at the building blocks of a process construct.

- **Variable declarations**
  The process construct should contain `var` declarations for all signals that are assigned within the process. The `var` declaration of each signal should appear before the first assignment to it. Currently there is a restriction on chains of `var` declarations within a process: each `var` declaration should start with `var` keyword, for example:

  “`var foo: boolean; bar: boolean;`”

  is not allowed, but both

  “`var foo, bar: boolean`” and “`var foo: boolean; var bar: boolean`”

  are allowed.

  While using a var outside a process defines a state variable, this is not the case here, unless `assign next` is used (see 2. below).

- **Assignments**
  The three usual forms of RuleBase assignments are supported:

  `assign S := expr;`
  `assign next (S) := expr;`
  `assign init (S) := expr;`

  The keyword `assign` can be omitted. `define` constructs are illegal within a process. `S` is a signal or a concatenation of signals.

  The assignment of the first form:
S := expr;
is similar to the VHDL variable assignment and to the blocking Verilog assignment, in that references to S, which are “executed” after this assignment, will already refer to the new value of S. Therefore, the order of the statements is important. For example,

    foo := 0;
    bar := foo;
    foo := 1;

will assign 0 to bar (even though foo is re-assigned later on).

The assignment of the form:

    next (S) := expr;

behaves more like the VHDL signal assignment and to non-blocking Verilog assignment, in that it doesn’t influence the values of S that can be observed in this cycle.

The use of `next` makes S a state variable.

    next (foo) := 0;
    bar := foo;

will assign the current-cycle value of foo, which is not necessarily 0, to bar. The next-cycle value of foo will be 0 (in the absence of further assignments to `next (foo)` in the process).

The assignment of the form:

    init (S) := expr;

is very special in that it will only be “executed” in the very first cycle, and will have no effect on subsequent cycles.

- **CASE statements**

```vhdl
    case
      guard_1: stat_1;
      guard_2: stat_2;
      ...
      guard_n: stat_n;
```
else: stat_i;
esac;

Each guard, is a Boolean expression. The else clause is optional. Each stat_i
is either a single assignment, or an arbitrary sequence of statements
enclosed in braces.

• IF statements
The IF statement is less general than the CASE statement and can take one
of two forms:
  if condition then
  statements
endif;
or
  if condition then
  statements
else
  statements
endif;

The following is an example of a process construct that makes use of differ-
ent statements:

```plaintext
module server (start,grant)(request,done)
{
  process {
    var state: { idle, wait, busy };  
    init(state) := idle;  
    next(state) := state; -- default behavior
  }
```

IBM Haifa Research Laboratory, Israel
Provided by special agreement with IBM
var request, done: boolean; -- state machine outputs
request := false; done := false; -- their default behavior

case
state=idle & start:
   next(state) := wait;

state=wait: {
   request := true;
   if grant then
      next(state) := busy
   endif;
}

state=busy: {
   done := {true,false};
   if done then
      next(state) := busy
   endif;
}

esac;
} -- process

} -- module
1.5 Environment Constraints

Trans, fairness, assume, restrict and hints are environment constructs that enable you to set constraints on signals. They allow you to describe the environment by declarative means instead of giving each signal a functional behavior. These environment constraints can be combined with other environment constructs such as var, assign, define, etc.

1.5.1 Trans

The trans statement enables you to specify a set of legal transitions between states of the design, thus ignoring all other transitions, which are illegal. Rule-Base will force the model to hold the boolean expression specified within the trans statement at every cycle.

The syntax of the trans construct is as follows:

```
trans <expr>
```

-- Where <expr> is a boolean expression

<expr> can include both environment and design signals.

The statement trans <expr> implies that all transitions which do not comply with <expr> are cut off.

Note that <expr> should not be just any boolean expression, but a boolean expression that describes transitions between states of the design. If <expr> does not contain any next() expression, then it does not refer to any transition, therefore nothing will be cut off; in other words, such a trans statement will not have any effect. This implies that <expr> should contain at least one ’next’.

For example:

```vunit transexamp {
```

IBM Haifa Research Laboratory, Israel
Provided by special agreement with IBM
var a,b,c: boolean;
trans next(a) = next(b);
assert G (a | b | c) ;
}

1.5.2 Fairness

A fairness statement is used to describe a fairness constraint. It describes a condition that must be met an infinite number of times. It is an important tool in specifying abstract environment models. The fairness statement has the following format:

```
fairness expression;
```

The following is a legal fairness statement:

```
fairness state != busy;
```

Currently, the fairness expression cannot contain temporal operators. This limitation protects users from commonly encountered mistakes.

1.5.3 Assume

The assume statement enables you to specify an assumption (invariant) that you want to be true at any cycle. RuleBase will force the model to hold this assumption at every cycle. The assumptions are written as PSL properties.

The syntax of the assume construct is as follows:

```
assume PSL_property ;
```

Most of the PSL safety assertions can be used within assume directives. These PSL assertions (properties) are the same assertions that can be verified on the fly.
Note: there is a special restriction on the use of suffix implication operators within assume directives (see 2.4.7.5: Suffix implication in assume directives).

Examples:

- *read* and *write* are inputs to a design.
- *read* should not be followed by *write* (one or two cycles later).

  This can be specified by:

  ```
  var read, write: boolean;
  assume G (read -> next_a[1..2] (!write)) ;
  ```

Note: G, next_a, next, before and sequences, mentioned in the sequel, are constructs of the PSL language.

Additional requirements:

- The first input command must be a write.
  ```
  assume write before read ;
  ```
- A sequence of three consecutive writes is illegal.
  ```
  assume {[*]; write[*3]}(false) ;
  ```

Assume can help you define complex behavior of inputs.

Using the assume construct, you can start the verification process with an initial free environment, and you can add environment assumptions when you encounter “false negatives” (counter-examples that result from illegal input sequences).

Writing an environment with assumptions enables you to apply compositional verification to your design using the assume-guarantee approach.

The assume-guarantee approach is as follows:
1. Assume that the input signals obey some assumptions.
2. Take those assumptions and guarantee they hold by turning them into vunits and verifying them on the blocks that produced them.

Consider a design that is partitioned into two blocks: block1, block2 (see Figure 3). In the verification of block1, one can write environment assumptions on the input signals generated by block2 using the assume construct. Later, when proceeding to the verification of block2, the already written assumptions can be turned into assertions and verified on block2.

Assumptions cannot be written inside the module or process.

Notes:
- In some cases, the assume construct can cause state space explosion problems by introducing many variables. (These variables are needed to construct a deterministic automata that represents the assume construct.)
such cases, it may be more convenient for the user to use other constraint constructs, such as restrict, or define the behavior in the usual way using define and assign.

- Non-deterministic variables may cause confusing results, if they are used in the same assume statement but at different points of time, for example:

```vmodel
var xx, yy, zz: boolean;
assign next(xx) := xx;
assume G(yy=xx -> next(zz=xx));
```

The user may then expect that the value of yy at any given cycle will be the same as the value of zz at the following cycle. However, the assume statement above allows a behavior in which, for a given cycle, \( xx=0 \) and \( yy=1 \), and on the following cycle \( xx=0 \) and \( zz=0 \).

In order to force a behavior in which the value of yy on a cycle is equal to the value of zz on the following cycle, the following assumptions can be written:

```vmodel
assume G(yy=0 -> next(zz=0));
assume G(yy=1 -> next(zz=1));
```

In general, assumes are most useful in a free environment.

- There is an additional case of false negatives, as seen by users when the counter example does not show the restricted behavior, but is final (i.e., would necessarily show such behavior if prolonged). Such counter examples can be eliminated by writing the same assume on the “causing variables”. The following provides a simplified example of eliminating such a counter example by writing the same assume on the “causing variables”:

```vmodel
vmode default {
  var xx, yy: boolean;
}
```
assign init(xx) := \{0,1\};
assign next(xx) := xx;
assign init(yy) := 0;
assign next(yy) := \!xx;
}

vunit dummy_vunit {
  assume G(!yy) ;
  assert G (yy) ;
}
The user gets the counter example of length 1 where xx=0 and yy=0. This trace only has the next state in which yy=1 and xx=0 is forbidden according to the assumption. Thus, the trace the user gets in this example is not real. By tracing back to the forbidden behavior of yy, one can see that it is forbidden for xx to be 0. By replacing the existing assume with

  assume G(xx) ;

on the same assertion, the user gets the real counter example of length 1 where yy=0 and xx=1. (In this case, real means that it can be prolonged to any length.)

1.5.4 Restrict

The restrict environment construct is used to limit the state space exploration to certain paths. The restrict looks like a regular expression, and its semantics resemble the semantics of a regular expression. Only paths that match the regular expression will be checked.

The syntax of the restrict construct is as follows:

  restrict \{regular_expression\}

- Where the regular expression events can be any of the sequence events.

Example:
restrict { !read[*]; read; !read[*] }

- Restrict RuleBase to check only paths with at most one read command.

Restrict does not have any meaning when it starts with a ["], since every computation path is a prefix of such a restrict; hence, this restrict will not force a limitation on the model.

There are several motivations for the use of restrict, including:

1. A ‘guide/direct’ search to start with specific behavior.
   Example: Every path should start with two reads followed by a write
   restrict {read[*2]; write; ["]}
   
   Note: The ["] at the end of the above restrict is necessary; if you omit it, RuleBase will only check paths with the length of three.

2. An easy way to define an input that behaves according to a specific pattern.
   Example:
   Bus is defined as follows:
   ```
   var bus: {idle, BOP, data, EOP};
   ```
   We restrict the bus behavior to the following pattern:
   ```
   restrict { {bus=idle[*]; bus=BOP; bus=data[*4], bus=EOP}[*] }
   ```
   That is, there can be any number of transactions, with any number of idle cycles in between, in which each transaction starts with BOP, followed by four cycles of data and terminated with EOP.

3. A quick way to verify that a specific design failure does not exist in the ‘design after fix’:
   Given an assertion that failed and a trace that shows a counter example for the assertion, we fix the design and would like to verify (quickly) that we will not get the same failure that we had before. We describe the counter example in the restrict statement, and by that we force RuleBase to check the same path that have failed in the previous run.
   If the counter example no longer exists, you will receive a vacuous result for the new run. This run is quick since it restricts the search space to a very specific pattern of inputs.

IBM Haifa Research Laboratory, Israel
Provided by special agreement with IBM
1.5.5 Hints

The `hint` statement enables you to specify a list of boolean invariants that will guide RuleBase while it is exploring the reachable states space. RuleBase uses each hint in the list to restrict the search of the state space by switching to the next hint in the list when no additional states can be explored using the current one.

**Hints** statement is not a part of the PSL 2.0 language, but it is a Rulebase feature. Therefore, it should be written in smart comments (as it was explained in “Smart Comments” on page 23).

**Syntax:**

```plaintext
--+ hint expr, ..., hint expr[NUM], ...;
```

**Semantics:**

- `hint expr` – continues to search reachable states with the transition relation constrained by `expr` until the fixpoint is reached.
- `hint expr[NUM]` – only performs `NUM` of steps with the constraint.
- RuleBase automatically adds `hint TRUE` at the end of the list, so you do not need to do it.
- When a hint ends (either the fixpoint is reached or a given number of cycles has passed), RuleBase passes on to the next one.

In the case of a liveness property with hints, every liveness property is checked on the fly at every fixpoint that is reached that has a hint.

**Examples:**

```plaintext
var cmd: {read, write, flush, stall };

--+ hint cmd = read, hint cmd = write, hint cmd != flush[5];
```

Hints may be combined with liveness and counters as follows:
liveness + hints
RuleBase does not execute liveness algorithms that use hints in the calculated fixed point as some may expect (CAV99 - Ravi & Somensi). Instead, it performs the following algorithm, which is related to liveness on the fly for all hints (according to their order):

1. Compute the reachable states using the hint (exactly like in on-the-fly mode).
2. Before changing to the next hint, build the transition relation, simplified according to the reachable states (like in liveness on-the-fly).
3. Evaluate the assertion with the classic algorithm on this partial model.
4. If the assertion fails, generate a counter example; otherwise, move to the next hint.

This is exactly like liveness on the fly, which is executed after every hint fixed point instead of after every ’n’ iterations.

counters mode + hints
RuleBase turns the counters mode off during the on the fly model checking. It executes the liveness + hints algorithm described above with the following change: Every time the transition relation is built, the counters are added to the model. When RuleBase continues reachability with hints, the counters are removed again.

1.5.6 Additional Environment Constraint Examples

- cmd, busy are design inputs. busy is active one cycle after cmd:
  var cmd, busy : boolean;
  assume G (cmd -> X busy) ;

- When sending a command, cmd should be active for three cycles, and then inactive for at least two cycles.
  var cmd : boolean;
  assume {[*]; !cmd ; cmd[*3]}(next_a[1..2] (cmd))
  assume {[*]; cmd[*3]}(next_a[1..2](!cmd))

- The above environment written with restrict:
  var cmd : boolean;
  restrict { ![cmd[*]; cmd[*3]; !cmd[*2]!]}{[*]}

IBM Haifa Research Laboratory, Israel
Provided by special agreement with IBM
• Consider a design block that should work properly under the following assumptions:
  • \textit{start}, \textit{end} (the input signals) are pulses.
  • \textit{start} and \textit{end} are interleaving (i.e., there is always a \textit{start} between two \textit{ends} and vice versa).
  • The first \textit{end} will be proceeded by a \textit{start}.

  \begin{verbatim}
  var start, end: boolean;
  assume G (start \rightarrow (X end before_start));
  assume G (end \rightarrow (X start before_end));
  assume G !(start \& end);
  assume start before_end;
  \end{verbatim}

• The above environment written with restrict:

  \begin{verbatim}
  var start, end : boolean;
  restrict | {(!(start \& !end)[*]; start \& !end; !(start \& !end)[*]; !(start \& end)[*]; !start \& end)[*] };
  \end{verbatim}

1.6 Linking the Environment to the Design

In RuleBase, the name connects (links) the design and environment. Thus, to give behavior to an input signal of the name “reset” in your design, give a signal, of the same name, behavior in your environment, using either the \texttt{define} statement (see “Define Statement” on page 15), or the \texttt{var} statement (see “Var Statement” on page 13) in combination with the \texttt{assign} statement (see “Assign Statement” on page 14).

It is important to make sure that you use the name of the signal exactly as recognized by RuleBase (including capitalization). A list of the design signals that RuleBase recognizes can be found under the “Debugging/Signals before reduction” menu option.

1.7 Overriding Design Behavior

The environment can be used to override the behavior of part of the design. To override the behavior of an internal design signal, give it behavior using the
define statement, or the var statement in combination with the assign statement, which specifies **override** as follows:

```plaintext
define override sig := ...
```

or:

```plaintext
var override sig: boolean;
assign init(sig) := ...
next(sig) := ...
```

Overriding design behavior is especially useful if you have implemented a specific behavior of a signal, but want to make sure the design works for any behavior of the signal. For instance, suppose that we have a signal called “predict” that implements a complicated predict function. Some other piece of logic uses the “predict” signal in its calculations. Suppose our assertion is the following:

```
G (predict -> next[2] !low_priority_request);
```

Also suppose that this assertion should be true regardless of the implementation of the predict function. We can make the job of RuleBase easier by eliminating all of the logic driving “predict”, and overriding it with a totally non-deterministic behavior, as follows:

```plaintext
var override predict: boolean;
```

Now, predict can have any behavior.

When overriding design signals, it is important to make sure that you are using the name of the signal exactly as recognized by RuleBase (including capitalization). A list of the design signals that RuleBase recognizes can be found under the “Debugging/Signals before reduction” menu option.
1.7.1 Overriding Initial Values

Sometimes, it is necessary to override the initial value of a flip-flop in the design without modifying its next-state function. In these cases, specify the initial value as follows:

```vhdl
assign init(abc) := 1;
assign init(def) := {0,1};
```

The first statement above assigns an initial value of 1 to signal abc. The second statement assigns a non-deterministic initial value to signal def. In other words, the value of signal def at power-on is not known.

1.7.2 Using Original Design Behavior

When design behavior is overridden with an `override` statement, it is sometimes necessary to use the original behavior of the design in addition to the overriding one. In such cases, the original design behavior can be accessed by specifying `original`, as follows.

For example, suppose ack is a signal in the design. If you write the following:

```vhdl
vmode check_override {
  var override ack:boolean ;
  assign ack := 0 ;
}

vunit override_original {
  inherit check_override ;
  assert G (original(ack) = 0) ;
}
```

RuleBase will check the assertion according to the original behavior of ack and not according to the behavior that overrides it.
When the original behavior of ack is accessed, an auxiliary signal NET_original_ack is created and it takes the original behavior of ack. However, the behavior of ack shown in the scope will be the overriding behavior, not the original one, and NET_original_ack will not appear in the scope.

**Note for advanced users:** The reason that NET_original_ack is not shown in the scope is that it is filtered out of the SMV log and the scope (like all signals whose names begin with NET_).

**To see NET_original_ack in the scope**

- Insert the following two lines to your relubase.setup file:
  ```
  setenv SMVFLAGS "$SMVFLAGS -no_filter_synopsys"
  setenv rb_dont_filter_names 1
  ```

Or:

- Add the following definition to your envs file:
  ```
  define orig_ack := NET_original_ack
  ```

You will then be able to view orig_ack in the scope.

**1.8 Using Non-determinism and Fairness**

It may not yet be clear to you how an environment is used to describe *every possible* input sequence. This is important if we are to fulfill the promise made that formal verification is equivalent to exhaustive simulation. To achieve this exhaustiveness, we use non-determinism.

This section discusses non-determinism and its uses. It is necessary to understand this subject thoroughly in order to use formal verification. Afterwards, we discuss fairness, a closely related concept. Fairness is a way of limiting non-determinism so that the paths that we filter out the paths that we do not want to consider.
A non-deterministic environment is an environment in which we specify more than one possibility for the behavior of a given variable. When we make a non-deterministic assignment, we are indicating to RuleBase that all possibilities must be considered. Do not confuse a non-deterministic assignment with the X value sometimes used in simulation, or with a don’t care assignment as used in synthesis. A don’t care assignment gives a measure of freedom to the synthesis tool—it indicates that any value chosen by the tool is acceptable. In synthesis, the actual logic will either have one value or the other. A non-deterministic assignment, on the other hand, does not give any freedom. Rather, it forces RuleBase to consider the exact outcome of all possible choices.

This section assumes that the assertions checked are of the form “for all possible execution paths, some property holds true.” If assertions of this type are proven in an abstract system, it will also hold true in every concrete system that implements the abstract system. Experience has proven that most of the assertions used in practice are of this type.

1.8.1 Coding Non-determinism

1.8.1.1 Free Variables

A free variable is any variable that is declared, but not assigned a behavior using an assign statement. For instance, assume the following is part of an environment that models a CPU driving a memory bus:

```plaintext
var command: {read, write, none};
```

Since we have not specified any behavior for the `command` variable, RuleBase must consider all possible sequences of commands.

A non-deterministic choice between values of a variable can also be made by enumerating all possible values. Thus, we could have made the command variable free, as follows:

```plaintext
var command: {read, write, none};  
assign command := {read, write, none};
```
1.8.1.2 Non-deterministic Choice

Many times, we do not want a variable to be completely free, but rather con-
strained in some way while still exhibiting non-deterministic behavior in cer-
tain cases. For this purpose, we can use non-deterministic choice among
expressions. The non-deterministic choice is an expression that indicates a
choice between a number of values. For instance, the following expression:

\{write, none\}

indicates a non-deterministic choice between the values “write” and “none”.
Suppose that our CPU contains a MESI-state cache. Then, it will never issue
the read command unless it is in an invalid state. However, the write command
may be issued in any case. We would then model our CPU as follows:

```
var command: \{read, write, none\};
assign command :=
    case
        mesi_state = invalid: \{read, write, none\};
        else: \{write, none\};
    esac;
```

In this environment we have specified that the command can be any of the
three declared values if the variable mesi_state equals invalid. Otherwise, the
variable command can take on either the value “write” or the value “none”.

Example:

Say we have an arbiter that receives two commands: c1 and c2. If both com-
mands have the value “none”, then the arbiter outputs “none”. If one of the
command is something other than “none”, then that command is chosen. If
both commands are something other than “none”, then the arbiter may choose
either command non-deterministically. We can model this as follows:

```
module an_arbiter (c1, c2) (output_command)
{
```
public class arbiter {
    public arbiter (c1, a1, c2, a2) (output_command, output_address) {
        var choose: {1,2};
        var output_command: {read, write, none};
        var output_address: boolean;

        assign output_command :=
            case
                (c1 = none) & (c2 = none): none;
                (c1 = none): c2;
                (c2 = none): c1;
            else: {c1, c2};
        esac;
    }
}

1.8.1.3 Auxiliary Non-deterministic Variables

The arbiter shown above is rather simplistic. To complicate things, let us assume that command c1 comes with address a1, and command c2 comes with address a2. Then, if we choose command c1, it makes no sense to choose a2. In this case we must choose a1. One way to associate one non-deterministic choice with another is to use an auxiliary non-deterministic variable. The following example illustrates this point.

module another_arbiter (c1, a1, c2, a2) (output_command, output_address) {
    var choose: {1,2};
    var output_command: {read, write, none};
    var output_address: boolean;

    assign output_command :=
        case
            (c1 = none) & (c2 = none): none;
            (c1 = none): c2;
            (c2 = none): c1;
        esac;
    }

FIGURE 2. Another arbiter

PSL/Sugar GDL Flavor User's Guide
Provided by special agreement with IBM
By using the free auxiliary variable “choose”, we have tied the non-deterministic choice between c1 and c2 to that between a1 and a2. Notice that in the case where both c1 and c2 are none, we let the address be free. This is an accurate picture of an arbiter in which the address is undefined in the case that no command is chosen.

### 1.8.2 Using Non-determinism to Create an Abstract Model

Suppose we need to model an arbiter that uses a round-robin or other algorithm in order to ensure that every requestor gets a turn. Now, assume that this arbiter is not part of the model under test, but a piece of logic that we know is correct. Creating an exact model of the arbiter will be time-consuming and error-prone. We would probably spend a good amount of time debugging the model rather than verifying our design under test.

If the properties to be verified only depend on the fact that the arbiter eventually gives every requestor a turn, and not on the specific algorithm used by the arbiter, then we may want to use non-determinism to make our modeling job easier. By using a non-deterministic arbiter, as shown in Section 1.8.1, we ensure that any property we prove will be true in the case that the real arbiter is used. This is because a non-deterministic arbiter models all possible sequences of events wherever the non-deterministic choice appears. Since the real behavior is one of the possible choices, it follows that anything proved for the non-
deterministic arbiter is true for the real arbiter. A model that includes more behavior than the entity being modeled is called an abstract model.

There is one catch, however. Since our non-deterministic arbiter models all possible behaviors, it also models the behavior in which c1 is always chosen whenever a non-deterministic choice is to be made. We need a way to filter out this possibility and the way to do so is through fairness.

### 1.8.3 Fairness

Recall that the fairness statement has the following format:

```
fairness expression;
```

The meaning of the fairness statement is that we are only interested in sequences in which the expression specified will happen infinitely often. That is, we are not interested in input sequences in which, at some point in time, the expression becomes false and stays that way forever.

By making the following two fairness constraints within the arbiter of Figure 2

```
fairness choose = 1;
fairness choose = 2;
```

we indicate to RuleBase that we are only interested in input sequences in which choose takes on the values 1 and 2 infinitely often. That is, we filter out sequences in which, at some point in time, choose gets stuck at either value.

### 1.8.3.1 Danger of Fairness

Fairness is a powerful, but dangerous tool. The danger of fairness is that too many paths may be unintentionally filtered out, some of which may include violations of our properties. Here is an example:

```
module server (start) (ready)
{
```

...
var state : { idle, busy, done }; 

assign 
init (state) := idle; 
next (state) := 
    case 
        state=idle & start : busy; 
        state=busy : { busy, done }; 
        state=done : idle; 
    else : state; 
    esac; 
define ready := state=idle; 
} 

vmode default { 
    instance server : server (START_SIG) (READY_SIG); 
    fairness server/state = done; 
} 

In the above example, we give the variable “state” a non-deterministic behavior while it is busy. We also constrain RuleBase with a \texttt{fairness} statement so that it only checks paths on which the machine does not stay busy forever. However, this is a dangerous formulation of the fairness requirement. Since for each “done” there is one “start”, the paths left in the model also have “start” infinitely often. If some deadlock condition in the verified design prevents start from being asserted, this deadlock will not be detected, because the fairness constraint filtered out paths on which start is not asserted infinitely often.

To overcome the problem in the above example, the \texttt{fairness} statement should be formulated in a way that prevents state from staying busy, while having no side effects:

\texttt{fairness} server/state!= busy;
1.9 Using Counter Files

Counters in the design may induce many iterations during reachability analysis, because only one counter state is reached at each step. If you have big counters and experience this problem, try the following:

If your design is small, first try to run without reachability: Set options/verification/reachability=no and verify-safety-OnTheFly=no. This may solve your problem.

If your design is not small, or if the above solution resulted in BDD explosion, try the “counters trick”:

2. Create file “counters” in the verification directory that contains the names of the counter variables, one at each line (vectors should be split into single bits).
3. Add the following line to rulebase.setup:
   ```
   setenv SMVFLAGS "$SMVFLAGS -counters_file ../counters"
   ```

   **Note:** This will result in approximate reachable state space, so errors cannot be detected on the fly.

If you expect errors in early iterations, we recommend that you clean them first in OnTheFly mode (options/verification/reachability=yes and verify-safety-OnTheFly=yes), and only then use the counters trick.

1.10 Modeling Clocks

To use formal verification properly, it is essential to understand the way RuleBase deals with clocks, and to choose the proper clock scheme. This section assumes that the clock signal is generated externally and drives the verified design through input clock pins.
The most simple case is a design that only has one clock, in which only one level or edge of the clock is used in the design. In this case, the clock input should be held constant at the value ‘1’:

```vhdl
define CLK := 1; -- CLK is the clock input pin (*)
```

RuleBase understands it as the clock being active in every cycle. This works even when some of the flip-flops are gated. The gated flip-flops will work only when the gate is active.

The next scheme has one clock, but both levels (or edges) are used in the design. In this case, we define the clock as having alternate values 0 and 1, as follows:

```vhdl
var CLK: boolean; assign init(CLK) := 0; next(CLK) := !CLK;
(**)
```

Notes:

- If your design uses master-slave latches, then the master latches will change on one level of the clock, and the slave latches on the other. However, if the only use of the master latches is to drive the slaves (i.e., there is no use of the master latch output other than by its slave), then you can still use the simpler clock scheme described above, which will give you better performance. To do this, you must model the master-slave pair as a single edge-triggered flip-flop or latch.
- Although (***) may be used in designs that have one clock with one phase, model-checking of (*) is more efficient.
- When the clock is defined as in (**), assertions should include explicit references to the clock signal. For example, the following assertion:
  “G( p -> X q )”
should be rewritten as:
  “G( (p&CLK) -> next_event(CLK)(q) )”
This rewriting may also be necessary in the more complicated clock schemes described below. If all signals in the assertion refer to the same clock, as in the examples above, RuleBase can rewrite the assertion automatically. To do that, write

\[(G( p \rightarrow X q )) @ CLK;\]

See Section 2.4.7: Additional Suffix Implication operators for more details.

Before continuing further with more clock schemes, it is important to note that complex schemes usually contribute to size problems more than simpler ones. When planning the micro-architecture of the design, it is advised to partition the design in a way that each part will have the simplest scheme possible, preferably one clock.

The next scheme has multiple synchronized clocks. For example, assume that there are two clocks, with a 1:3 ratio between their frequencies. In this case, we fix the faster clock at value ‘1’ (always active), and generate a pulse every third cycle for the slow clock:

\[
\text{define } FAST\_CLOCK := 1;
\]

\[
\text{var } \text{clock\_counter: } 0..2;
\]

\[
\text{assign } \text{next}(\text{clock\_counter}) := (\text{clock\_counter} +1) \mod 3;
\]

\[
\text{define } SLOW\_CLOCK := \text{clock\_counter} = 0;
\]

In contrast to clocks in real systems, whose duty cycle is usually 50%, slow clocks in RuleBase should only be active for one cycle each time. (If this is a problem because the clock is generated internally, contact us.)

A similar case is a ratio of 2:3:

\[
\text{var } \text{clock\_counter: } 0..5;
\]

\[
\text{assign } \text{next}(\text{clock\_counter}) := (\text{clock\_counter} +1) \mod 6;
\]

\[
\text{define } SLOW\_CLOCK := \text{clock\_counter in } \{ 0, 3 \};
\]

\[
\text{define } FAST\_CLOCK := \text{clock\_counter in } \{ 0, 2, 4 \};
\]
If the clocks are not synchronized, some tricks are necessary in order to work in a synchronous framework. One case is presented to demonstrate the range of possibilities. In general, even when the clocks are not synchronized, the ratio of frequencies is kept within a limited range. Assume, for example, that the ratio can range from 1:2 to 1:3, which means (among other things) that sometimes the faster clock beats twice (and possibly three times) before the slower clock beats once. One possible solution is to model a slow clock that non-deterministically generates a pulse once every two or three cycles:

```plaintext
define FAST_CLOCK := 1;
var clock_counter: 0..2;
assign next(clock_counter) :=
  case
    clock_counter=0  : 1;
    clock_counter=1  : {2,0};
    clock_counter=2  : 0;
  esac;
define SLOW_CLOCK := clock_counter = 0;
```

Even if the clock scheme in your design is a complex one, we recommend that you begin verification with the simplest scheme possible. You are likely to detect some of the design errors regardless of the scheme.

Only after the simplified design seems to be error free, should you move to a more complex and realistic scheme and hunt for the problems that otherwise cannot be detected.

### 1.11 Modeling Reset

Another important signal that appears in most of the designs is the reset signal. Usually, reset is activated for some time after power-up, and then deactivated for normal operation. Reset must be active long enough to initialize all memory elements with the correct values. In many designs, a few cycles (1 to 10)
are enough. The following example shows an environment model that generates a 4-cycle active-high reset:

```gdl
var reset_counter : 0..4;
assign
    init(reset_counter) := 0;
    next(reset_counter) := if reset_counter=4 then 4 else reset_counter+1 endif;
define RESET := reset_counter != 4;
```

It is important to identify the optimal duration of reset. It should be long enough for correct operation, but not too long. A big counter may contribute to the size problem inherent to formal verification and may result in unnecessarily long counter examples.
CHAPTER 2  
*PSL/Sugar – The RuleBase Specification Language*

### 2.1 Overview

The PSL language is the specification language of RuleBase. It is used in order to formally describe properties to which the design under verification must adhere. PSL is an extension of the temporal logic LTL (Linear Temporal Logic). LTL is designed with academic orientation, and needs some adjustments in order to be used in industry. Particularly, complex LTL specifications are difficult to read and write. PSL adds, on top of LTL, a set of new operators that simplify formulation of complex properties and add significant expressive power.

The following sections describe both LTL and PSL. Section 2.2, Semantic Model, provides background on the underlying model on which LTL and PSL operate (it is not necessary for the understanding of the rest of the chapter and you can skip it if you like). Section 2.3, LTL Operators and Section 2.4, PSL Operators describe the LTL and PSL operators. Section 2.6 describes the OBE (Optional Branching Extension) segment of PSL. The remaining sections offer some practical advice. Before you start to write assertions, we recommend that
you read CHAPTER 4: Managing VUnits, VModes, VProps and Environments.

2.2 Semantic Model

RuleBase is used in order to verify that a given finite-state machine satisfies a given list of properties. The machine consists of a design, usually written in a hardware description language, composed with a GDL (General Description Language) description of the target environment in which the design is expected to run. There are cases, such as in protocol verification, where both the design and the environment are written in GDL. The finite state machine has no free inputs; every input of the design is driven by some signal of the environment, and every input of the environment is driven by the design. While there are no free inputs, the machine usually has multiple choices when moving to the next state, because some of the state-variables, mainly those that model the environment, have non-deterministic behavior.

A non-deterministic finite state machine can be unfolded into an infinite tree that represents the machine’s computations. The tree root represents the initial state of the machine, each tree node corresponds to a state in the machine, and the edges that emanate from a state are the possible transitions to other states. The infinite paths of the tree, beginning at the root, are the machine’s computations. A machine with multiple initial states is unfolded into multiple trees. In the unfolded tree, different nodes may correspond to the same state of the machine. Figure 3 shows an example of a machine and its computation tree.
Each path in the tree represents one possible computation, and RuleBase assertions are interpreted over such paths.

Within RuleBase, rules are written in the specification language PSL. PSL is built on top of LTL (Linear Temporal Logic). LTL, and hence PSL, are specially designed to work with the computation paths described in the previous paragraphs. In the temporal logic LTL, time is discrete, and the world consists of a current state, mapped to a specific node in the computation tree, and of the future beginning at this state (the remainder of the computation path beginning at this state). LTL has no way of referring to the past. The only way to reason about the past is to have information stored in state variables.

An important premise in LTL is that time is infinite. A computation is an infinite sequence of points in time, which start at the current state. Thus, from any point in time (any “current state”), there is an infinite computation continuing into the future. In Figure 4, the beginning of one path (recall that all paths are infinite) is shown in bold. In this figure, and in later figures, the only reason that some points do not show a future is lack of space. **Every point in time has**
**a future.** To simplify the figures, state names are sometimes omitted from tree nodes.

![Diagram](image)

**FIGURE 4.** Beginning of one possible path

An LTL assertion reasons about a single computation path. For example, an LTL assertion may state that a request is always followed immediately by an acknowledge. When we write such an assertion, we imagine that we have selected a single (infinite) computation path, and that the assertion refers to this path. If we say "an acknowledge will be received in the next state", there is only one "next state" in the computation path -- the state that immediately follows the current state. However, as we can see in the computation tree, there are actually many computation paths on which we would like to verify our assertion. In LTL model checking, each assertion is checked on all paths in the computation tree. In other words, we can imagine each LTL assertion as having the words "for all paths" written implicitly at the beginning of the assertion.

### 2.3 LTL Operators

LTL assertions have the following syntax:
1. Signal names and constants are LTL assertions.

2. LTL assertions combined with boolean operators are also LTL assertions:
   !f1,  f1 & f2,  f1 | f2,  f1 -> f2,  f1 <-> f2,  f1 xor f2,

3. If f, f1, and f2 are LTL assertions, then the following are also LTL assertions:
   X f,  G f,  F f,  [f1 U f2]
   These four operators are called **temporal operators**.

Boolean operators have their usual meaning.

Temporal operators are used for reasoning about events that take place along some time interval. **G** means “the assertion holds from now on”, **F** means “the assertion holds now or will hold in the future”, **X** means “the assertion will hold in the next point of time”, and “f1 U f2” means “f2 holds now or sometime in the future and f1 will hold until (but not necessarily including) f2 holds”.

The temporal operator letters and their meanings are:
- **G** = Globally
- **F** = Future
- **X** = neXt
- **U** = Until

The following sections detail the four temporal operators.

Temporal operators take precedence over Boolean operators. Therefore, you should use parentheses to enclose the assertion to which the temporal operator is applied.
2.3.1 The G Operator

The $G$ operator means “from now on”. This is depicted in Figure 5 below. The points in time affected by the operator are marked with a black arrow.

As can be seen by looking at Figure 5, all points in time starting at the current state are marked. Consider an example with two signals, “read” and “write”, which should never be active simultaneously. This fact can be stated in LTL as follows:

$$G \neg (\text{read} \land \text{write}) \quad (\text{Prop. 1})$$

Because the Boolean assertion “$(\neg \text{read} \land \text{write})$” is prefixed by $G$, it will be checked at every point in time starting at the current state.
2.3.2 The F Operator

The F operator means “now or at some future point in time”. This is depicted in Figure 6 below.

![Diagram of the F Operator]

By examining Figure 6, we can see that starting at the current state, at least one future point is marked. For example, say that at the current state, a request has been made and it requires an acknowledge. The acknowledge may take place at different points in time, depending on the circumstances, but it must always eventually take place. This can be expressed in LTL as:

\[ F \text{ack} \quad \text{(Prop. 2)} \]

The above assertion is not very useful, since in real life a request is made at many points in time and under many circumstances. In real life, our world would probably look more like this:
FIGURE 7. The F operator in the real world

In Figure 7, a request is made at three different points in time. Starting at each point where a request is made, there is an infinite path continuing into the future. For each one of these paths, at least one future point is marked with an acknowledge. This can be expressed in LTL as:

\[ G (\text{req} \rightarrow F \text{ack}) \]  

where “\(\rightarrow\)” is the “implies” operator. Thus, this assertion can be read as: at every point in time, if there is a request, then at some future time, we must receive an acknowledge. In simpler terms: whenever there is a request, eventually there is an acknowledge.

There are still some open questions regarding Property 3. Why is \(G\) required in Property 3? Why not simply state:

\[ \text{req} \rightarrow F \text{ack} \]  

(Prop. 4)
The answer is that Property 4 refers only to the initial state. For a hardware model, the initial state is located at power on. Thus, Property 4 refers only to a request that occurs at power on. In order to express events that take place after power on, we must always enclose the assertion in one of the four basic temporal operators \((G, F, X, U)\). Specifically, in order to express a request that can happen at any time, we must enclose Property 4 in the temporal operator \(G\).

### 2.3.3 The X Operator

The \(X\) operator means “at the next point in time”. This is depicted in Figure 8 below.

![Figure 8: The X Operator](image)

In Figure 8, along the path that starts at the current state, the very next point in time is marked. For example, if a request is made at the current state, and an acknowledge is required at the very next time step. This is expressed as:

\[
X \text{ack} \quad \text{(Prop. 5)}
\]
As is the case with the $F$ operator described above, Property 5 is not practical, since in real life a request is made at many points in time and under many circumstances. In real life our world would probably look more like Figure 9.

![Diagram of the X operator in the real world](image)

**Figure 9. The X operator in the real world**

In Figure 9, a request is made at three different points in time. Starting at each point where a request is made, there is an infinite path continuing into the future. For each one of those paths, the very next point in time is marked. This can be expressed in LTL as:

$$G (req \rightarrow X \text{ ack})$$  

(Prop. 6)

Property 6 can be read as follows: for every request, we must get an acknowledgement at the next point in time.

It is worthwhile to compare Figure 7 with Figure 9. In the former, a request must be acknowledged eventually. In the latter, a request must be acknowledged at the very next point in time.
2.3.4 The U Operator

The U operator has two operands, and is used as follows:

\[ [q \ U \ r] \]  \hspace{1cm} (Prop. 7)

which reads: q is true until r is true. Note that:

- r must occur eventually.
- r can occur in the current state, in which case q may not appear at all.
- q need not hold at the time r holds.

This is depicted in Figure 10 below.

By examining Figure 10, we can see that from the current state, all points are marked by q until a point where r holds is reached. For example, suppose that we want to ensure that a busy signal is asserted from the moment a request is made until the time that an acknowledge is received. This is expressed in LTL as:

\[ G (\text{req} \rightarrow [\text{busy} \ U \ \text{ack}]) \]  \hspace{1cm} (Prop. 8)
In this case, Figure 10 represents a sub-path of the complete computation path, with a request that occurs at the current state.

The $U$ operator requires that the terminating condition eventually happen. That is, there are two ways Property 8 can fail. First, if the busy signal is inactive somewhere between req and ack, and second, if the ack never occurs. Because it makes a demand on its terminating conditions, $U$ is known as a strong operator.

At this point, the four basic LTL operators $G$, $F$, $X$, and $U$ have been covered. While combinations of the basic LTL temporal operators presented here can provide much expressive power, complex LTL assertions are difficult to read and write. To overcome this limitation, RuleBase provides higher-level operators that add more expressive ability.

### 2.4 PSL Operators

PSL adds several operators on top of LTL in order to answer real needs that arise in practical formal verification. Although many PSL assertions can be expressed in pure LTL, many other assertions are practically impossible to express in LTL because they would be too complex. PSL is also stronger than pure LTL in the theoretical aspect -- first, in its ability to express any regular expression, as described in Section 2.4.6., and second, in its inclusion of the OBE (Optional Branching Extension), as described in Section 2.6.

Experience indicates that the strong version of the LTL until operator (the $U$ operator) is not suitable for the formulation of many properties. Expressing a weak until (in which there is no demand that the terminating condition must eventually occur) in LTL is laborious and error prone. PSL provides the weak until operator, and in addition, provides both weak and strong versions of some higher-level operators (next_event, before, etc.). A strong operator name has ‘!’ as its last character (e.g., before!).

The following sections describe PSL operators, beginning with the simpler ones.
2.4.1 Basic Temporal Operators

2.4.1.1 always

One of the simplest temporal properties states that some boolean expression holds at all times. For instance, the following property expresses the fact that signals ena and enb should always be mutually exclusive:

always !(ena & enb)  \hspace{1cm} \text{(Prop. 9)}

The use of the temporal operator always indicates that the boolean expression !(ena & enb) must hold at every cycle. It is important not to forget the always operator. Leaving it out gives the property

!(ena & enb)  \hspace{1cm} \text{(Prop. 10)}

which states that ena and enb are mutually exclusive at the initial cycle only. Think of it this way: PSL properties reason about sequences of cycles. Every PSL property starts reasoning at the beginning of the sequence, and progresses from there only by means of a temporal operator. Thus, Property 10 reasons about only the initial cycle, because it has no temporal operator to "push it forward". In Property 9, the use of the temporal operator always causes the boolean expression !(ena & enb) to be applied to all cycles in the sequence, rather than just the initial cycle.

2.4.1.2 never

The never operator allows us to specify conditions that should never hold. Thus, to state that ena and enb are mutually exclusive, we could have used the following

never (ena & enb)  \hspace{1cm} \text{(Prop. 11)}

instead of Property 9.
The operand of a never operator must be a boolean expression or a sequence (see Section 2.4.8).

2.4.1.3 next

The always and never operators allow us to specify a condition that holds or does not hold at all cycles. Other operators allow us to be more specific about the timing. The operator next takes us forward one clock cycle. Thus, the property

\[
\text{always } (\text{req} \rightarrow \text{next ack})
\]

says that whenever signal req is asserted, signal ack is asserted at the next cycle. Property 12 uses the implication operator \( \rightarrow \) which can be read as "if-then". Thus, \((\text{req} \rightarrow \text{next ack})\) can be read as "if req is asserted, then at the next cycle ack is asserted". By enclosing this within the always operator in Property 12 we get "at every cycle, if req is asserted, then at the next cycle ack is asserted", or simply "whenever signal req is asserted, signal ack is asserted at the next cycle".

2.4.1.4 eventually!

The next operator moves us forward exactly one cycle. The eventually! operator allows us to move forward without specifying exactly when to stop. The property

\[
\text{always } (\text{req} \rightarrow \text{eventually! ack})
\]

states that whenever signal req is asserted, signal ack should be asserted some time in the inclusive future. Anytime will do: either now, or at the next cycle, or the one after that, and so on. The exclamation point (!) which is part of the name of the eventually! operator indicates that it is a strong operator. For more on weak and strong operators, see below.
2.4.2 Until Operators

2.4.2.1 until

The until operator is the weak version of the LTL operator $U$. It is written:

\[ p \text{ until } q \quad \text{(Prop. 14)} \]

and means that for all paths, $p$ is true until $q$ occurs. However, the weak until does not require that $q$ eventually occur (in that case $p$ must be true forever). For example, to express the rule “always, once a transaction starts, there will be no additional transaction starts before the end of the first transaction”, you can use the following PSL assertion:

\[ \text{always (trans_start } \rightarrow \text{ X (!trans_start until trans_end))} \quad \text{(Prop. 15)} \]

Property 15 does not require that every transaction end, only that a new one does not start before the first one ends.

The right-hand-side operand of the until operator must be a boolean expression.

2.4.2.2 until!

The until! operator is a strong version of the until operator. It is equivalent to the LTL operator $U$.

The right-hand-side operand of the until! operator must be a boolean expression.

2.4.2.3 until_

Property 14 requires that $p$ be true until, but not necessarily including, the cycle on which $q$ is true (if there exists such a cycle). The statement

\[ p \text{ until\_ } q \quad \text{(Prop. 16)} \]
means “p until q” and also requires that at the first cycle where q is true (if at all), p is also true.

Both operands of the \texttt{until} operator must be boolean expressions.

\textbf{2.4.2.4 until!}

The \texttt{until!} operator is a strong version of the \texttt{until} operator.

Both operands of the \texttt{until!} operator must boolean expressions.

\textbf{2.4.3 Before Operators}

\textbf{2.4.3.1 before}

The \texttt{before} operator has the format

\[ p \text{ before } q \]  \hspace{1cm} (Prop. 17)

and means that on all paths, the first p must happen strictly before (and not together with) the first q. The \texttt{before} operator is a weak operator, that is, it does not require that p eventually happen.

Both operands of the \texttt{before} operator must be boolean expressions.

\textbf{2.4.3.2 before!}

The \texttt{before!} operator is a strong version of the \texttt{before} operator. Thus, the assertion:

\[ \text{always } (\text{req } \rightarrow (\text{data\_receive before! ack})) \]  \hspace{1cm} (Prop. 18)

requires that after request, \texttt{data\_receive} is asserted strictly before \texttt{ack}, and \texttt{data\_receive} must eventually be asserted.
Both operands of the `before!` operator must be boolean expressions.

### 2.4.3.3 before_ and before!_

(p `before_` q) and (p `before!_` q) are similar to (p `before` q) and (p `before!` q), but require that the first `p` happen before or together with the first `q`.

Both operands of the `before_` and `before!_` operators must be boolean expressions.

### 2.4.4 Next Operators

#### 2.4.4.1 next[i]

The `next[i]` operator is an extension of the `next` operator. The assertion

\[
\text{always (req -> next[3] \text{ack})} \quad \text{(Prop. 19)}
\]

indicates that after an assertion of `req`, we expect to see `ack` asserted on the third cycle.

#### 2.4.4.2 next_a[i..j]

The `next_a[i..j]` operator is used to indicate a subset of all cycles - specifically, the `i`'th through `j`'th cycles - on which we require something to occur. Thus, to specify that after a request (assertion of signal `req`), there should not be another request for the next four cycles, we can code:

\[
\text{always (req -> next_a[1..4] (!req))} \quad \text{(Prop. 20)}
\]

The "a" in `next_a` is intended to hint that we interpret the `next_a[i..j]` operator as a kind of bounded application of the `always` operator - Property 20 expects `!req` to hold for all cycles between the next and the fourth next, inclusive.
2.4.4.3 next_e\[i..j\]

The \textbf{next_e}\[i..j\] operator states that we require something to hold on one of the next \(i\)'th through \(j\)'th cycles. For instance, to specify that after a request (assertion of \textit{req}), we expect to see an acknowledge (assertion of \textit{ack}) no sooner than 3 cycles, but within 20 cycles, we can code:

\[
\text{always (req} \rightarrow \text{next_e}[3..20](\text{ack}))
\]

(Prop. 21)

The "e" in \textbf{next_e} is intended to hint that we interpret the \textbf{next_e}\[i..j\] operator as a kind of bounded application of the \textit{eventually}! operator - Property 21 expects \textit{ack} to hold on at least one of the cycles 3 through 20.

The \textbf{next_e}\[i..j\] is a weak operator - we do not require that there be at least \(j\) cycles. For instance, in Property 21, if there are less than 20 cycles after an assertion of \textit{req}, we do not require that \textit{ack} eventually be asserted.

The operand of the \textbf{next_e} operator must be a boolean expression.

2.4.5 Next\_event Operators

The \textbf{next_event} operators are a conceptual extension of the \textbf{next} operators. While \textbf{next} refers to the next cycle, \textbf{next_event} refers to the next cycle in which some boolean condition holds.

2.4.5.1 next_event(b)(f)

The PSL assertion

\[
\text{always (high_pri_req} \rightarrow \text{next_event(grant})(\text{dst==high_pri}))
\]

(Prop. 22)

is used to express the requirement that whenever a high priority request is received (signal \textit{high_pri_req} is asserted), then the next grant (assertion of signal \textit{grant}) must be to a high priority requester (signal \textit{dst} has the value \textit{high_pri}).
The \texttt{next\_event} operator is a weak operator. Thus, in Property 22, \texttt{grant} is not required to occur.

The first operand of the \texttt{next\_event} operator must be a boolean expression.

\subsection{2.4.5.2 next\_event!(b)(f)}

The PSL assertion

\begin{equation}
\text{always (high\_pri\_req} \rightarrow \text{next\_event!}(\text{grant})(\text{dst==high\_pri})) \quad \text{(Prop. 23)}
\end{equation}

is used to express the requirement that whenever a high priority request is received (signal \texttt{high\_pri\_req} is asserted), then the next grant (assertion of signal \texttt{grant}) must be to a high priority requester (signal \texttt{dst} has the value \texttt{high\_pri}).

The \texttt{next\_event!} operator is a strong operator. Thus, in Property 22, \texttt{grant} is required to occur.

The first operand of the \texttt{next\_event!} operator must be a boolean expression.

\subsection{2.4.5.3 next\_event(b)[i](f)}

Just as we can use \texttt{next[i]} to indicate the \texttt{i}’th next cycle, we can use \texttt{next\_event(b)[i](f)} to indicate the \texttt{i}’th occurrence of \texttt{b}. For instance, in order to express the requirement that every time a request is issued (signal \texttt{req} is asserted), signal \texttt{last\_ready} must be asserted on the fourth assertion of signal \texttt{ready}, we can code:

\begin{equation}
\text{always (req} \rightarrow \text{next\_event(ready)[4]}(\text{last\_ready})) \quad \text{(Prop. 24)}
\end{equation}

The first operand of the \texttt{next\_event[i]} operator must be a boolean expression.

\subsection{2.4.5.4 next\_event\_a(b)[i..j](f)}

The form \texttt{next\_event\_a(b)[i..j](f)} indicates that we expect \texttt{f} to hold on the \texttt{i}’th through \texttt{j}’th occurrences of \texttt{b}. For instance, the following:
next_event_a (b)[2..4](f)  

(Prop. 25)

states that in the second, third, and fourth times that \( f \) occurs, \( b \) occurs as well. The "a" is intended to hint that the meaning of \([i..j]\) in next_event_a is similar to the meaning of \([i..j]\) when used with the next_a[i..j] operator.

The first operand of the next_event_a operator must be a boolean expression.

2.4.5.5 next_event_e(b)[i..j](f)

In order to indicate that we expect something to happen on one of the next \( i \)'th to \( j \)'th occurrences of an event, we can use the next_event_e operator. The form \( \text{next_event_e}(b)[i..j](f) \) indicates that we expect \( f \) to hold on one of the \( i \)'th through \( j \)'th occurrences of \( b \). For instance, assertion

\[
\text{next_event_e}(f)[3..4](b)  

(Prop. 26)
\]

states that in one of the third or fourth times that \( f \) occurs, \( b \) should occur as well. The "e" is intended to hint that the meaning of \([i..j]\) in next_event_e is similar to the meaning of \([i..j]\) when used with the next_e[i..j] operator.

Both operands of the next_event_e operator must be boolean expressions.

2.4.6 Sequences - Sugar Extended Regular Expressions (SEREs)

The sequence is a PSL construct used to describe computation paths on which some assertion must hold. It looks like a regular expression, and its semantics resemble the semantics of regular expressions. The sequence suits the world of hardware design. It can be regarded as a textual representation of a timing diagram, or as a generalized control program for simulation. Its main advantage is the simplicity of writing certain properties that are difficult to formulate using other CTL/LTL and PSL operators.

IBM Haifa Research Laboratory, Israel
Provided by special agreement with IBM
SEREs can be used as building blocks of PSL properties. In order to use a SERE to build a PSL property, we link a SERE with another PSL property or with another SERE. In addition, some of the PSL property operators can be applied directly to SEREs.

### 2.4.6.1 Suffix implication \{ \} ()

This PSL operator provides a way to combine a SERE with a temporal property. This operator has two parts, a list of events \{e1; e2; ...\} (SERE), and a PSL assertion (f).

\{ e1; e2; ...; en \} (f) \hspace{1cm} (\text{Prop. 27})

### 2.4.6.2 Building SEREs

The sequence can be regarded as an if statement, in which the event list is a condition that indicates when to check the assertion. It means "if at some computation path all the events take place in the order they are defined, then the assertion must hold on this path at the last cycle of the last event in the list" (an event may last more than one cycle). A semicolon between two events denotes a move of one cycle forwards (however, if an event takes zero cycles, a semicolon either before it or after it is ignored).

An event can be one of the following:

1. \( p \)
   
   p is a Boolean expression.
   
   The expression \( \text{'}p\text{'} \) holds for one cycle.
   
   Example:
   
   \{read; write; cancel\}

2. \( p[^*] \)
   
   p is a Boolean expression.
   
   The expression \( \text{'}p\text{'} \) holds consecutively for zero or more cycles.
   
   Example:
   
   \{read; write[^*]; cancel\}
3. \( p[+] \)
   \( p \) is a Boolean expression.
The expression \( 'p' \) holds consecutively for one or more cycles.
Example:
\{'read; write[+]; cancel\}'

4. \( p[*i] \)
   \( p \) is a Boolean expression.
The expression \( 'p' \) holds consecutively for exactly \( i \) cycles.
\( p[*3] \) is equivalent to \{P; P; P\}
Example:
\{'read; write[*3]; cancel\}'

Note: If the boolean expression \( 'p' \) is 'true', then the 'true' can be ommited.
Examples:
   \( \text{true} \) - Skip one cycle.
   '\[*]' - Skips zero or more cycles.
   '\[+]' - Skips one or more cycles.
   '\[*i]' - Skip exactly \( i \) cycles.

5. \( p[*i..j] \)
   \( p \) is a Boolean expression.
The expression \( 'p' \) holds consecutively for at least \( i \) and for at most \( j \) cycles.
Example:
\{'read; write[*3..4]; cancel\}'

6. \( p[*i..] \)
   \( p \) is a Boolean expression.
The expression \( 'p' \) holds consecutively for at least \( i \) cycles.

7. \( p[*.j] \)
   \( p \) is a Boolean expression.
The expression \( 'p' \) holds consecutively for at most \( j \) cycles.

8. \( p[=i] \)
   \( p \) is a Boolean expression.
   \( p \) occurs exactly \( i \) times, not necessarily consecutively.
p[=3] is equivalent to {!P[*]; P; ![P[*]; P; ![P[*]; P; ![P[*]}}

9. \textit{p[=i..j]}
   \begin{itemize}
   \item \textit{p} is a Boolean expression.
   \item \textit{p} occurs at least \textit{i} times and at most \textit{j} times, not necessarily consecutively.
   \item \textit{i} and \textit{j} are natural numbers (i\geq 0, j\geq i).
   \end{itemize}

10. \textit{p[=i..]}
   \begin{itemize}
   \item \textit{p} is a Boolean expression.
   \item \textit{p} occurs at least \textit{i} times, not necessarily consecutively.
   \item \textit{p[=3..]} is equivalent to {![P[*]; P; ![P[*]; P; ![P[*]; P; ![P[*]}}
   \end{itemize}

11. \textit{p[=..j]}
   \begin{itemize}
   \item \textit{p} is a Boolean expression.
   \item \textit{p} occurs at most \textit{j} times, not necessarily consecutively.
   \end{itemize}

12. \textit{p[->i..j]}
   \begin{itemize}
   \item \textit{p} is a Boolean expression.
   \item '[-->i..j]' as well as all other variations of this operator ('[->i], '[->i..], '[->..j])
   \item are a form of "goto" operators.
   \item For instance: 'p[->i]' says to go to the \textit{i}-th assertion of signal 'p' and to stop
   \item there. 'p[->1..2]' is equivalent to {{![P[*]; P} | ![P[*]; P; ![P[*]; P}}
   \end{itemize}

Example
\{read; write[->3..6]; cancel\}

13. \textit{Q[*n]}
   \begin{itemize}
   \item A sub-sequence \textit{Q} followed by '[*n]', where \textit{n} is a positive integer.
   \item The sub-sequence holds \textit{n} consecutive times.
   \end{itemize}

14. \textit{Q[*]}
   \begin{itemize}
   \item A sub-sequence \textit{Q} followed by '[*]'.
   \item The sub-sequence holds zero or more consecutive times.
   \end{itemize}

15. \textit{Q[+]}
   \begin{itemize}
   \item A sub-sequence \textit{Q} followed by '[+]'.
   \item The sub-sequence holds one or more consecutive times.
   \end{itemize}

16. \textit{P | Q}
Two sub-sequences \( P \) and \( Q \) separated by ‘|’. (sequence disjunction).
Either the first sub-sequence holds, or the second sub-sequence holds. For example, the assertion
\[
\text{always } (\{p;\{q;r\} \mid \{s;t;u\}\})
\]
is equivalent to:
\[
(\text{always } (\{p;\{q;r;u\}\})) \& (\text{always } (\{p;\{s;t;u\}\}))
\]

17. \( P \&\& Q \)
Two sub-sequences \( P \) and \( Q \) separated by ‘&&’. (sequence conjunction).
\( P \) and \( Q \) must occur at the same time (start and end at the same cycle).
\( P \) and \( Q \) must be of the same length (same number of cycles).

Examples:
If read arrives before write and both read and write are not cancelled (and get a grant) then read will be serviced before write:
\[
\{[*]; \{\text{read; (!cancel)}[*]; \text{grant_read}\} \&\&
\{\text{true; write; (!cancel)}[*]; \text{grant_write}\}\}
\]
(operate_read before operate_write)

Exactly 3 write events should occur during the sequence:
\[
\{\ldots; \{\text{req; read}[:]; \text{flush; cancel}\} \&\& \{\text{write}[^=3]\} \ldots\ldots\} \ldots\ldots
\]

18. \( P \& Q \)
Two sub-sequences \( P \) and \( Q \) separated by ‘&’. Like the ‘&&’ operator, only this time \( P \) and \( Q \) don’t have to be of the same length.

19. \( P[^i..j] \)
\( P \) is a subsequence
\( i \) and \( j \) are natural numbers and \( i\geq 0, j\geq i, j!=0 \)
\( P \) holds between \( i \) to \( j \) times.

Example:
\[
\{\{\text{read; write}[^{0..3}]; \text{flush}\}[^{1..4}]\}
\]

20. \( P[^i..] \)
\( P \) is a subsequence.
i is a natural number and i>=0
P holds at least i consecutive times.

Example:
\{\text{read}; \text{write}[7..]; \text{flush}[2..]\}

21. **P : Q**
Two sub-sequences separated by ‘:’ (fusion).
The first cycle of Q starts when P reaches its last cycle.

Examples:
\{\text{start}; \text{req}; \text{busy:done}; \text{end}\} (ack)
\{\text{start}; \{\text{read}; \text{busy[*]}\} | \text{write}; \text{flush }\}\} : \{\text{done}; \text{ready}\} } (ack)

is equivalent to:
\{\text{start}; \{\text{read&done}|\text{read;busy[*]; busy&done}\} | \text{write}; \text{flush&done}; \text{ready}\} (ack)

22. **P within Q**
Two sub-sequences P and Q separated by ‘within’.
Q holds at the current cycle, and P starts at or after the cycle in which Q
starts, and completes at or before the cycle in which Q completes.

Example:
\{!\text{data_active[*3]}\} within \{\text{trans_end}; !\text{trans_start[*]}; \text{trans_start}\}
is equivalent to:
\{[*]; !\text{data_active[*3]; [*]}\} && \{\text{trans_end}; !\text{trans_start[*]; trans_start}\}

Examples:
- Additional ways to express \text{always} (waiting -> next ( next_event (done)( next idle )) ) :
  - \{[*]; \text{waiting}; !\text{done[*]}; \text{done}; \text{true}\}(\text{idle})
  - \text{always} \{\text{waiting}; \text{done[->]}\}( \text{next(idle)} )
  - \{[*]; \text{waiting}; \text{done[->]}; \text{true}\}( \text{idle}\)
- The fourth ready after start should be accompanied by result=ok:
  \{[*]; \text{start}; \{ !\text{ready[*]}; \text{ready }\}[*4] \}( \text{result=ok} )
The next example is interesting from a theoretical point of view. It is a PSL assertion that cannot be expressed in bare LTL. It expresses the fact that f is true at every even cycle (0, 2, ...):

\{ {true; true}[*]; true \} (f)

Sequences may be useful for showing interesting paths, even if you don’t intend to find bugs. Suppose that you want to see a scenario in which a cache line is modified, and later becomes exclusive without being invalidated in between. The following sequence claims that this path is impossible, and its counter example will demonstrate such a path (if one exists):

\{ [*]; modified; \!invalid & \!exclusive[*]; exclusive \} (false)

*False* is a assertion that can never be true, so a counter example will be provided if the sequence in braces is possible.

### 2.4.7 Additional Suffix Implication operators

#### 2.4.7.1 Weak suffix implication \{ \} |-> \{ \}

P |->Q (two sub-sequences separated by ‘|->’):
If a path that is compatible with P occurs, it must be followed (starting at the same cycle where P ends) by a path whose prefix is compatible with Q.

**Example:**

req |-> (ack until (ready until (busy until end)))  
is equivalent to:
\{req\} |-> \{ack[*]; ready[*]; busy[*]; end\}

**Note:** within assume directives, there are special restrictions on the use of suffix implication (see Section 2.4.7.5: Suffix implication in assume directives).
2.4.7.2 Weak suffix next implication  \( \{} \Rightarrow \{ \} \)

\( P \Rightarrow Q \) (two sub-sequences separated by ‘\( \Rightarrow \)’):
If some path compatible with \( P \) occurs, then it must be followed (starting one cycle after \( P \) ends) by a path whose prefix is compatible with \( Q \).

Example:

\[
\text{error} \Rightarrow (AX(\text{cancel\_data1}, & (AX(\text{cancel\_data2} & AX(\text{idle until error}))))))
\]

is equivalent to:

\[
\{\text{error}\} \Rightarrow \{\text{cancel\_data1; cancel\_data2; idle[*]; error}\}
\]

Note: within assume directives, there are special restrictions on the use of suffix implication (see Section 2.4.7.5: Suffix implication in assume directives).

2.4.7.3 Strong suffix implication  \( \{} \rightarrow \{ \}! \)

\( P \rightarrow Q! \) (\( P, Q \) are sub-sequences):
If a path that is compatible with \( P \) occurs, then it must be followed (starting at the same cycle where \( P \) ends) by a path that is compatible with \( Q \) and so, reaches \( Q \)’s end (i.e. reaches the last cycle of \( Q \)).

Comments: Strong version of \( P \Rightarrow Q \)

Examples:

\[
\{a;b\} \rightarrow \{c;d[*];e\}! \quad \text{- e must happen}
\]
\[
\{a;b\} \rightarrow \{c;d[*]; e\} \quad \text{- e may not happen (if} \ d \ \text{is ‘forever’) i.e.} \quad a; b\&c;d;d;d;d;d;d; \ldots \quad \text{- is a valid sequence}
\]

\[
\text{req} \rightarrow (\text{ack until}! (\text{ready until}! (\text{busy until}! \text{end})))
\]

is equivalent to:

\[
\{\text{req}\} \rightarrow \{\text{ack}[*]; \text{ready}[*]; \text{busy}[*]; \text{end}!\}
\]

Note: strong suffix implication is not supported in assume directives.
2.4.7.4 Strong suffix next implication \( \{ \} |=> \{ \}! \)

\( P |=> Q! \) (\( P \), \( Q \) are sub-sequences):
The same as \( P |-> Q! \), with the difference that \( Q \) starts one cycle after \( P \) reaches its end.

**Note:** strong suffix next implication is not supported in assume directives.

2.4.7.5 Suffix implication in assume directives

Within assume directives (see Section 1.5.3, Assume), there are special restrictions on the use of suffix implication. In assume directives, suffix implication in the two forms \( P|->Q \) and \( P|=>Q \) is supported. However, the sub-sequence \( Q \) (the right-hand-side sub-sequence) must be a "simple SERE", in one of the following forms:

- a boolean expression
- \( b; S \) (where \( b \) is a boolean expression, and \( S \) is a "simple SERE")
- \( b[^*]; c \) (where \( b, c \) are boolean expressions)
- \( b[^*]; c; S \) (where \( b, c \) are mutually exclusive boolean expressions, and \( S \) is a simple SERE; note that the mutual exclusion of \( b \) and \( c \) must be syntactically explicit, i.e., they must contain contradictory sub-expressions, such as "a" and "!a").

The two forms of strong suffix implication, \( P|->Q! \) and \( P|=>Q! \), are not supported in assume directives.

2.4.8 always, never and eventually! applied to a SERE

We have previously seen the operators **always**, **never** and **eventually!** as applied to boolean expressions or other properties. For example, Property 13 applies the **always** operator to the property \((req |-> eventually! ack)\) and the **eventually!** operator to the boolean expression \(ack\).
The **always**, **never** and **eventually!** operators can also be applied to SEREs. For example, suppose that after a request, we require not merely that `ack` eventually be asserted, as in Property 13, but also that in the cycle following the assertion of `ack`, `abortin` is not asserted. In other words, we might see any number of assertions of `ack` followed by `abortin`, but eventually we expect to see one which is not aborted. This behavior, illustrated in Figure 11, can be expressed in PSL as follows:

\[
\text{always } (\text{req} \rightarrow \text{eventually!} (\text{ack} & \text{next !abortin}))
\]  
(Prop. 28)

**FIGURE 11. Behavior described by Property 29 and Property 30**

However, we can express this more concisely by applying the **eventually!** operator to a SERE, as follows:

\[
\text{always } (\text{req} \rightarrow \text{eventually!} \{\text{ack;}!\text{abortin}\})
\]  
(Prop. 29)

We can also apply **never** to a SERE, so the following property:

\[
\text{never } \{\text{abortin;}\text{abortin}\}
\]  
(Prop. 30)

states that we should never see two consecutive cycles in which `abortin` is asserted.

Finally, for completeness, we can also apply **always** to a SERE. However, this construct is not very useful. For instance, the following:

\[
\text{always } \{\text{p};\text{q}\}
\]  
(Prop. 31)
states that in every cycle, we will see the sequence \( \{p;q\} \). In other words, that on the first cycle, we will see \( p \) followed by \( q \), and on the second cycle, we will see \( p \) followed by \( q \), etc. In short, we see \( p \) on all cycles, and \( q \) on all but the first.

### 2.4.9 The abort operator

The **abort** operator specifies a condition that removes any obligation for a given property to hold. The syntax of **abort** is:

\[
\text{Property abort Boolean}
\]

where **Property** is some PSL property, and **Boolean** is a Boolean expression.

The left operand of the abort operator is the property to be aborted. The right operand of the **abort** operator is the Boolean condition that causes the abort to occur.

An abort property holds in the current cycle of a given path iff:

- the property that is the left operand holds, or
- the series of cycles starting from the current cycle and ending with the cycle in which the Boolean condition that is the right operand holds does not contradict the property that is the left operand.

For example:

Using abort to model an asynchronous interrupt: A request is always followed by an acknowledge, unless a cancellation occurs.

\[
\text{always ((req -> eventually! ack) abort cancel)};
\]
2.5 Forall operator

In the previous sections, we have seen some temporal operators, and also seen how to build a PSL assertion using SEREs. The **forall** operator is a different kind of operator. Rather than have some temporal meaning, it provides an easy shorthand for expressing a group of related properties. For instance, suppose that we want to specify that whenever a read is received (signal `read_start` is asserted), it will eventually complete (signal `read_complete` will be asserted). The catch is that the reads are tagged, so that in order to know whether or not an assertion of `read_complete` belongs to an assertion of `read_start`, we must compare the tags. Suppose that there are eight possible tags, indicated by `tag(2..0)`. We could code eight separate PSL assertions, as follows:

```plaintext
assert always((read_start & tag(2..0) = 0) ->
  eventually! (read_complete & tag(2..0) = 0)) ;
assert always((read_start & tag(2..0) = 1) ->
  eventually! (read_complete & tag(2..0) = 1)) ;
assert always((read_start & tag(2..0) = 2) ->
  eventually! (read_complete & tag(2..0) = 2)) ;
assert always((read_start & tag(2..0) = 3) ->
  eventually! (read_complete & tag(2..0) = 3)) ;
assert always((read_start & tag(2..0) = 4) ->
  eventually! (read_complete & tag(2..0) = 4)) ;
assert always((read_start & tag(2..0) = 5) ->
  eventually! (read_complete & tag(2..0) = 5)) ;
assert always((read_start & tag(2..0) = 6) ->
  eventually! (read_complete & tag(2..0) = 6)) ;
assert always((read_start & tag(2..0) = 7) ->
  eventually! (read_complete & tag(2..0) = 7)) ;
```

(Prop. 32)

Instead of writing out eight very similar properties, we can use the **forall** operator to get a single property as follows:

```plaintext
assert forall i in 0..7: always((read_start & tag(2..0) = i) ->
  eventually! (read_complete & tag(2..0) = i)) ;
```

(Prop. 33)

The syntax of **forall** is:

```plaintext
forall identifier in type : PSL_property
```

PSL/Sugar GDL Flavor User's Guide
Provided by special agreement with IBM
where \textit{identifier} is some identifier, \textit{type} is either a list of constants enclosed in braces, an integer range \((i..j)\), or boolean, and \textit{PSL\_property} is some PSL property.

\texttt{forall} adds extra state variables. In many cases, this will not cause size problems, provided that you have a good BDD order that includes these variables.

\section{The Optional Branching Extension (OBE)}

The PSL operators we have seen until now belong to the PSL Foundation Language (FL), which is based on LTL. PSL FL assertions reason about a single computation path at a time.

The Optional Branching Extension (OBE) of PSL allows us to reason about \textit{computation trees}. It is based on the branching-time logic CTL (Computation Tree Logic). Section 2.6.1 describes the semantic model on which CTL is based, and Section 2.6.2 describes the CTL operators used in the PSL OBE.

\subsection{CTL Semantic Model}

As we have seen in Section 2.2, the computations of a non-deterministic finite state machine can be described as an infinite tree. In CTL, the world consists of a current state, mapped to a specific node in the computation tree, and of many possible futures (all computation paths emanating from this state). A CTL assertion reasons about the entire subtree that begins at the current state.

\subsection{CTL Operators}

CTL assertions have the following syntax:

1. Signal names and constants are CTL assertions.
2. CTL assertions combined with boolean operators are also CTL assertions: 
   \( !f_1, \ f_1 \& f_2, \ f_1 \mid f_2, \ f_1 \rightarrow f_2, \ f_1 \leftrightarrow f_2, \ f_1 \oplus f_2, \)

3. If \( f, f_1, \) and \( f_2 \) are CTL assertions, then the following are also CTL assertions:
   \( AX \ f, \ EX \ f, \ AG \ f, \ EG \ f, \ AF \ f, \ EF \ f, \ A[f_1 U f_2], \ E[f_1 U f_2] \)
   These eight operators are called \textit{temporal operators}.

Boolean operators have their usual meaning.

Temporal operators are used to reason about events that take place along some time interval. Each temporal operator consists of two letters. The first letter is either \( A \) or \( E \), where \( A \) means “the assertion holds in all paths beginning in the current state”, and \( E \) means “the assertion holds in at least one path beginning in the current state”. The second letter is \( G, F, X, \) or \( U \), where \( G \) means “the assertion holds from now on”, \( F \) means “the assertion holds now or will hold in the future”, \( X \) means “the assertion will hold in the next point of time”, and “\( f_1 U f_2 \)” means “\( f_2 \) holds now or \( f_1 \) will hold until (but not necessary including) \( f_2 \) holds”.

The temporal operator letters and their meanings are:

- \( A = \text{All} \)
- \( E = \text{Exists} \)
- \( G = \text{Globally} \)
- \( F = \text{Future} \)
- \( X = \text{neXt} \)
- \( U = \text{Until} \)

The following sections detail the eight temporal operators.

Temporal operators take precedence over Boolean operators. Therefore, you should use parentheses to enclose the assertion to which the temporal operator is applied.
2.6.2.1 AG and EG

By combining the meaning of $A$ with the meaning of $G$, the resulting $\text{AG}$ means “for all paths, from now on”. This is depicted in Figure 5 below. The points in time affected by the operator are marked with a black triangle.

![Figure 12. AG](image)

As can be seen by looking at Figure 12, all points in time on paths that start in the current state are marked. Consider an example with two signals, “read” and “write”, which should never be active simultaneously. This fact can be stated in CTL as follows:

$$\text{AG } \neg(\text{read } \& \text{ write})$$  
(Prop. 34)

Because the Boolean assertion “$\neg(\text{read } \& \text{ write})$” is prefixed by $\text{AG}$, it will be checked at every point in time starting at the current state.

$\text{EG}$, on the other hand, means “for some path, from now on”. This is depicted in Figure 13 below.

---

IBM Haifa Research Laboratory, Israel
Provided by special agreement with IBM
In Figure 13, you can see that all points in time along one infinite path are marked. This illustrates the fact that in order for $EG$ to be satisfied, you need at least one path where every point in time satisfies the demand. For example:

$$EG \ (transaction\_starts \rightarrow \text{read}) \quad \text{(Prop. 35)}$$

states that there is a possible computation (infinite branch) in which all the transactions are reads.

### 2.6.2.2 AF and EF

By combining the meaning of $A$ with the meaning of $F$, we find that $AF$ means “for all paths, now or at some future point in time”. This is depicted in Figure 14 below.
By examining Figure 14, we can see that starting at the current state, along every possible path, at least one future point is marked. For example, say that at the current state, a request has been made and it requires an acknowledge. The acknowledge may take place at different points in time, depending on the circumstances, but it must always eventually take place. This can be expressed in CTL as:

$$\text{AF} \text{ ack}$$  \hspace{1cm} (Prop. 36)

The above assertion is not very useful, since in real life a request is made at many points in time and under many circumstances. In real life, our world would probably look more like this:
In Figure 15, a request is made at three different points in time. Starting at each point where a request is made, there are many infinite paths. For each one of those paths, at least one future point is marked. This can be expressed in CTL as:

\[ \text{AG} (\text{req} \rightarrow \text{AF} \text{ack}) \]

(Prop. 37)

where “\(\rightarrow\)” is the “implies” operator. Thus, this assertions can be read as: for all paths, at every point in time, if there is a request, then for all paths emanating from that point, at some future time, we must receive an acknowledge. In simpler terms: whenever there is a request, eventually there is an acknowledge.

There are still some open questions regarding Property 37. Why is \(\text{AG}\) required in assertion 4? Why not simply state:

\[ \text{req} \rightarrow \text{AF} \text{ack} \]

(Prop. 38)
The answer is that Property 38 refers only to the initial state. For a hardware model, the initial state is located at power on. Thus, Property 38 refers only to a request that occurs at power on. In order to express events that take place after power on, you must always enclose the assertion in one of the eight basic temporal operators (AG, AF, AX, AU, EG, EF, EX, EU). Specifically, in order to express a request that can happen at any time, you must enclose Property 38 in the temporal operator AG.

EF, on the other hand, means “for some path, at some point in time”. This is depicted in Figure 16 below.

![current state](image)

**FIGURE 16. EF**

By examining Figure 16, you can see that there is some point in some future path from the current state which is marked. For example, EF can be used to express that it must always be possible for our state machine to return to state “idle”, as follows:

\[ \text{AG EF (state = idle)} \]

(Prop. 39)
which reads as: for all paths, at all points in time, there is some path in which, at some point in time, the state will be idle. In simpler terms: it is always true that a path exists to idle. Thus, \( \mathbf{EF} \) can be used to express a lack of deadlock.

### 2.6.2.3 AX and EX

**AX** means “for all paths, at the next point in time”. This is depicted in Figure 17 below.

![Current State Diagram](image)

**FIGURE 17. AX**

In Figure 17, along all paths that start in the current state, the very next point in time is marked. For example, if a request is made at the current state, and an acknowledge is required at the very next time step. This is expressed as:

\[
\text{AX} \, \text{ack} \quad \text{(Prop. 40)}
\]

As is the case with **AF** described above, Property 40 is not practical, since in real life a request is made at many points in time and under many circumstances. In real life our world would probably look more like Figure 18.
In Figure 18, a request is made at three different points in time. Starting at each point where a request is made, there are many infinite paths. For each one of those paths, the very next point in time is marked. This can be expressed in CTL as:

\[ \text{AG} (\text{req} \rightarrow \text{AX} \text{ack}) \]

(Prop. 41)

Property 41 can be read as follows: for every request, we must get an acknowledgment at the next point in time.

It is worthwhile to compare Figure 15 with Figure 18. In the former, a request must be acknowledged eventually. In the latter, a request must be acknowledged at the very next point in time.

\[ \text{EX} \]

means “for some path, at the very next point in time”. This situation is depicted in Figure 19 below.
Once again, by studying Figure 19, you can see that for some path from the current state, the very next point in time is marked.

2.6.2.4 AU and EU

AU has two operands, and is used as follows:

\[ A [q U r] \]  \hspace{1cm} \text{(Prop. 42)}

which reads: for all paths, q is true until r is true. Note that:

- r must occur eventually.
- r can occur in the current state, in which case q may not appear at all.
- q need not hold at the time r holds.

This is depicted in Figure 20 below.
By examining Figure 20, you can see that from the current state, all points on all infinite paths are marked until a point where r holds is reached. The marked points are those in which q must be true. For example, suppose that you want to ensure that a busy signal is asserted from the moment a request is made up until the time that an acknowledge is received. This is expressed in CTL as:

\[ \text{AG} (\text{req} \rightarrow A[\text{busy U ack}]) \]  

(Prop. 43)

In this case, Figure 20 represents a subset of the complete time tree, with a request that occurs at the current state.

The AU operator requires that the terminating condition eventually happen. That is, there are two ways Property 43 can fail. First, if the busy signal is inactive somewhere between req and ack, and second, if the ack never occurs. Because it makes a demand on its terminating conditions, AU is known as a strong operator.
EU means “for some path, until”. The computation tree for EU is left as an exercise for the reader.

2.7 Named Properties, Sequences, and Endpoints

2.7.1 Named Properties

A given property may be applicable in more than one part of the design. In such a case, it is convenient to be able to define the property once and refer to the single definition wherever the property applies. Declaration and instantiation of named properties provide this capability.

A property declaration defines a property and gives it a name. The syntax of a property declaration is:

```
property Name := Property ;
```

or

```
property Name (Formal Parameters) := Property ;
```

where Name is an identifier and Property is a PSL property.

For example:

```
property ResultAfter3 := start -> next[3] (result);
```

The name of a declared property must not be the same as the name of any signal in the environment. Named properties with the same name may be declared in different vunits. Priority of declaration is decided in the following order: current vunit > inherited vunit > default vunit.

A property instantiation creates an instance of a named property. An instance of a named property holds at a given evaluation cycle if and only if the named property holds in that evaluation cycle.
For example:

Given the declaration for the property `ResultAfter3` above,

```
always ResultAfter3
```

is equivalent to:

```
always (start -> next[3] (result))
```

A property declaration may also specify a list of formal parameters which can be referenced within the associated property.

For example:

```
property ResAfterN (boolean start; property result; const n; boolean stop) :=
always (start -> next[n] (result | stop));
```

This property could also be declared as follows:

```
property ResAfterN (boolean start, stop; property result; const n) :=
always (start -> next[n] (result | stop));
```

The two declarations have slightly different interfaces (i.e., different formal parameter orders), but they both declare the same property `ResAfterN`. This property describes behavior in which a specified result (a property) occurs n cycles after an enabling condition (parameter start) occurs, or a cancellation may occur (parameter stop).

For example:

Given the first declaration for the property `ResAfterN` above, the instantiations:
ResAfterN (write_req, eventually! ack, 3, cancel)
ResAfterN (read_req, eventually! (ack | retry), 5, (cancel & write_req))

are equivalent to:

always (write_req -> next[3] ((eventually! ack) | cancel))
always (read_req ->
     next[5] ((eventually! (ack | retry)) | (cancel & write_req)))

2.7.2 Named Sequences

A given sequence may describe behavior that can occur in different contexts (i.e., in conjunction with other behavior). In such a case, it is convenient to be able to define the sequence once and refer to the single definition in each context in which the sequence applies. The declaration and instantiation of named sequences provide this capability.

A sequence declaration defines a sequence and gives it a name. The syntax of a sequence declaration is:

sequence Name := Sequence ;

or

sequence Name (Formal Parameters) := Sequence ;

where Name is an identifier and Sequence is a SERE with enclosing braces.

For example:

sequence BusArb := \{ br; (br & !bg)[*0..5]; br & bg \};

The name of a declared sequence must not be the same as the name of any signal in the environment. Named sequences with the same name may be declared
in different vunits. Priority of declaration is decided in the following order: current vunit > inherited vunit > default vunit.

A sequence instantiation creates an instance of a named sequence. An instance of a named sequence describes the same behavior that is described by the sequence.

For example:

Given the declaration for the sequence **BusArb** above,

```
{[*]; start_br } |=> { BusArb }
```

is equivalent to:

```
{[*]; start_br } |=> { br; (br & !bg)[*0..5]; br & bg }
```

A sequence declaration may also specify a list of formal parameters, which can be referenced within the associated sequence.

For example:

```plaintext
sequence GenericlBusArb (boolean req, grant; const n) :=
{ req ; (req & !grant)[*0..n];  req & grant };
```

The named sequence **GenericBusArb** represents a generic bus arbitration sequence involving formal parameters req and grant, as well as a parameter n, which specifies the maximum delay in receiving the grant.

```plaintext
sequence ReadCycle (sequence ba; boolean bb, ar, dr) :=
{ ba; {bb[*]} && {ar[->]; dr[->]}; !bb }
```

The named sequence **ReadCycle** represents a generic read operation involving a bus arbitration sequence and Boolean conditions bb (bus busy), ar (address ready), and dr (data ready).
For each formal parameter of the declared sequence, the sequence instantiation must provide a corresponding actual parameter.

For example:

Given the declarations for the sequences `GenericBusArb` and `ReadCycle` above, the instantiation:

```plaintext
BusArb (breq, back, 3)
```

is equivalent to:

```plaintext
{ breq; (breq & !back)[*0..3]; breq & back }
```

and

```plaintext
ReadCycle({BusArb(breq, back, 5)}, breq, ardy, drdy)
```

is equivalent to:

```plaintext
{ { breq; (breq & !back)[*0..5]; breq & back }; {breq[*]} && {ardy[->]; drdy[->]}; !breq }
```

### 2.7.3 Named Endpoints

An `endpoint` declaration defines a signal that holds when the end of the associated sequence is reached. The syntax of an endpoint declaration is:

```plaintext
endpoint Name := Sequence ;
```

or

```plaintext
endpoint Name (Formal Parameters) := Sequence ;
```

where `Name` is an identifier and `Sequence` is a SERE with enclosing braces.
For example:

    endpoint EndTrans := {req;!ack[*];ack};

**EndTrans** is high at the first cycle after req in which ack is asserted.

You can use an endpoint anywhere a Boolean signal can be used (including in the modeling layer).

The formula:

    always {EndTrans} |=> {ack[*2]};

requires that whenever the first ack after req is asserted, it is followed by two consecutive ack’s.

An endpoint declaration can also specify a list of formal parameters that can be referenced within the associated sequence.

The endpoint declaration:

    endpoint ActiveLowReset (boolean rb; const n) := {!rb[*n..];rb};

represents a generic reset sequence in which the reset signal is asserted (set to 0) for at least n cycles before being released (set to 1).

The name of a declared endpoint must not be the same as the name of any signal in the environment. Named endpoints with the same name may be declared in different vunits. Priority of declaration is decided in the following order: current vunit > inherited vunit > default vunit.

The endpoint declaration:

    endpoint Full(boolean full_cond) := {full_cond; [*]};
defines a generic endpoint that is asserted in all cycles following a cycle in which `full_cond` was asserted. You can instantiate `Full` with a different condition in each context.

The formula:

\[
\text{always } (\text{send} \rightarrow \text{Full}(\text{head} = \text{tail}));
\]

requires that the send command is asserted only when the buffer is full.

The `endpoint` declaration:

\[
\text{endpoint dout_end(const d) := } \{(\text{dout}=d)[*3]\};
\]

denotes a signal that is asserted at the end of a sequence of three cycle in which dout is equal to d.

The following formulas require that if there is a read followed by three consecutive writes, then in each of the three cycles, the data written is equal to the data read.

\[
\text{%for d in 0..7 do}
\text{assert } \{(\text{read} \& \text{di}=d);!\text{write}[*];\text{write}[*3] \rightarrow \{\text{dout_end(d)}\}
\text{%end}
\]

2.7.4 Formal Parameters for Named Constructs

A named construct declaration may also specify a list of formal parameters, which can be referenced within the associated construct.

The formal parameter types that may be used in named properties are const, boolean, property, and sequence.
The formal parameter types that may be used in named sequences are const, boolean and sequence.

The formal parameter types that may be used in named endpoints are const, boolean and sequence.

In the IEEE version additional formal parameters may be used in all types of named constructs: bit, const bit, numeric, const numeric, const boolean, bitvector, const bitvector, hdltype boolean(i..j).

For each formal parameter of the declared property, the property instantiation must provide a corresponding actual parameter, as described in the following list.

- bit, boolean: a boolean expression.
- const bit, const boolean: a statically evaluable boolean expression.
- numeric: an integer expression.
- const, const numeric: a statically evaluable integer expression.
- bitvector: a boolean vector of any width. Reference to specific bits of the vector in the declaration is not allowed.
- const bitvector: a statically evaluable boolean vector of any width.
- hdltype boolean(i..j): a boolean vector of size j-i+1. Reference to specific bits of the vector in the declaration is allowed. Note that referring to bit number i+n in the declaration will be referring to the n\textsuperscript{th} bit of the actual vector.
- property: a PSL property.
- sequence: a PSL sequence.
2.8 Satellites – More Expressiveness

Although PSL increases expressiveness capabilities, there are still properties that cannot be expressed, and others that are too complicated to formulate. *Satellites* may provide solutions in many of these cases. A satellite is a state-machine that records events that occur in the design under verification. assertions can then refer to these past events by accessing the satellite’s internal state. Satellites do not affect the design because information only flows from the design to the satellite (except when fairness is used in certain ways).

For example, assume that a queue of depth k reads data on one side and writes it on the other side. Assume that we want to prove that the queue never contains more than k data items. Formulation of this property in PSL is difficult, but it becomes easy with a satellite. An up/down counter is defined, whose range is 0 to k, and which is incremented on reads and decremented on writes. It is now necessary only to verify that the counter never exceeds k. We can use the same counter to check for an underflow: Its value should never be less than 0.

Some assertions might have become easier if one could talk about past events. Assume that we want to state that “if p occurs, then at that time q should be active since the last occurrence of r”.

We can define the operator `since` as a module:

```vhdl
module since ( e1, e2 )( e1_since_e2 )
{
    var state: boolean;
    assign next(state) :=
        case
            !e1 : 0;
            e1 & e2 : 1;
            else : state;
        esac;
```

PSL/Sugar GDL Flavor User's Guide  
Provided by special agreement with IBM
define e1_since_e2 := (e1 & e2) | (e1 & state);
}

and use it to formulate the required property:

instance i1 : since( q, r ) ( q_since_r );
assert always ( p -> q_since_r );
### 3.1 Overview

In this chapter we present a list of useful assertion patterns. Its main purpose is to help the novice user, but experienced users may also find interesting patterns. We want to emphasize the fact that one does not need to know all of these patterns to perform successful verification work. Most of the assertions in an average project only employ a small set of patterns. However, you may find ideas that will simplify your work.

### 3.2 Basic Properties

- *ok* is always true:
  
  $\mathbf{G} \; \mathbf{o}k$

- *some_requirement* is always true when reset is inactive:
  
  $\mathbf{G} \; ( \neg \text{reset} \rightarrow \text{some\_requirement} )$

*Note:* Many designs begin in an unspecified state, and are being stabilized during reset. Failure of an assertion during reset is not interesting, so we filter this time interval as shown above.
- Variable `state` can never have the value `error`:
  \[G(\text{state} \neq \text{error})\]
- Variables `state1` and `state2` are never in the same state:
  \[G(\text{state1} \neq \text{state2})\]
- Variables `state1` and `state2` are never in state `critical` together:
  \[G(\text{state1} \neq \text{critical} \mid \text{state2} \neq \text{critical})\]
  or
  \[G(!\text{(state1 = critical} \& \text{state2 = critical})\]
- If `busy` is true then `working` is also true:
  \[G(\text{busy} \rightarrow \text{working})\]
- If `almost_done` is true now, `done` will be true in the next cycle:
  \[G(\text{almost_done} \rightarrow X \text{done})\]
- If `hold` becomes active, it remains active for at least one more cycle:
  \[G(\text{rose}(\text{hold}) \rightarrow X \text{hold})\]
  Note: `rose(\text{hold})` is true if `hold` is currently 1 and was 0 in the last cycle.
- `got` should rise 3 cycles after `get` rises:
  ```
  \text{always(rose(get) \rightarrow next[3](rose(got)))}
  ```
- If we are `going_to_abort` now, we `abort` within 0 to 4 cycles:
  ```
  \text{always(going_to_abort \rightarrow next_e[0..4](abort)})
  ```
- If `master1_needs_bus` becomes active, `master2_accesses_bus` should be inactive for at least 3 cycles, beginning from the next cycle:
  ```
  \text{always(master1_needs_bus \rightarrow next_a[1..3](!master2_accesses_bus)})
  ```
- `Counter` is always between 3 and 7:
  \[G(\text{counter} \geq 3 \& \text{counter} \leq 7)\]
  or
  \[G(\text{counter} \in \{3,4,5,6,7\})\]
- `Status` never has the values `warning` or `error` or `fatal`:
  \[G(!\text{(status} \in \{\text{warning, error, fatal}\})\]
  or
  \[G(\text{status} \neq \text{warning} \& \text{status} \neq \text{error} \& \text{status} \neq \text{fatal})\]
- At most one of the signals `x`, `y` or `z` is 1 (mutual exclusion):
  \[G(x+y+z \leq 1)\]
If \( error \) becomes active, it will remain active forever:
\[
G( \text{error} \rightarrow G \text{error} )
\]

3.3 Arrays

- Define a bit vector \( \text{vec} \) of 4 bits that may have at any moment any of the values 3, 8, or 14:
\[
\text{var vec(0..3): boolean; assign vec(0..3) := \{3,8,14\};}
\]

Note: The above is **NOT** equivalent to “\( \text{var vec(0..3): \{3,8,14\};} \)” which declares an array of four enumerated signals, each of them may have one of the values 3, 8, or 14.

- If the head pointer of a queue is equal to the tail pointer, \( \text{queue\_empty} \) must be true:
\[
\text{always( (head(0..3) = tail(0..3)) -> queue\_empty )}
\]

- The bitwise and of vectors \( \text{vec}(0..7) \) and \( \text{mask}(0..7) \) has at least one bit set:
\[
\text{always( (vec(0..7) & mask(0..7)) \neq 0 )}
\]

- Exactly one bit of the bit vector \( v(0..7) \) is 1:
\[
\text{always( (%for ii in 0..7 do v(ii) + %end 0) = 1 )}
\]

The above is expanded to:
\[
\text{always( v(0)+v(1)+v(2)+v(3)+v(4)+v(5)+v(6)+v(7)+0 = 1 )}
\]

3.4 Before

- If a \( \text{request} \) occurs, then an \( \text{ack} \) should occur (strictly) before the next \( \text{request} \):
\[
\text{always( request \rightarrow next(ack before request) )}
\]

Notes:
- The second \( \text{request} \) may not occur, in which case \( \text{ack} \) is not required.
- \( \text{before} \) (without an underscore) means strictly before: \( \text{request} \) will come (if at all) at least one cycle after \( \text{ack} \).
- The \( \text{next} \) means that we expect \( \text{ack} \) to come at least one cycle after \( \text{request} \).
Another way to formulate the above requirement, which allows more explicit specification of boundary conditions:

\[
\{ [*]; request; !ack[*]; request \}( false )
\]

Notes:

- The path begins with any sequence of events. Then a request occurs, and (beginning from the next cycle) ack is inactive for zero or more cycles. Finally, there is another request. The false on the right hand side means that if such a sequence exists then the property should fail.
- A technique we use is:
  Instead of specifying what should happen, specify what should not happen (as a bad sequence of events), and require false to be satisfied at the end of this sequence. Since false is a property that may never be satisfied, existence of the bad sequence in our design will cause RuleBase to produce a counter-example.

### 3.5 Until

- If request is asserted, it will remain active until (inclusive) grant:
  \[
  \sum_{\text{always}}( request \Rightarrow ( request \text{ until } grant ))
  \]

Notes:

- grant may never occur after this request, in which case request must stay active forever.
- until (with an underscore) means that request must also hold at the first cycle where grant holds.
- Another way to formulate the above requirement:
  \[
  \{ [*]; request & !grant; !grant[*]; !request \}( false )
  \]
- If request is asserted, it will remain active until (not inclusive) grant:
  \[
  \sum_{\text{always}}( request \Rightarrow \text{next}( request \text{ until } grant ))
  \]
- Other ways to formulate the above requirement:
  \[
  \{ [*]; request; !grant[*]; !request & !grant \}( false )
  \]
  or
  \[
  \{ [*]; request; !grant[*] \}( request )
  \]
3.6 Forall

- If `data_in(0..7)` has some value during `read`, in the next time that `write` is active `data_out(0..7)` will have the same value:

```
assert forall val(0..7) in boolean:
always( (read & data_in(0..7)=val(0..7)) -> next_event(write)(data_out(0..7)=val(0..7) ) )
```

Notes:

- `forall` is a means for applying a assertion to multiple values at a time. It is equivalent to writing a separate assertion for each value that the `forall` variable can take:

```
always( (read & data_in(0..7)=0) -> next_event(write)(data_out(0..7)=0) )
```

...  
```
always( (read & data_in(0..7)=255) -> next_event(write)(data_out(0..7)=255) )
```

- `forall` has its penalty — an extra state variable (8 bits in the example above) — but this variable does not usually contribute excessively to the size problem, if the BDD order is reasonable.

3.7 Eventuality

- If `request` is asserted, `ack` should be asserted in the future, beginning from the next cycle:

```
G( request -> X F ack )
```

- If `request` rises, `ack` should be asserted at the same cycle or in the future:

```
G( rose(request) -> F ack )
```

- No matter what is the current state, it is always possible to reach a state where `mstate=idle`:

```
G F (mstate=idle)
```

3.8 More Sequences

- If `grant` is active, and there is no `retry` in the next cycle, `busy` must become active two cycles after `grant`:

```
{ [*]; grant; !retry }{ next busy )
```

or

```
{ [*]; grant; !retry; true }{ busy )
```
or
{ [*]; grant, !retry, !busy } (false )
or
G( grant -> X( !retry -> X busy ))

- The fourth data_ready after start should be accompanied by last_data:
  { [*]; start; {!data_ready[*]; data_ready}[4] } (last_data )
- The fourth data_ready after start should be accompanied by last_data, unless there was an abort in the middle:
  { [*]; start & !abort; {!data_ready & !abort[*]; data_ready & !abort}[4] } (last_data )
CHAPTER 4

Managing VUnits, VModes, VProps and Environments

4.1 Overview

There are many possible ways to structure a verification project. The basic elements of all structures are the same: Modeling Layer statements (GDL), assertions, vprops, vmodes, and vunits. However, as the project becomes more complicated, spans a longer period, and more people become involved, it becomes more important to use a standard methodology.

The main contributor to project complexity is Behavioral partitioning. Behavioral partitioning is an effective method to attack the size problem. In this method, the environment is degenerated in various ways to reduce the size of the design to be verified. Assertions should then be run in multiple reduced environments to cover the full environment. Unless managed carefully, these multiple environments may get out of control.

This chapter suggests a methodology of managing multiple vunits, vprops and vmodes. The methodology is a result of our experience in many formal verification projects. Section 4.2, Defining VUnits, VModes and VProps describes the syntax and semantics of vunits, vmodes and vprops. Section 4.3, Using
Vmodes to Limit the Environment shows an example of how to approach the size problem by using modes, and Section 4.4, Verification Project Management suggests how to structure a verification project that has multiple environments.

4.2 Defining VUnits, VModes and VProps

RuleBase is vunit oriented. A vunit is the basic entity that can run. A vunit defines a group of related assertions (PSL assertions) to be verified in one run. It may also re-define parts of the design or environment, thereby overriding the default behavior for the specific run.

The vunit syntax is as follows:

```plaintext
vunit name {
  "optional textual description of the vunit"

  -- optional inherit statements :
  inherit vunit-name, vunit-name, ... ;
  inherit vmode-name, vmode-name, ... ;
  inherit vprop-name, vprop-name, ... ;

  -- optional test_pins statements (written in smart comment) :
  ++ test_pins signal_name, signal_name, ......;

  -- at least one assertion
  assert "optional textual description" PSL-property ;
  assert "optional textual description" PSL-property ;
  ... 

  -- optional Modeling Layer statements (GDL statements) or assertions :
  <GDL statements (var, assign, define, instance, fairness)> 
}
```

PSL/Sugar GDL Flavor User's Guide
Provided by special agreement with IBM
A `vmode/vprop` is a unit that cannot be run by itself, and is used to group and name environments/assertions respectively. It can only be inherited by vunits or by other vmodes/vprops. The rules of inheritance are as follows:

- **vunits** can be inherited only by other **vunits**
- **vmodes** can be inherited only by other **vmodes** or **vunits**
- **vprops** can be inherited only by other **vprops** or **vunits**.

The `test_pins` statement forces RuleBase to keep some signals during the reduction stage, even if they are not needed for verification of the specific vunit. Sometimes these signals are needed to provide a better understanding of counter-examples. The statement

```plaintext
-- test_pins enable, command;
```

forces RuleBase to keep track of signals `enable` and `command`, even if they are not needed for verification. These signals can later be viewed in Scope windows. `test_pins` is not a part of PSL 2.0 language, but it is a RuleBase feature. Therefore it should be written in smart comments (as it was explained in “Smart Comments” on page 23).

The vmode syntax is as follows:

```plaintext
vmode name {
    “optional textual description of the vmode”

    -- optional inherit statements :
    inherit vmode-name, vmode-name, ... ;

    -- optional Modeling Layer statements :
    <GDL statements (var, assign, define, instance, fairness)>
}
```

In other words, vmode syntax differs from vunit syntax in two ways:

- no assertions should appear in vmode
• vmode may inherit only another vmodes (not vunits and not vprops)

The prop syntax is as follows:

```vprop```
```name```
```{
```
“optional textual description of the vprop”

```-- optional inherit statements :```
```inherit```
```vprop-name, vprop-name, ... ;```

```-- optional assertions:
```
```assert```
```“optional textual description” PSL-property ;```
```
```
```
```}
```

In other words, vprop syntax differs from vunit syntax in two ways:

• no Modeling Layer statements should appear in vprop
• vprop may inherit only another vprops (not vunits and not vmodes)

A vunit must contain at least one assertion (not required in vmode or vprop). All the other parts are optional. "Inherit" statements (if any) should be the first statements of each vunit/vmode/vprop. The order of the rest of the statements is unimportant, and all kinds of statements may appear numerous times. We recommend that you fill the textual description of assertions and vunits. This description may help during the analysis of verification results and facilitate maintenance.

If you create a vmode with a unique name "default", then everything you write in that vmode will be implicitly inherited by all the vunits in your verification project. We will often refer vmode default as default environment. The existance of vmode default is optional.
• Note: *vmode default* cannot contain any inherit statements (otherwise it will create circularity).

Vunits and vmodes may include Modeling Layer statements (*var, assign, define, fairness* and *instance*). The behavior assigned to signals by these statements overrides the signals’ behavior in the default environment (default verification mode). A vunit may inherit Modeling Layer statements from other vunits or vmodes, and a vmode may inherit Modeling Layer statements from other vmodes. Inherited statements override the default environment, but are overridden by statements written directly in the body of the vunit/vmode. The exact hierarchy of behavior is as follows:

1. Signal definition in the default environment (default verification mode) overrides the definition in the design (HDL).
2. Inherited signal definition overrides the definition in the default environment.
3. Signal definition in the running vunit overrides inherited signal definition.

### 4.2.1 Assertions

Each assertion in a vunit or a vprop has the following form:

```assert “optional textual description” PSL-property ;```

The keyword "assert" instructs RuleBase to verify that the PSL property holds. The PSL property is written using PSL operators, as described in CHAPTER 2: PSL/Sugar – The RuleBase Specification Language.

The optional textual description is a string containing free text, which is used for documenting the assertion and clarifying its purpose.

### 4.2.2 Cover statements

Vunits and vprops may also contain cover statements, of the form:
cover Sequence;

This statement instructs RuleBase, while it verifies the assertions in the vunit, to notify the user if a behavior that matches the given sequence is encountered.

For example,

\[
\text{cover \{start\_trans;!end\_trans[*];start\_trans \& end\_trans\};}
\]

Instructs RuleBase to check if there is at least one case in which a transaction starts, and then another one starts on the same cycle in which the previous one completed. If such a case is encountered during the verification run, RuleBase will print a notification message.

The sequence is written using SERE operators, as described in Section 2.4.6. Note that cover \{r\} is semantically equivalent to cover \{[*];r\}. That is, there is an implicit [*] starting the sequence.

\textit{Note:} cover statements are only supported by the FormalSim engine.

### 4.3 Using Vmodes to Limit the Environment

One way to approach the size problem is to limit the behavior of the environment. RuleBase uses information from the restricted environment to automatically reduce the size of the model to be verified. To help reductions, some signals in the environment may be set to constant values, or restricted to some other simple behavior. This over-reduction is usually done by using vmodes, rather than in the default environment, as shown in the example below.

Suppose that a design obtains a command and an address from the environment, in addition to other things. The default environment will include the following lines:

\[
\text{vmode default \{ }
\]

\[
\text{..........}
\]
var command: { load, store, add, jmp };
define CMD(0..2) := -- these are the actual command inputs of the design
case
  command=load : 010b;
  command=store : 111b;
  command=add : 011b;
  command=jmp : 100b;
esac;

var CMD_VALID: boolean;

var ADDR(0..15): boolean;
assign next(ADDR(0..15)) :=
  if CMD_VALID then ADDR(0..15) else nondets(16) endif;
  -- ADDR is stable when CMD_VALID is active, and is free to change otherwise

} -- end of default environment

Now, suppose that the design is too large, or verification takes too long, even though you have used all basic methods to cope with size problems. In this case, you may want to perform behavioral partitioning, and define verification modes that restrict the default behavior. Several possibilities of such vmodes are shown below:

vmode load_add {
  "two commands only. CMD(0..1) become constant"
  var command: { load, add };


Now, vunits can run in the restricted environment by inheriting the above vmodes. For example:

```vunit
vunit some_property {
    inherit load_add_eight_addr_bits;
    assert ... ;
}
```

Since over-reduction limits the model checking run to only a subset of the possible input sequences, multiple runs of the same vunit using different environments are sometimes necessary to provide good verification. Managing these multiple environments is described below in Section 4.4.
4.4 Verification Project Management

A well-formed verification project usually consists of the following elements:

- Default environment (default verification mode)
- Verification modes that define restricted environments
- Vprops that group related assertions
- Vunits

Default environment:
The default environment should model the full behavior of the environment. When writing the default environment, we recommend that you “forget” the small details of how you intend to attack the size problem. This does not mean that the environment is written without considering this problem—on the contrary, the environment models should be abstract and small. Specific reductions should only be reflected in vmodes, which are to be written at a later stage.

Vmodes that define restricted environments:
In many cases the default environment does not cause enough reduction of the design to be verified. Behavioral partitioning is one of the methods that may help in these cases. In behavioral partitioning, multiple reduced environments are defined, each of them is represented as a vmode. Then each assertion is run in all these verification modes (See Section 4.3).

Vprops that group related assertions:
The necessity to run each assertion in multiple environments suggests that you keep assertions in separate units (vprops), to be inherited by vunits.

Vunits:
In this methodology, the list of vunits is a matrix of environment modes (vmodes) and vprops, in which each assertion may run in many verification modes.

Example:
-- verification modes
vmode read_only {
    define command := read;
}

vmode write_only {
    define command := write;
}

-- PSL properties
vprop no_starvation {
    assert G F grant1 ;
    assert G F grant2 ;
}

vprop no_collision {
    assert G !(grant1 & grant2) ;
}

-- vunit matrix
vunit read_only_no_starvation { inherit read_only, no_starvation; }

vunit read_only_no_collision { inherit read_only, no_collision; }

vunit write_only_no_starvation { inherit write_only, no_starvation; }

vunit write_only_no_collision { inherit write_only, no_collision; }
Current PSL Support in RuleBase PE

Note that all constructs in the temporal layer are supported within the restrictions of the PSL Simple Subset, as defined in PSL LRM 1.1, section 4.4.4 (except for OBE operators, which are supported without restrictions).

Flavors

RuleBase PE supports two flavors: GDL and Verilog (Verilog flavor is in beta mode).

Boolean Layer

(1) GDL or Verilog expressions (according to the flavor)
(2) The logical operators -> and <->
(3) The union operator
(4) The builtin functions rose(), fell(), prev(b), prev(b,n), next()

Note: The operand of prev() may be a Boolean expression or Boolean vector.
Temporal Layer

(1) FL operators always, G, never, next, X, next[i], next_a, next_e,
    next_event, next_event_a, next_event_e, until, until_. W, before,
    before_
(2) FL strong operators eventually!, F, until!, until!_, U, before!, before!_,
    next_event!
(3) All SERE and sequence operators
(4) Suffix implication: {SERE}|->{SERE}, {SERE}|=>{SERE}, {SERE}(f),
    {SERE}|->{SERE}!, {SERE} |=>{SERE}! either at the top level of a
    property or nested inside always and abort.
(5) Replicated properties using the forall operator
(6) Clocked properties using the @ operator, applied at the top level of a prop-
    erty
(7) The abort operator.
(8) Named properties, sequences, and endpoints, including parameters
(9) All OBE operators

Verification Layer

(1) Verification directives assert, cover, restrict (cover is supported in Rule-
    Base PE with FormalSim).
(2) The assume verification directive, applied to safety properties of the sim-
    ple subset. Note: When using the |-> and |=> operators in assume direc-
    tives, the rhs SERE must be a "simple SERE", in one of the following
    forms:
    (a) A Boolean expression
    (b) b; S (for Boolean b and "simple SERE" S)
    (c) b[*]; c (for Boolean expressions b, c)
(d) $b[*]; c; S$  (for mutually exclusive Boolean expressions $b$, $c$, and "simple SERE" $S$)

(3) Verification units of type $\text{vunit}, \text{vmode}, \text{vprop}$

(4) Verification unit inheritance

(5) Default clock declarations

**Modeling Layer**

(1) In GDL flavor, all of the GDL language is supported

(2) In Verilog flavor, the synthesizable subset of Verilog is supported (IEEE 1364.1-2002 annex A), except for the following constructs: library, configuration, localparam, genvar, PATHPULSES, generated instantiation, attributes

(3) Structures are not supported
Overview

The PSL 1.0-to-2005 converter transforms PSL 1.0 syntax to PSL IEEE-2005 syntax, preserving the semantics (or "meaning") of properties but changing their syntax.

On cursory examination, PSL 1.0 syntax appears nearly identical to PSL IEEE-2005 syntax. However, this similarity hides differences that can cause the same property to be interpreted in different ways under PSL 1.0 and PSL IEEE-2005. Take, for example, the following property:

AX p | AX q

The syntax is valid under both PSL 1.0 and PSL IEEE-2005. However, in PSL 1.0, the logical--or operator has lower precedence than the CTL AX operator, causing the property to be read as:

(AX p) | (AX q)
… while in PSL IEEE-2005 the logical-or operator has higher precedence than AX, causing the property to be read as:

\[ AX \ ( p \ | \ (AX \ q) \ ) \]

In conclusion: although PSL 1.0 properties may have valid PSL IEEE-2005 syntax, this does not guarantee that their reading remains the same. The converter transforms the syntax in a manner that guarantees that the PSL 1.0 reading is preserved under PSL IEEE-2005.

**Preprocessor Directives**

The converter is unable to handle preprocessor directives directly. In order to convert the contents of a preprocessor directive, the directive must first be expanded, but it is then impossible to "un-expand" the directive and return it to its original form. Therefore, the converter processes the input file in two different modes:

- **Without preprocessing**: preprocessor directives are ignored, and copied to the output as-is, with no conversion. E.g., the directive

  \#define MY_PROPERTY never { a ; b ; c}

  is be copied to the output file as-is, even though under PSL IEEE-2005, it should be:

  \#define MY_PROPERTY never { a ; b ; c} !

  The output file is not guaranteed to be semantically identical under PSL IEEE-2005 to the original file under PSL 1.0.

- **With preprocessing**: the C preprocessor and the M42 preprocessor are first used to expand # and % directives, and the PSL1.0-based parser is used on the result. The output file is guaranteed to be semantically identical under
PSL IEEE-2005 to the original file under PSL 1.0. However, it is likely to be less readable, as preprocessor directives will have been expanded and inlined.

**Usage**

psl1to2005 <input file> <output file>

The converter reads <input file>, and creates the following output files:

- `<output file>`: a file containing the contents of <input file>, converted to PSL IEEE-2005 syntax, without preprocessing.

In order for preprocessing to run, the environment variable RBROOT or FOCS_DIR must be set. If neither variable is set, the converter will be unable to run the M42 preprocessor, and will produce only the first output file.

**Conversion**

The converter alters property syntax to account for the following differences between PSL 1.0 and PSL IEEE-2005:

- **Operator precedence.** In PSL IEEE-2005 operator precedence is different from in PSL 1.0.  
  Conversion: Sub-properties are placed inside parentheses, ensuring a unique reading for properties regardless of operator precedence.

- **Sequence syntax and semantics.** In PSL IEEE-2005, sequences are properties. Additionally, some of the sequence operators have changed precedence relative to other operators, leading to sometimes unexpected results.  
  Conversion: Sub-sequences are placed inside curly brackets, ensuring a unique reading for sequences regardless of operator precedence.
Sequences to which the never operator is applied are made strong (by adding a "!").

- **ValueSet syntax.** Under PSL 1.0, the following is a valid declaration:

  ```psl
  var p : 1 .. 10 ;
  ```

  However, under PSL IEEE-2005, the value-set must be surrounded with curly brackets, so the declaration have would be rewritten as:

  ```psl
  var p : { 1 .. 10 } ;
  ```

  **Conversion:** Curly brackets are added to ranges that function as value-sets.

- **Clock expression syntax.** Under PSL 1.0, any Boolean PSL expression can serve as a clock expression; under PSL IEEE-2005, clock expressions must be surrounded with parentheses, unless they are either an identifier or a PSL built-in function call. For example, the following expression:

  ```psl
  p @ 1
  ```

  … is valid under PSL 1.0 syntax, but in PSL IEEE-2005 it would have to be rewritten as:

  ```psl
  p @ ( 1 )
  ```

  **Conversion:** Parentheses are added to all clock expressions except identifiers and built-in function calls.

- **Keyword case-sensitivity.** In PSL 1.0, keywords are case-insensitive, so that, e.g., "NEXT" and "next" are both interpreted as a keyword. In PSL IEEE-2005, however, keywords are case-sensitive: "next" is interpreted as a keyword while "NEXT" is interpreted as an identifier.

  **Conversion:** Keywords are converted to upper- or lower-case, as appropriate. (E.g., "NEXT" is converted to "next", and "af" is converted to "AF".)

- **Smart-comment directives.** Previous versions of PSL allowed the use of smart-comment directives without requiring that they be encapsulated in a smart comment; PSL IEEE-2005 does not allow usage of smart-comment directives outside a smart comment.

  **Conversion:** Smart-comment directives are encapsulated in a smart comment, unless they are inside one already.
• **Identifiers.** Identifiers are left as is. Files after conversion, in which the same identifier is used sometimes in lower case and sometimes in upper case, will not pass parsing in PSL2005 (although they passed parsing in PSL 1.0).
Index

Symbols
%for 19
%if 21
A
A 90, 91, 93
abstraction 51
AF 93
AG 64, 91
and between sequences 81, 82
arrays 23
array operations 24
boolean vectors 23
concatenation 28
defining 23
nondets() 29
ones() 29
rep() 28
zeroes() 29
assert 36, 40, 47, 110, 112, 115, 116, 117
assign 13, 47
assign and define, differences between 15
assume 37, 39
AU 69, 98
AX 67, 96
B
before 74
before! 74
before!_ 74
before_ 74
boolean 47
boolean vectors 23
built-in functions and macros 12
bvtoi() 27
C
case expression 10
case statement 33
clocks 55
comments 18
concatenation 28
concatenation on the left-hand-side 28
Constants 7
constants
  enumerated 7
counter-example 5
D
case expression 10
case statement 33
clocks 55
comments 18
concatenation 28
concatenation on the left-hand-side 28
Constants 7
constants
  enumerated 7
counter-example 5
D
danger of fairness 53
define 14, 47
don’t care 48
E
E 90, 91
EDL (Environment Description Language) 6, 113
EF 67, 93, 95
EG 64, 91, 92
else 10
endif 11
enumerated constants 7
environment 5
Environment constraints 35
Environment models, describing 5
environments, multiple 113
envs 47, 110
esac 10
EU 70, 98, 100
EX 69, 96, 97
exhaustive simulation 48
expressions 7
F
F 63, 65, 66, 90, 91, 93
fairness 37, 48, 52
fairness, dangers of 53
false formula 83
false negative 5
fell() 12
filtering out paths 52, 53
formula 47, 110, 112
formula examples 103
formulas 110, 111, 112
free variable 49
G
G 63, 64, 66, 90, 91
H
Hint 42
I
if expression 10
if statement 34
in 11
init 13
Initially 35
instance 7, 16, 17

func invar 36, 37
    itobv() 27

declare limiting non-determinism 48
linking environment to design 6, 45

declare Macros 18
    mod 9
    mode 47, 111
    module 16
    multiple clocks 56

declare N
    AX 71
    next 13, 14, 15, 17, 25, 27, 30, 32, 33, 34, 35, 36, 39, 40, 45, 53, 55, 56, 57, 58
    next_event 70, 76
    next_event! 76
    non-determinism 11, 48
    non-determinism, limiting 48
    non-deterministic choice 49
    non-deterministic enumerated constants 49
    nondets() 29

declare O
    ones() 29
    operator precedence and associativity 9
    operators 8
        arithmetic 9
        boolean 8
        case expression 10
        if expression 10
        non-deterministic choice 11
        relational 9
    or between sequences 81
    original 46, 47
    override 45, 46, 47
    overriding design behavior 45
    overriding initial values 46

declare P
    Preprocessing 18
    prev 8
    process 30
        assign statement 32
        case statement 33
        example 34
        if statement 34

var statement 32
PSL 70

declare R
    regular expression 78
    relbase.setup 47
    rep() 28
    reserved words 22, 23
    reset 57
    Restrict 41
    rose() 12
    rule 47, 110, 111, 112

declare S
    satellite 100
    scope 47
    scope rules 18
    see also
        endcase 10
    Sequence 78
    sequence 78
    sequence implies sequence 83
    sequential processes 30
    SMV 6, 15, 47
    state variable 12
    statements 7
    strong operator 70, 73, 74, 99

declare T
    temporal operators 63, 90
    test_pins 110
    The 64, 65, 67, 69
    then 11
    trans 36
    true 79

declare U
    U 63, 69, 91
    uninteresting paths, filtering 48
    until 73, 74
    until! 73
    until!_ 74
    until_. 73

declare V
    var 12, 47
    variables 7
    vmode 111, 113, 116
    vprop 111, 116
    vunit 7, 16, 109, 110, 116
W
weak operator 70, 73, 74
whilenot 87, 88
whilenot! 88
within 86
within! 87

X
x 48, 63, 67, 90, 91
X value 48
xor 8

Z
zeroes() 29