

EDL

1.0 Introduction

EDL (Environment Description Language) is the modeling language of the IBM Formal Verification Toolset. Its primary purpose is to describe the environment for formal verification. However, it is also used in conjunction with Sugar to aid specification of a design.

This is not a formal semantics document, but rather an informal description of EDL.

2.0 Language Constructs

2.1 Expressions

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.

An enumerated constant is one of the symbolic values which a variable can take. For instance, if we declare the following:

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

then each of the 5 symbolic values “idle”, “st1”, “st2”, “st3”, and “waiting” are 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
```

Variables are described in Section 2.2.

Arrays are described in Section 3.0.

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
```

Relational operators can be applied only to integer or boolean expressions.

Arithmetic operators:

else : $expr_n$;
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. The **else** part is not required, but if not present, the result is undefined. 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. It 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 4.0). 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**;

2.1.5 Non-deterministic choice

RuleBase uses non-determinism to describe many possible behaviors at once. The non-deterministic constructs of RuleBase have the following format:

$\{ec_1, ec_2, \dots, ec_n\}$ a non-deterministic choice, where c_i is an enumerated constant.

$expr_1$ **union** $expr_2$: the union of choices represented by $expr_i$

$n_1 .. n_2$: another way to express $\{n_1, n_1+1, \dots, n_2\}$, where n_i are integers.

2.1.6 Other expressions

The following are also expressions:

($expr$) a parenthesized expression

$expr$ **in** $\{v_1, v_2, \dots, v_n\}$ shorthand for

$((expr = v_1) | (expr = v_2) | \dots | (expr = v_n))$

2.1.7 Built-in functions

The built-in functions **fell()**, **rose()** and **prev()** have the following functionality:

- **fell**(expr) is true iff expr is 0, and was 1 on the previous cycle
- **rose**(expr) is true iff expr is 1, and was 0 on the previous cycle
- **prev**(expr) is true iff expr was 1 in the previous cycle

2.2 The var statement

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

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

The type can be one of the following:

- **boolean**
- { enum1, enum2, ... }
- number1 .. number2

(Arrays will be described in Section 3.0)

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.

2.3 The assign statement

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

```
assign init(name) := expression;  
assign next(name) := expression;  
assign name := expression;
```

The first statement assigns an initial value to a state variable. The second statement defines the next-state function of a state variable. A state variable assigned with an **assign init** and/or assigned **next** is simply a memory element, or register (flip-flop or latch). The third statement assigns a value to a combinational state variable.

The following are examples of legal **assign** statements:

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

2.4 The 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 second and following consecutive **define** statements.

2.5 The module statement

An environment file can be totally flat, with no hierarchy at all. In this case all statements are considered to be enclosed by one big main module. 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 which 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 *inputs* is a list of formal parameters passed to the module, *outputs* is a list of formal parameters produced by the module, and *statements* is any sequence of **var**, **assign**, **define**, **fairness** and **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 (**var** or **define** or **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 **module** statement:

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

Modules cannot be *declared* inside other modules but they can be *used* (instantiated) by other modules.

2.6 The instance statement

A **module** statement is only a definition - it has no effect until it is instantiated (called). The **instance** statement instantiates a module using the following format:

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

where *instance_name* is the name of the specific instance (one module can be multiply instantiated), *module_name* is the name of the module being instantiated, *inputs* is a list of expressions passed as inputs to this instance and *outputs* is a list of output parameters, actually connecting 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 2.5:

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

2.7 The fairness statement

A **fairness** statement is used to describe a fairness constraint. A **fairness** statement describes a condition that must be met infinitely often. 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 grant;
```

This fairness constraint will filter out paths on which *grant* is asserted a finite number of times, and leave only those paths on which it is asserted an infinite number of times.

Other types of fairness are described in section 8.1.

2.8 Scope rules

Statements inside a module cannot reference variables outside that module (no *global* symbols). External signals and variables needed by the module must be passed as parameters to the instance. A module can assign values to external signals and variables only 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 formulas 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. Overriding is explained in detail in Section 6.0.

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 “*/”. Comment text is ignored by RuleBase. A comment can be inserted 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.

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

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 { <item> , <item> , ... , <item> } do
...
%end
```

-- where `<item>` is either a number, an identifier.

In the first case, the text inside the %for-%end pairs will be replicated $\text{expr2}-\text{expr1}+1$ times (assuming that $\text{expr2} \geq \text{expr1}$). In the second 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 using simply “ii” if “ii” is a separate token in the text. For instance:

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

is equivalent to:

```
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:

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

is equivalent to:

```
define aa0 := 0 > 2;
define aa1 := 1 > 2;
define aa2 := 2 > 2;
define aa3 := 3 > 2;
```

- If “ii” needs to be used as part of an expression, it can be accessed using % {<expr>} as follows:

```
%for ii in 1..4 do
  define aa% {ii-1} := % {ii-1} > 2;
%end
```

is equivalent to:

```
define aa0 := 0 > 2;
define aa1 := 1 > 2;
```

```
define aa2 := 2 > 2;  
define aa3 := 3 > 2;
```

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

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

2.9.2 %if

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**. The syntax of the **%if** construct is as follows:

```
%if <expr> %then  
...  
%end
```

or:

```
%if <expr> %then  
...  
%else  
...  
%end
```

3.0 Arrays

3.1 Defining arrays

An array of state variables is defined as follows:

```
var name ( index1 .. index2 ) : type ;
```

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

Examples:

```
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

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 };
```

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 a variable index as follows:

array_name(V: index1..index2) where *V* is a integer variable, and *index1..index2* are constants indicating 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:

```
:= = !=
```

```
Example: aa(0..7) := if bb(0..2)=cc(0..2) then dd(0..7) else ee(1..8) endif;
```

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

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: `v(0..7) := x(0..7) & y(0..7) | !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: `c := v(0..7) > x(0..7);` `d := 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 * aa(0..7);
```

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

Examples:

```
define aa(0..7) := zeroes(4) ++ bb(0..3) * zeroes(4) ++ cc(0..3);
      co++sum(0..7) := 0++a(0..7) + 0++b(0..7);
```

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

Shift: `>> <<`

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:

```
define cc(0..7) := aa(0..7) << 2;
var shift_amount: 0..5;
define dd(0..7) := bb(0..7) >> shift_amount;
      ee(0..8) := 0++ff(0..7) << 1;
```

3.3 Construction of 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);
```

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 ++ e(0..1);
```

is equivalent to the following four statements:

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

The following built-in functions can help to construct arrays of repeated elements:

rep (expr, N) is equivalent to expr concatenated with itself N times.

Shorthands:

zeroes(N) is equivalent to **rep**(0,N)

ones(N) is equivalent to **rep**(1,N)

nondets(N) is equivalent to **rep**({0..1},N)

3.4 More array examples

```
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 };
```

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

```
assign nda(0..2) := {001b, 010b, 111b};           -- non-deterministic assignment 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) );
```

3.5 Quantification Over Data Values

When specifying the behavior of data, it is often necessary to refer to specific data values. For example, suppose that we want to say that the data which is read in during a *read* operation will be written out in the next *write* operation. One way to do it is to write a formula for each data value:

```
%for i in 0..31 do      -- assuming that the data type is 0..31
  formula { AG( (read & data_in=i) -> next_event(write)(data_out=i) ) }
%end
```

This is equivalent to 32 formulas, and might be inefficient if there are too many values. The above can be done in one formula using the **forall** construct as follows:

```
forall i: 0..31:
  formula { AG( (read & data_in=i) -> next_event(write)( data_out=i) ) }
```

The syntax of **forall** is:

```
forall variable : type :
```

where *variable* is an EDL variable which is defined only for the purpose of quantification. It should not be defined elsewhere. *type* is any legal type, including a bit vector.

More examples:

```
forall i(0..31): boolean:  
  formula { AG( (read & data_in(0..31)=i(0..31)) ->  
              next_event(write)(data_out(0..31)=i(0..31)) ) }
```

```
forall i: 0..15:  
  formula { AG( counter=i -> AX counter=(i+1) mod 16 ) }
```

4.0 Sequential Processes

Process constructs of EDL are similar to “process statements” of VHDL. 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 statements, 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,

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

is equivalent to the concurrent assignment

```
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 for the moment that we need to model a ripple-carry adder in EDL, but for some reason cannot use the “+” operator:

```
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  
}
```

Note that 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 really happens is that RuleBase analyzes the process construct, keeping track of interim assignments, and generates concurrent definitions for signals driven by the process.

Now we shall take a closer look at the building blocks of a process construct.

1. Variable declarations

The process construct should contain **var** declarations for all signals which are assigned within the process. The **var** declaration of each signal should appear before the first assignment to it.

2. Assignments

The three usual forms of RuleBase assignments are supported:

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

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

The assignment of the first form:

```
S := expr;
```

is similar to variable assignment of VHDL and to blocking assignment of Verilog, in that references to S which are “executed” after this assignment will already refer to the new value of S. For example,

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

will assign 0 to bar (even in spite of the fact that foo is re-assigned later on).

The assignment of the form:

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

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

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

bar := foo;

will assign to bar the current-cycle value of foo, which is not necessarily 0. 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 special in that it will be “executed” only in the first cycle, and will have no effect in subsequent cycles.

3. CASE statements

case

guard₁: stat₁;

guard₂: stat₂;

...

guard_n: stat_n;

else: stat_e;

esac;

Each guard_i 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**.

4. IF statements.

The IF can take one of two forms:

if condition **then**

statements;

endif;

or

if condition **then**

statements;

else

statements;

endif;

We will conclude this section with an example of a process construct which makes use of different statements:

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

    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
```

5.0 Environment constraints

Invar, **assume** and **restrict** are environment constructs that enable setting constraints on signals. They let you describe the environment by declarative means instead of giving each signal a functional behavior.

5.1 Invar

The **invar** statement enables you to specify a boolean invariant that you want to be true at any cycle. In other words, it will filter out states in which the invariant does not hold.

The syntax of the invar construct is as follows:

```
invar <expr>  
-- Where <expr> is a boolean expression.
```

The boolean expression within the invar can include both environment and design signals.

Example:

Given a design with the inputs *request1*, *request2*, *request3*, the design should work properly only under the constraint that at most one request can be active at any given cycle.

This can be specified by:

```
var request1, request2, request3: boolean;  
invar (request1 + request2 + request3 <= 1)
```

request1, *request2*, *request3* signals have non-deterministic behavior that obeys the above invariant.

5.2 Assume

Assume can be seen as an extension of the invar construct. It enables you to write more expressive assumptions on your model, telling RuleBase to force your model to hold those assumptions. The assumptions are written as Sugar properties.

The syntax of the assume construct is as follows:

```
assume {safety_sugar_formula}
```

Examples:

- *read* and *write* are inputs to a design.
- *read* should not be followed by a *write* (1 or 2 cycles later).

This can be specified by:

```
var read, write: boolean;  
assume {AG (read -> ABG[1..2] (!write))}
```

Another requirement:

- The first input command must be a write

```
assume {write before _read}
```

5.3 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 regular expression. Only paths that match a prefix of 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 SERE events.
```

Example:

```
restrict { !read[*], read, !read[*] }  
- restrict RuleBase to check only paths with at most one read command.
```

6.0 Linking the environment to the design

In RuleBase, the connection between the design and the environment is by name. Thus, in order to give behavior to an input signal of name “reset” in your design, just give a signal of that name behavior in your environment, using either the **define** statement (see Section 2.4), or the **var** statement (see Section 2.2) in combination with the **assign** statement (see Section 2.3). It is important to make sure that you are using the name of the signal exactly as RuleBase knows it (including capitalization).

7.0 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, simply give it behavior using either the define statement, or the var statement in combination with the assign statement, specifying **override** as follows:

define override sig := ...

or:

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

Overriding design behavior is especially useful if we 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 formula is the following:

```
AG (predict -> AX[2] !low_priority_request)
```

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

```
var override predict: boolean;           -- predict can now have any behavior
```

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:

```
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.

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

8.0 Fairness

Recall that the **fairness** statement has the following format:

fairness expression;

The meaning of the **fairness** statement is that we are interested only 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.

8.1 Additional (advanced) fairness types

The following additional fairness types are supported:

- **FG p ;**
Leaves in the model only paths on which from some point onwards, *p holds forever*.
- **FG->FG p , q ;**
Leaves in the model only paths on which, if there exists a point from which *p holds forever*, then there also exists a point from which *q holds forever*.
- **FG->GF p , q ;**
Leaves in the model only paths on which, if there exists a point from which *p holds forever*, then *q holds infinitely often* .
- **GF->GF p , q ;**
Leaves in the model only paths on which, if *p holds infinitely often*, then *q also holds infinitely often*.