GDL:
General Description Language

1.0   Introduction

GDL (General 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 PSL to aid specification of a design.

This document describes the GDL language.

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). Note that 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 on. 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:

expr - expr   minus

expr + expr   plus

expr * expr   multiplication

expr / expr   division

expr **mod** expr   modulo

Concatenation:

++

2.1.3 Operator precedence

The precedence of operators is as follows:

++   (concatenation)   (highest precedence)

!   (not)

+ -

* / mod

= != < <= > >=

Temporal operators

&   (and)

| (or)

xor ^

<->   (iff)

->   (implies)   (lowest precedence)

2.1.4 Case and If expressions

GDL provides two constructs which 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

   condition$_1$: expr$_1$;

   condition$_2$: expr$_2$;

   ...

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. 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 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

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

{ $expr_1$, $expr_2$, ... , $expr_n$ }   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 $\{n_1, n_1+1,..., n_2\}$, where n_i are integers.

2.1.6 Other expressions

The following are also expressions:

(expr)   a parenthesized expression

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

$$((expr = v_1) \mid (expr = v_2) \mid ... (expr = v_n))$$

2.2   The var statement

A **var** statement declares 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 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 assign 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 the 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   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.

2.8   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. A comment can be inserted anywhere a space is legal, except in text strings.

GDL supports cpp pre-processing directives (e.g., #define, #ifdef, #else, #include, and #undef), and the PSL macros %for and %if.

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 an 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));

4.0   Sequential Processes

Process constructs of GDL 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 GDL, 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 happens technically is that the tool 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 a variable assignment of VHDL and to a 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.

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

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

3. CASE statements

case

    guard_1: stat_1;

    guard_2: stat_2;

    ...

    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
```