The SMV language

K. L. McMillan
Cadence Berkeley Labs
2001 Addison St.
Berkeley, CA 94704
USA
mcmillan@cadence.com

March 23, 1999

Abstract

This document describes the current state of the input language used by the SMV model checker.

Contents

1 SMV language overview

The SMV language can be divided roughly into three parts – the definitional language, the structural language, and the language of expressions. The definitional part of the language declares signals and their relationship to each other. It includes type declarations and assignments. The structural part of the language combines definitional components. It provides language constructs for defining modules and structured data types, and for instantiating them. It also provides constructor loops, for describing regularly structured systems, and a collection of conditional structures that make describing complicated state transition tables easier. Finally, expressions in SMV are very similar to expressions in other languages, both hardware description languages and programming languages. For this reason, expressions will be discussed last, as any expressions appearing in discussions of other parts of the language should be self explanatory.

2 Data types and type declarations

A type declaration is of the form

```
<signal> : <type>;
```

where `<signal>` is the name of a signal and `<type>` is the set of values that the signal may take.

2.1 Boolean, enumerated and subrange types

The simple types are *boolean*, *enumerated* and *subrange*. The type “boolean” is simply an abbreviation for the set {`0,1`}. Thus,

```
foo : boolean;
```

declares a signal named “foo”, which can take on the value 0 or 1. An enumerated type is a set of symbols. For example,

```
bar : {ready,willing,able};
```

declares a signal named "bar", which can take one of the symbolic values "ready", "willing" or "able". A type can also be a subrange of the integers. For example,

```
count : 0..7;
```

declares a signal "count" which can take any value inclusively in the range from 0 to 7. Numeric values in type declarations may also be expressions, consisting of numeric constants, and the numeric operators +, -, *, /, mod, <<, >> and ** (see section 9).

2.2 Arrays

The only remaining type in the language is the array type. [1] A declaration of the form:

```
<signal> : array <x>..<y> of <type>;
```

declares an array of signals of type <type>, with subscripts running from <x> to <y>. For example, the declaration

```
zip : array 2..0 of boolean;
```

is equivalent to declaring.

```
zip[2] : boolean;
zip[1] : boolean;
zip[0] : boolean;
```

There is one additional aspect to the array declaration, however, which relates mainly to binary arithmetic. An array declared as

```
little : array 2..0 of boolean;
```

is "little endian", in the sense that `little[0]` is treated as the least significant bit, and `little[2]` is treated as the most significant bit. By contrast,

```
big : array 0..2 of boolean;
```

is "big endian", in the sense that big[0] is treated as the most significant bit, and big[2] is treated as the least significant bit. This distinction is treated in more detail in the section on vector expressions and binary arithmetic.

An element of an array can be referenced by adding a subscript in square brackets. The subscript must evaluate to an integer in the declared range of the array.

[1] N.B.: Structured types are built using the module construct.

2.3 Multidimensional arrays

Arrays of arrays can also be declared. For example,

```
matrix : array 0..1 of array 2..0 of boolean;
```

is equivalent to

```
matrix[0] : array [2..0] of boolean;
matrix[1] : array [2..0] of boolean;
```

The boolean signals declared in this way are

```
matrix[0][0]        matrix[0][1]        matrix[0][2]
matrix[1][0]        matrix[1][1]        matrix[1][2]
```

There is no fixed limit to the number of dimension of an array declared in this way.

2.4 Generic arrays

Arrays whose elements are of different types can also be declared. This is done by declaraing a *generic array*, as follows:

```
<signal> : array <x>..<y>;
```

where the type of the elements is unspecified. The types of the elements can then be declared separately. For example:

```
state : array 0..2;
state[0] : {ready, willing};
state[1] : {ready, willing, able};
state[2] : {ready, willing, able, exhausted};
```

This differs from simply declaring the elements of the array in that the generic array declaration tells the compiler the upper and lower bounds of the array and whether its storage order is "big endian" or "little endian". This allows array references with variable subscripts and binary arithmetic expressions to be compiled.

2.5 Structs

3 Signals and assignments

A *signal* is an infinite sequence of values of a given type. For example,

```
0;1;0;1;...
```

is a sequence of type boolean (of course, it is alo an integer sequence).

3.1 Operations on signals

An operator is applied to a signal one element at a time. For example, the operator ~ stands for logical "not". Thus if

```
foo =  0;1;0;1;...
```

then

```
~foo =  1;0;1;0;...
```

That is, it is the result of applying logical "not" to each element of the sequence. Similarly, & stands for logical "and". Thus, if

```
                 foo = 0;1;0;1;...
and              bar = 0;0;1;1;...
```

then

```
foo & bar = 0;0;0;1;...
```

3.2 Assignments

An *assignment* is of the form

```
<signal> := <expr>;
```

where `<expr>` is an expression that combines other signals using operators like ~ and &. Unlike an assignent in a typical "procedural" language, this assignment means exactly what it says: that `<signal>` *is equal to* `<expr>`. So for example, suppose we make the assignment

```
zip := foo & bar;
```

If foo and bar are as above, then

```
zip = 0;0;0;1;...
```

Note that this assignment says *what* to compute, but does not say *when* to compute it. Thus, for example, we might first compute the entire sequences foo and bar (or more accurately, finite subsequences), and then compute the sequence zip. Or, we might compute the first elements of all three sequences, then the second elements, and so on. This is a significant departure from hardware description languages such as verilog HDL, which explicitly schedule computations by means of an event queue. In SMV, the issues of what to compute and when to compute it are separated. Thus, the same program might be compiled into a highly parallel hardware implementation, or a completely sequential software implementation.

3.3 Unit delay assignments – the "next" operator

A special operator is provided for describing *recurrences*. Recurrences are circular or recursive systems of equations, and are the way that sequential systems are described in SMV.

If `x` is a signal, then `next(x)` is, intuitively, the "next" value of `x`. More precisely, the ith value of `next(x)` is equal to the $(i+1)$st value of `x`. Thus, for example, if

```
x = 0,1,2,3,...
```

then

```
next(x) = 1,2,3,4,...
```

By assigning a value to the "next" value of a signal, we can define a sequential machine. For example, assuming `x` and `y` are boolean signals,

```
next(x) := y ^ x;
```

defines a signal `x` which "flips" each time the signal `y` is true (the `^` operator stands for "exclusive or"). By definiton, the "next" value of `x` is equal to `(y ^ x)`, which is equal to `x` if `y` is false and `~x` if `y` is true. Note that this "recurrence" places a restriction on the order in which we can compute the values of `x`: we cannot comput the $(i+1)$st value until we have computed the ith value. Since the values of `x` must be computed in sequence, we have defined a sequential machine.

Note, however, that the above assignment does not tell us the initial value of `x`. Thus, we obtain a different sequence depending on whether `x` starts at 0 or 1. We can set this initial value by assigning

```
init(x) := 0;
```

In this case, if we had

```
y = 0;1;0;1;...
```

we would get

```
x = 0;0;1;1;0;0;1;1;...
```

On the other hand, if we assigned

```
init(x) := 1;
```

we would obtain the sequence

```
x = 1;1;0;0;1;1;0;0;...
```

As another example, to declare a signal `x` that maintains a running sum of the values of a signal `y`, we would say

```
init(x) := 0;
next(x) := x + y;
```

3.4 State machines

Here is an example of a small finite state machine, expressed in SMV. It starts in a state "idle" and waits for a signal "start" to be asserted. On the next cycle, it changes to a state "cyc1", then to state "cyc2", then returns to "idle". In state "cyc2", it asserts a signal "done":

```
start,done : boolean;
state : {idle,cyc1,cyc2};

next(state) :=
  switch(state){
    idle: start ? cyc1 : idle;
    cyc1: cyc2;
    cyc2: idle;
  };

done := (state = cyc2);
```

This illustrates two forms of conditional expressions in SMV. The "switch" operator evaluates its argument "state", then chooses the first expression in the curly brackets that is tagged with that value. Thus, if state = cyc1, then the value of the switch expression is cyc2. There is also a simpler form of conditional expression, that appears in the example as

```
start ? cyc1 : idle
```

If "start" is true, this evaluates to "cyc1", else to "idle".

The above state machine can be expressed more "procedurally" using the structural conditional constructs describe in the next section. We would write:

```
default done := 0;
in switch(state){
  idle:
    if start then next(state) := cyc1;
  cyc1:
    next(state) := cyc2;
  cyc2:
    next(state) := cyc2;
    done := 1;
}
```

This style of expression is semantically equivalent to the previous one, but can be much more readable for large complex state machines.

4 Rules for assignments

An SMV program amounts simply to a system of equations, with a set of unkowns that are the declared signals. With an arbitrary set of equations, there is, of course, no guarantee that a solution exists, or that the solution is unique. Examples of systems that have no solutions are

```
x := x + 1;
```

or

```
next(x) := x + 1;
next(x) := x - 1;
```

An example of a system with many solutions is

```
x := y;
y := x;
```

We avoid these difficulties by placing certain rules on the structure of assignments in a program, to guarantee that every program is executable. This means, among other things, that a schedule must exist for computing the elements of all the sequences. The rules for assignments are:

- The single assignment rule – each signal may be assigned only once.
- The circular dependency rule – a program may not have "cycles" in its dependency graph that are not broken by delays.

4.1 The single assignment rule

In addition, SMV follows a "single assignment" rule. This means that a given signal can be assigned only once in a program. Thus, we avoid the problem of conflicting definitions. The definition of "single assignment" is compicated somewhat by the "next" and "init" operators. The rule is this: one may either assign a value to x, or to next(x) and init(x), but not both. Thus, the following are legal:

x := foo;	next(x) := foo;
init(x) := foo;	init(x) := foo; next(x) := bar;

while the following are illegal:

x := foo; x := bar	next(x) := foo; next(x) := bar;
x := foo; init(x) := bar;	x := foo; next(x) := bar;

An important note: assigning an array reference with a variable index counts as asigning every element in the array, as far as the single assignment rule is concerened. Thus, for example,

```
x[0] := foo;
x[count + 1] := bar;
```

is a violation of the single assignment rule, if "count" is not a constant. This is because it cannot be determined at compile time that "count + 1" is not equal to 0. One way to look at this, is that

```
x[i] := foo;
```

(assuming "i" is not a constant) is considered to be exactly equivalent to

```
if (i = 0) x[0] := foo;
if (i = 1) x[1] := foo;
if (i = 2) x[2] := foo;
...
if (i = n) x[n] := foo;
```

(assuming x is declared array 0..n).

If you want to make two assignments to variable indices in the same array, use the "default" construct (described below). Thus

```
default next(x[i]) := foo;
in next(x[j]) := bar;
```

would be legal. In the case where i = j, the second assignment would take precedence.

4.2 The circular dependency rule

If we have the assignment

```
x := y;
```

then we say that x *depends on* y. A *combinational loop* is a cycle of dependencies that is unbroken by delays. For example, the assignments

```
x := y;
y := x;
```

form a combinational loop. Although as equations, they may have a solution, there is no fixed order in which we can compute x and y, since the ith value of x depends on the ith value of y and vice versa.

To be more precise, an assignment of form

```
next(x) := <expr>;
```

introduces "unit delay dependencies". There is a unit delay dependency from `x` to every signal refernced in `<expr>`. An assignment of the form

```
<signal>) := <expr>;
```

introduces "zero delay dependencies", in the same way. A combinational loop is a cycle of dependencies whose total delay is zero. Currently, combinational loops are illegal in SMV.

Therefore, legal SMV programs have the following property: for any sequence values chosen for the unassigned (free) signals, there is at least one solution for the assigned signals. There may be multiple solutions in the case where a signal has an unassigned initial value, or the case of nondeterministic assignments (see below).

There are cases where a combinational loop "makes sense", in that there is always a solution of the equations. In this case, the order in which signals are evaluated may be conditional on the values of some signals. For example, take the following system:

```
x := c ? y : 0;
y := ~c ? x : 1;
```

If c is false, then we may first evaluate x, then y, obtaining x = 0, then y = 0. On the other hand, if c is true, we may first evaluate y, then x, obtaining y = 1, then x = 1. The existence of conditional schedules such as this is difficult to determine, since it may depend on certains states (or signal values) being "unreachable". For example, if we have

```
x := c ? y : 0;
y := d ? x : 1;
```

it may be the case that c and d are never true at the same time, in which case x and y can always be evaluated in some order. Loops of this kind do sometimes occur in hardware designs (especially in buses and ring-structured arbiters). The expected approach to this problem is require the user to provide constraints on the order of evaluation (so that the program can be compiled), and to verify that these constraints always have a solution. Currently, however, combination loops are simply disallowed.

4.3 Range violations and unknown values

When a signal is assigned a value that is not an element of its declared type, the result is that the value of the signal is undefined. Thus, the signal may take on any value in its type. Another way to view this is that any assignment

```
x := expr;
```

is treated as if it were a shorthand for:

```
x := (expr in TYPE) ? expr : TYPE;
```

where `TYPE` is the set of values in the type of signal `x`. This means that if the value of `expr` is not in the set `TYPE`, then the value of `x` is chosen nondeterministically from the set `TYPE`. See the next section for a discussion of nondeterministic choice.

4.4 Order of assignments and declarations

Because assignments are treated as a system of simultaneous equations (or inclusions), the order in which assignments appear in the program is irrelevant. There may be multiple type declarations for a given signal, provided they all agree on the type. Type declarations and assignments may appear in any order. [2]

5 Nondeterministic assignments

Especially in the early stages of a design, a designer may not want to completely specify the value of a given signal. Incomplete specification may represent either a design choice yet to be made, incomplete information about the environment of a system, or a deliberate abstraction made to simplify the verification of a system. For this purpose, SMV provides *nondeterministic* choice. A nondeterministic choice is represented by a set of values. If we make the assignment

```
signal := {a,b,c,d};
```

then the value of `signal` is chosen arbitrarily from the set `{a,b,c,d}`. As another example, suppose that in our previous state machine, we don't want to specify how many cycles will be spent in state "cyc1". In this case, we could write:

```
next(state) :=
  switch(state){
    idle: start ? cyc1 : idle;
    cyc1: {cyc1,cyc2};
    cyc2: idle;
  };
```

Note that in case state = cyc1, the value of the switch expression is the set {cyc1,cyc2}. This means that the next value of "state" may be either "cyc1" or "cyc2". In general, the mathematical meaning of the assignment

```
x := y;
```

where y is a set of values, is that x is *included in* the set y. Ordinary values are treated as sets of size one. Thus, properly speaking, an SMV program is a simultaneous set of inclusions, rather than equations.

[2] The current compiler accepts programs in which some type declarations are omitted. In particular, the type of a signal which is assigned deterministically with zero delay need not be declared. New programs should not rely on this feature, however.

6 Modules

A *module* is a bundle of definitions (type declarations and assignments) that can be reused. Much like a subroutine, a module may have *formal parameters*. When creating an *instance* of the module, actual signals or expressions are plugged in for the formal parameters, thus linking the module instance into the program. Most often the formal parameters of a module are declared to be either inputs or outputs. Inputs are expected to be assigned outside the module, whereas outputs are expected to be assigned inside the module.

6.1 Module declarations

As an example, suppose we want to construct a binary counter, by designing a counter "bit", and then chaining the bits together to form a counter. In SMV, the counter bit might be declared as follows:

```
MODULE counter_bit(carry_in, clear, bit_out, carry_out)
{
  INPUT carry_in, clear : boolean;
  OUTPUT bit_out, carry_out : boolean;

  next(bit_out) := clear ? 0 : (carry_in ^ bit_out);

  carry_out := carry_in & bit_out;
}
```

The "INPUT" and "OUTPUT" declarations are specialized forms of type declarations, which also give the direction of signals being declared. These declarations must occur before any ordinary type declarations or assignments.

6.2 Instantiations

To create a three-bit counter, we can now write, for example:

```
        clear : boolean;
        count : array 2..0 of boolean;
        carry : array 3..0 of boolean;

        bit0 : counter_bit(carry[0], clear, count[0], carry[1]);
        bit1 : counter_bit(carry[1], clear, count[1], carry[2]);
        bit2 : counter_bit(carry[2], clear, count[2], carry[3]);
```

Here, three *instances* of the module "`counter_bit`" are created. These instances have names "bit0", "bit1", "bit2". Each instance is, in effect, a copy of the definitions in module "`counter_bit`". However, all the signal names referenced in the instance are prefixed with the instance name, so that they are unique to that instance. For example, the signals in module instance "bit0" are:

```
bit0.carry_in
bit0.clear
bit0.bit_out
bit0.carry_out
```

6.3 Input and output declarations

The effect of the INPUT declaration is to make an assignment from the actual paramenters to the corresponding formal parameters. Thus, in instance "bit0", the declaration

```
INPUT carry_in, clear : boolean;
```

has the effect of assigning

```
bit0.carry_in := carry[0];
bit0.clear := clear;
```

Similarly, the effect of the OUTPUT declaration is to make an assignment from the formal paramenters to the corresponding actual parameters. Thus, in instance "bit0", the declaration

```
OUTPUT bit_out, carry_out : boolean;
```

has the effect of assigning

```
count[0] := bit0.bit_out;
carry[1] := bit0.carry_out;
```

6.4 Instance hierarchies

Modules may, of course, contain instances of other modules, and so forth, provided the module references are not circular. So we can, for example, create three-bit counter module, as follows:

```
MODULE counter3(carry_in, clear, count, carry_out)
{
  INPUT carry_in, clear : boolean;
  OUTPUT count : array 2..0 of boolean;
  OUTPUT carry_out : boolean;

  carry : array 3..1 of boolean;

  bit0 : counter_bit(carry_in, clear, carry[0], carry[1]);
  bit1 : counter_bit(carry[1], clear, carry[1], carry[2]);
  bit2 : counter_bit(carry[2], clear, carry[2], carry[3]);

  carry_out := carry[3];
}
```

If we then instantiate this module with

```
foo : counter(cin,clr,cnt,cout);
```

we will have, for example, an instance of `counter_bit` called `foo.bit0`, which defines signals

```
foo.bit0.carry_in
foo.bit0.clear
foo.bit0.bit_out
foo.bit0.carry_out
```

MODULE declarations may not appear inside other MODULE declarations, however. That is, all MODULE declarations must be in the outermost scope.

6.5 Structured data types

A module with only type declarations and no parameters or assignments acts like a structured data type. For example, to define a data structure "hands" with fields "left" and "right", the following module might be defined:

```
MODULE hands()
{
  left, right : boolean;
}
```

An instance of this structured type can be created as follows:

```
party : hands();
```

This is exactly equivalent to

```
party.left, party.right : boolean;
```

The two fields of this record can be referenced as

```
party.left
party.right
```

In fact, any signal belonging to a module instance can be referenced directly by name in this way. Normally, however, it is recommended that only inputs and outputs be referenced.

An array of a given structured type may be created in the same manner as an array of signals. For example,

```
foo : array 1..0 of hands();
```

which would be equivalent to

```
party[1].left, party[1].right : boolean;
party[0].left, party[0].right : boolean;
```

As with signals, multidimensional arrays may be created.

6.6 Defined types

A type definition (typedef) is a special kind of module declaration with no parameters, and a slightly different syntax. The definition of "hands" above can equivalently be written as

```
typedef hands struct{
  left, right : boolean;
}
```

The general form of this declaration is

```
typedef <name> <type>
```

where `<type>` is any legal type specification.

7 Conditionals

Assignments or groups of assignments may be made conditional. This is especially useful when several assignments all depend on the same condition – it avoids repeating the conditional structure in each assignment.

7.1 Simple conditionals

The basic conditional structure is

```
if(<condition>)
  <stmt1>
else
  <stmt2>
```

A "statement" can be either an assignment, or a group of statements delimited by curly brackets.

The effect of the statement

```
if(c)
  x := foo;
else
  x := bar;
```

is exactly equivalent to

```
x := c ? foo : bar;
```

If x is assigned in the "if" part, but not assigned in the "else" part, then x is undefined when the condition is false. This means that `x` can take any value in its type. Similarly, if x is assigned in the "else" part, but not in the "if" part, then x is undefined when the condition is true. For example,

```
if(c)
  x := foo;
else
  y := bar;
```

is equivalent to

```
x := c ? foo : undefined;
y := c ? undefined : bar;
```

Mathematically, `undefined` is the set of all possible values.

If next(x) is assigned in one part of the conditional, but not the other, then

```
next(x) = x;
```

is the default. For example,

```
if(c)
  next(x) := foo;
else
  next(y) := bar;
```

is equivalent to

```
next(x) := c ? foo : x;
next(y) := c ? y : bar;
```

Conditionals are statements, and therefore can be nested inside conditionals. Groups of statements can also be nested inside conditionals. For example:

```
if(c)
{
  x := foo;
  if(d)
    next(y) := bar;
  else
    next(z) := bar;
}
else
  x := bar;
```

The "else" part may be omitted, although this is hazardous. It can result in an ambiguity as to which "if" a given "else" corresponds to, if there are nested conditionals. Care should be take to use curly braces to disambiguate. Thus, instead of:

```
if(c)
  if(d)
    <stmt>
  else
    <stmt>
```

the prefered usage is:

```
if(c){
  if(d)
    <stmt>
  else
    <stmt>
}
```

The effect of:

```
if(c)
  <stmt>
```

is equivalent to:

```
if(c)
  <stmt>
else {}
```

7.2 Defaults

The "default" construct provides a way of automatically filling in the cases where a signal is undefined with a default value. The syntax is:

```
default
  <stmt1>
in
  <stmt2>
```

The effect of this statement is to use the assignments in `<stmt1>` in any cases in `<stmt2>` where the given signal is unassigned. For example,

```
default
  x := foo;
in
{
  if(c)
  {
    x := bar;
```

```
    next(y) := y + 1;
  }
  else
    next(y) := y + 2;
}
```

is equivalent to

```
next(y) := c ? y + 1 : y + 2;
x := c ? bar : foo;
```

An assignment to next(x) may also appear in the default statement. The effect is again to insert the default assignment in any cases where next(x) is not defined. Default statements may be nested inside conditionals, and vice-versa, and groups of statements may appear in both the "default" part and the "in" part.

7.3 Complex conditionals – switch and case

The complex conditionals are "case" and "switch". A "case" statement has the form:

```
case{
  <cond1> : <stmt1>
  <cond2> : <stmt2>
  ...
  <condn> : <stmtn>
  [default : <dftlstmt>]
}
```

This statement is exactly equivalent to

```
if (<cond1>) <stmt1>
else if (<cond2>) <stmt2>
...
else if (<condn>) <stmtn>
[else <dfltstmt>]
```

Note this means that if all the conditions are false, and there is no default statement, then no assignments are made.

A "switch" statement has the form:

```
switch(<expr>){
  <case1> : <stmt1>
  <case2> : <stmt2>
  ...
  <casen> : <stmtn>
  [default : <dftlstmt>]
}
```

This is exactly equivalent to:

```
case{
  <expr> in <case1> : <stmt1>
  <expr> in <case2> : <stmt2>
  ...
  <expr> in <casen> : <stmtn>
  [default : <dftlstmt>]
}
```

Note that the set inclusion operator “in” is used instead of “=”. This means that each case may be a set of values rather than a single value. For example, if we want a counter that waits for a signal “start”, counts to seven, asserts a signal “done”, and resets, we might write:

```
default
  done := 0;
in
  switch(count){
    0 : if start then next(count) := 1;
    1..6 : next(count) := count + 1;
    7 : {
      next(count) := 0;
      done := 1;
    }
  }
```

8 Constructor loops

A looping construct is provided for expressing regular structures more succinctly. Loops are simply unrolled by the compiler into the equivalent “in-line” code.

8.1 Basic for-loops

For example,

```
for(i = 0; i < 3; i = i + 1){
  x[i] := i;
}
```

is in every way equivalent to

```
x[0] := 0;
x[1] := 1;
x[2] := 2;
```

The general form of the loop is

```
for(var = init; cond; var = next)
  <stmt>
```

The loop is unrolled in the following way: initially, `var` is set to `init`. Then, while `cond` is true, `stmt` is instantiated, and `var` is set to `next`. The loop variable `var` may appear in `stmt`. Each occurrence of `var` is replaced by its current value.

8.2 Creating arrays of instances

One important use of loops is to create arrays of module instances. For example, to create a three bit counter as an array of counter bits, we could write:

```
bits : array 2..0;

for(i = 0; i < 2; i = i + 1)
  bits[i] : counter_bit(carry[i],clear,count[i],carry[i+1]);
```

Note that `bits` is first declared as a generic array. Then the elements of the array are "filled in" inside the loop. In this way, each counter bit is connected to the appropriate signal, as a function of the loop index `i`.

Also note that module instances can be nested inside conditionals, provided that the condition evaluates to a constant at compile time. Since loops are unrolled at compile time, a loop index counts as a constant. Thus, for example, if we want to use a special module "`special_bit`" for bit 0 of the counter, we could write:

```
bits : array 2..0;

for(i = 0; i < 2; i = i + 1){
  if(i = 0)
    bits[i] : special_bit(carry[i],clear,count[i],carry[i+1]);
  else
    bits[i] : counter_bit(carry[i],clear,count[i],carry[i+1]);
}
```

8.3 Creating parameterized modules

Compile time constants can also be passed as parameters to modules. This allows us to write a generic n-bit counter module, which takes n as a parameter:

```
MODULE nbit_counter(n,carry_in,clear,count,carry_out)
{
  INPUT carry_in, clear : boolean;
  OUTPUT count (n - 1)..0 : boolean;
  OUTPUT carry_out : boolean;
```

```
  bits : array (n - 1)..0;
  carry : array n .. 0 of boolean;

  for(i = 0; i < n; i = i + 1)
    bits[i] : counter_bit(carry[i],clear,count[i],carry[i+1]);

  carry_out := carry[n];
}
```

The ability to nest module instances inside conditionals even makes it possible to write recursively defined modules. For example, the following code builds an n-input "or" gate as a balanced tree of 2-input "or" gates:

```
MODULE or_n(n,inp,out)
{
  INPUT inp : array 0..(n - 1) of boolean;
  OUTPUT out : boolean;

  case{
    n = 1 : out := inp[0];
    n = 2 : or2(inp[0],inp[1],out);
    default: {
      x,y : boolean;
      or_n(n / 2, inp[0 .. (n / 2 - 1)], x);
      or_n(n - n / 2, inp[(n / 2) .. n], y);
      or_2(x,y,out);
    }
  }
}
```

8.4 Chained constructor loops

It is commonly necessary to select the first element of an array satisfying a certain condition, or to perform some other computation involving prioritizing an array. A looping construct called "chain" is provided for this purpose. It acts exactly like a "for" loop, except that the assignments in one iteration of the loop act as defaults for the assignments in subsequent iterations. This makes it possible to assign a signal in more than one iteration of the loop, with the later assignment taking priority over the earlier assignment.

For example, suppose we want to specify a "priority encoder", that inputs an array of signals, and outputs the index of the highest numbered signal that is true. Here is a priority encoder that inputs an array of boolean signals of size "n":

```
MODULE priority_n(n,inp,out)
{
```

```
  INPUT inp : array 0..(n - 1) of boolean;
  OUTPUT out : 0..(n-1);

  chain(i = 0; i < n; i = i + 1)
    if (inp[i]) out := i;
}
```

Depending on the contents of the array "inp", the signal "out" might be assigned many times in different iterations of the loop. In this case, the last assignment is given precedence.

The construct:

```
chain(i = 0; i < 4; i = i + 1)
  <stmt>
```

is in every way equivalent to:

```
default
  <stmt with i = 0>
in default
  <stmt with i = 1>
in default
  <stmt with i = 2>
in
  <stmt with i = 3>
```

9 Expressions

An "expression" combines signals using a collection of operators. These operators include:

- boolean operators ("and", "or", "not" and "xor"),
- conditional operators ("if-then-else", "case" and "switch")
- arithmetic operators ("+", "-", "*", "/", "mod", and shifts)
- comparison operators ("=", "<", ">", ">=", "<=")
- set operators (union, integer subrange and inclusion)
- vector operators (concatenation, subrange)
- conversion operations (integer to bit vector and vice versa)

Since signals are sequences of values, all of these operators apply to the elements of a sequence one-by-one. Thus, if

$$
\begin{aligned}
x &= x_1, x_2, x_3, \ldots \\
y &= y_1, y_2, y_3, \ldots
\end{aligned}
$$

and $\cdot$ is a binary operator, then

$$x \cdot y = (x_1 \cdot y_1), (x_2 \cdot y_2), (x_3 \cdot y_3), \ldots$$

The following describes the various operators as they apply to individual values.

9.1 Parentheses and precedence

Parentheses "()" may always be put around an expression, without changing its value. If parentheses are omitted, then the order of operators is determined by their priority. The operators are listed here in order of priority, from "strongest binding" to "weakest binding":

```
::                  (concatenation)
-                   (unary minus sign)
*,/,<<,>>           (mult, div, left shift,right shift)
+,-                 (add, subtract)
mod                 (integer mod)
in                  (set inclusion)
union               (set union)
=,~=,<,<=,>,>=      (comparison operators)
~                   (not)
&                   (and)
|,^                 (or, exclusive or)
<->                 (iff)
->                  (implies)
,                   (tuple separator)
?:                  (conditional)
..                  (integer subrange)
```

9.2 Integer constants

Integer constants may appear in expressions, and are optionally signed decimal numbers in the range $2^{-31} \ldots (2^{31} - 1)$.

9.3 Symbolic constants

Symbolic constants may be declared by including them in a type declaration, such as,

```
x : {ready, willing, able};
```

The three symbols `ready`, `willing` and `able` are treated as distinct constants, which are also distinct from all the integers. A given symbolic constant may not appear in two different types. For example,

```
x : {foo, bar, baz};
y : {red, green, foo};
```

is illegal, since `foo` appears in two distinct types. This restriction is made so that programs may be type checked.

9.4 Boolean operators

The boolean operators are "`&`", for logical and, "`|`" for logical or, "`~`" for logical not, "`^`" for exclusive or , "`->`" for implies, and "`<->`" for if-and-only-if (exclusive nor). The boolean values are 0 (false) and 1 (true).

The "`&`" operator obeys the following laws:

```
x & 0 = 0
0 & x = 0
x & 1 = 1
1 & x = x
```

If neither x nor y is a boolean value, then "`x & y`" is undefined. (recall that an "undefined" expression yields the set of all possible values, which in the case of a boolean expression is {0,1}). In particular,

```
1 & undefined = undefined
0 & undefined = 0
```

That is, "undefined" values behave like "X" values in typical logic simulators. You can write an undefined value as simply {0,1}.

Also note that

```
1 & 37 = 37
0 & 37 = 0
37 & 37 = undefined
```

The other boolean operators behave similarly, obeying:

```
0 | x = x
x | 0 = x
1 | x = 1
x | 1 = 1

0 ^ x = x
x ^ 0 = x
1 ^ x = ~x
x ^ 1 = ~x

x -> y = ~x | y
x <-> y = ~(x ^ y)
```

9.5 Conditional operators ("if", "case" and "switch")

The simple conditional operator has the form

```
x ? y : z
```

It yields y if x is 1 (true) and z if x is 0 (false). If x is not a boolean value, it yields `undefined`.

The complex conditionals are "case" and "switch". The expression

```
case{
  c1 : e1;
  c2 : e2;
  ...
  cn : en;
  [default : ed;]
}
```

is equivalent to

```
c1 ? e1 : c2 ? e2 : ... cn ? en : ed
```

if there is a default case, and otherwise

```
c1 ? e1 : c2 ? e2 : ... cn ? en : undefined
```

That is, if all the conditions c1…cn are false, and there is no default case, then the case expression is undefined.

The expression

```
switch(x){
  v1 : e1;
  v2 : e2;
  ...
  vn : en;
  [default : ed;]
}
```

is equivalent to

```
(x in v1) ? e1 : (x in v2) ? e2 : ... (x in vn) ? en : ed
```

if there is a default case, and otherwise

```
(x in v1) ? e1 : (x in v2) ? e2 : ... (x in vn) ? en : undefined
```

That is, the switch expression finds the first set `v`*i* that contains the value `x`, and returns the corresponding `e`*i*. The `v`*i* can also be single values – these are treated as the set containing only the given value.

9.6 Representing state machines using conditionals

As an example, suppose we have a state machine with one boolean input "choice", that starts in state "idle", then depending on "choice" goes to either state "left" or "right", and finally returns to state "idle". Using a "case" expression, we could write:

```
next(state) :=
  case{
    state = idle : choice ? left : right;
    default : idle;
  };
```

The equivalent using a switch statement would be:

```
next(state) :=
  switch(state){
    idle : choice ? left : right;
    default : idle;
  };
```

The values in a switch statement can also be "tuples" (lists of expressions separated by commas, see section on tuples). Using this notation, we can write the above state machine as

```
next(state) :=
  switch(state,choice){
    (idle,            1)    : left;
    (idle,            0)    : right;
    ({left,right}, {0,1})   : idle;
  };
```

If we want to add outputs "left_enable" and "right_enable" to our state machine, to indicate that the state is "left" and "right" respectively, we can use a switch expression that returns a tuple. Thus:

```
(next(state),left_enable,right_enable) :=
  switch(state,choice){
    (idle,           1)     : (left,  0, 0);
    (idle,           0)     : (right, 0, 0);
    (left,       {0,1})     : (idle,  1, 0);
    (right,      {0,1})     : (idle,  0, 1);
  };
```

This provides a fairly succint way of writing the truth table of a state machine, with current state and inputs on the left, and next state and outputs on the right.

9.7 Arithmetic operators

The arithmetic operators are

```
+        addition
-        subtraction and unary minus sign
*        multiplication
/        integer division
mod      remainder of division
<<       left shift
>>       right shift
```

All results of arithmetic on integers are modulo 2^{32}, in the range $-2^{31} \ldots (2^{31} - 1)$. [3]

The operators "*", "/" and "mod" obey the law

```
y * (x/y) + (x mod y) = x
```

The remainder is *always* positive.

The expression "`x << y`" is equivalent to "`x *` 2^y". Similarly, "`x >> y`" is equivalent to "`x /` 2^y".

9.8 Comparison operators

The comparision operators are

```
=        equal
~=       not equal
<        less than
>        greater than
<=       less than or equal
>=       greater than or equal
```

When applied to integers, all return boolean values. The "equal" and "not equal" operators may also be applied to symbolic constants. Any integer is considered not equal to any symbolic constant. The inequality operators are undefined if either operand is a symbolic constant.

9.9 Set expressions

A set is specified as a list of elements between curly brackets:

```
{ elem, ... , elem }
```

Note that a set cannot be empty – there must be at least one element. Each element can be one of the following:

[3] N.B. Unsigned arithmetic on integers of arbitrary precision can be performed on bit vectors, however. See section on vectors.

- An expression x. In this case, the value of x is included in the set. Note that if x itself represents a set of values, then all elements of x are included.
- A subrange `x .. y`, where x and y are integer valued expressions. In this case all elements in the subrange `x .. y` are included in the set.
- A guarded expression `c ? e`. In this case, the value of e is included in the set if the condition c is true.

The set `x..y` can be abbreviated to `x..y`.

Note that a set expression may represent the empty set in the case that all elements are guarded, and all the guard conditions are false. In this case the result of the set expression is `undefined`. Thus, for example:

```
{1 ? foo, 1 ? bar}  =  {foo,bar}
{1 ? foo, 0 ? bar}  =  {foo}
{0 ? foo, 0 ? bar}  =  undefined
```

The reason for this rule is that a set expression is interpreted (with one exception, below) to represent a non-deterministic choice between the values in the set. A choice between the empty set of values is not meaningful.

9.9.1 The set inclusion operator

There is one operator for testings sets: the set inclusion operator "in". The expression "x in y" returns true if the value x is contained in the set y. The "in" operator obeys the following law:

```
(x in {y,z}) = ((x in y) | (x in z))
```

9.9.2 Extension of operators to sets

Most of the operators extend to sets, in a way which is consistent with the interpretation of sets as independent nondeterministic choices. Generally, a unary operator f obeys the law

```
f{x,y} = {f(x),f(y)}
```

Thus, for example,

```
-(2..3) = (-2..-3)
```

and

```
~{0,1} = {1,0}
```

For a binary operator *, we have

```
{x,y} * z = {x * z, y * z}
x * {y,z} = {x * y, x * z}
```

For example,

```
3 + {4,5} = {7,8}
```

and

```
0 & {0,1} = 0
1 & {0,1} = {0,1}
```

(which are actually special cases of the laws given above for "and").

This behavior of sets is somewhat counterintuitive when the equality operator is applied to sets. For example, the result of

```
{a,b} = {a,b}
```

is not equal to 1 (true). The way to understand this is to think of each set as representing an arbitrary choice among its elements. Thus, the result of the above expression is the set $\{0, 1\}$, since we may choose equal elements or we may choose unequal elements.

The exception to the above rule is the "in" operator, which compares a value and a set of values. In this case, only the left represents a nondeterministic choice. That is:

```
{x,y} in z = {x in z, y in z}
```

However, as stated previously,

```
x in {y,z}   =   (x in y) | (x in z)
```

[N.B. This makes "in" the only operator in the language which is not monotonic with respect to set containment. The "in" operator is only monotonic in its left argument. A formal verification system that relies on monotonicity (such as ternary symbolic simulation) should allow only constant sets on the right hand side of "in".]

9.9.3 Comprehension expressions

A set may be built iteratively using the construction `{f(i):i=x..y}`, where `f(i)` is some expression containing the parameter `i`, and `x`, `y` are integer constants. This expands to the set `{f(l),...,f(r)}`. For example, the expression:

```
{ i*i : i = 1..5 }
```

is equivalent to the set:

```
{1,4,9,16,25}
```

The form `{f(i) : i = x..y, c(i)}` represents the set of all `f(i)`, for `i = x..y` such that condition `c(i)` is true. That is,

```
{f(i) : i = x..y, c(i)}  =  {c(x) ? f(x), ... ,c(y) ? f(y)}
```

For example,

```
{ i : i = 1..5, i mod 2 = 1}  =   {1,3,5}
```

Or, for example, if `y` is of type `array 1..5 of boolean`, then

```
{ i : i = 1..5, y[i] }
```

represents the set of all indices `i` such that element `i` of array `y` is true. Note that in this case, if none of the elements of `y` is true, the result is `undefined`. This provides a straightforward way to describe a nondeterministic arbiter. In addition, the contruct provides a way to describe a nondeterministic choice among all the number in a given range *except* a specified number:

```
x := {i : i in 1..5, i ~= j};
```

10 Vectors and vector operators

A vector is a fixed-length string of values. It differs from an array in several respects:

- The elements of a vector are not indexed, hence vectors cannot be subscripted.
- Vectors may be concatenated.
- Arithmetic and comparison operators have special meanings when applied to vectors (they interpret the vectors as unsigned binary numbers).
- Logical operators have special meanings when applied to vectors (they perform "bit-wise" operations).

A vector `x` of `n` elements is denoted by a non-empty, comma-separated list of elements within square brackets:

```
[xn, ..., x1, x0]
```

The nth element can be extracted from a vector by the function `nth`, which takes a vector as its first argument, and an integer as its second argument. The function `nth` numbers the elements of the vector from zero on the right. So, for example,

```
nth([5,4,3,2,1,0], 2) = 2
```

10.1 The concatenation operator

A vector may also be constructed using the concatenation operator "::". As an example,

```
(0 :: 1 :: 1 :: 0)
```

is a vector of length 4. That is, the 4 boolean values in this expression are treated as vectors of length 1, and concatenated to produce a vector of length 4.

The concatenation operator is associative. Thus, for example,

```
((0 :: 1) :: (1 :: 0)) = (0 :: 1 :: 1 :: 0)
```

10.2 Extension of operators to vectors

Logical operators extend to vectors in the obvious way, by applying them one-by-one to the elements of the vector. That is, if f is a unary operator, then

```
f[x,y] = [f(x),f(y)]
```

For example,

```
~[0,1,0] = [1,0,1]
```

Binary logical operators extend similarly. That is, if * is a binary operator, then

```
[w,x] * [y,z] = [w*y, x*z]
```

For example,

```
[0,1] & [1,1] = [0,1]
```

A binary logical operator may be applied to two vectors only when the vectors have the same length.

10.3 Vector coersion operator

When combining vectors of different length with a binary operator, the shorter vector is prepended with a vector of zeros to make the vectors the same length. Thus, for example,

```
[a,b,c,d] * [e,f]
```

is equivalent to

```
[a,b,c,d] * [0,0,e,f]
```

In general, if x and z are booleans, then

```
(w :: x) * z = (w * 0) :: (x * z)
x * (y :: z) = (0 * y) :: (x * z)
```

The exceptions to the above rule are

- the arithmetic operators,
- the comparison operators, and
- the conditional operator
- the union and "in" (set inclusion) operators

10.4 Arithmetic on vectors

Arithmetic operators applied to vectors treat the vectors as unsigned binary numbers. If the vectors are of unequal length, the shorter vector is prepended with a vector of zeros to make the lengths equal. Thus

```
[0,1,1,0] + [1,0]
```

is equivalent to

```
[0,1,1,0] + [0,0,1,0]
```

The arithmetic operator is then applied to the unsigned binary numbers represented by the two vectors, yielding an unsigned binary representation of the result, of the same length as the argument vectors. The operators are the same as they are on integers, except the result is modulo 2^n, where n is the vector length, and the result is always positive (in the range $0 \ldots (2^n - 1)$). For example,

```
  [0,1,1,0]
+ [0,0,1,0]
= [1,0,0,0]
```

and

```
  [1,1,1,0]
+ [0,0,1,0]
= [0,0,0,0]
```

[N.B. Since the arithmetic is modular, it doesn't actually matter whether we look at it as signed or unsigned, except that extension is always by zeros. It would make sense to introduce a special kind of vector that sign-extends rather than zero-extending, to allow signed arithmetic. Or at least a sign-extension operator.]

10.5 Comparison operators on vectors

Comparison operators on vectors operate in the same manner as arithmetic opertors: the shorter vector is prepended with zeros, and the resulting vectors are compared as *unsigned* binary numbers.

10.6 Vector sets

Important note: The "union" operator can be applied to vectors, to produce a set of vectors. In particular, one can express a nondeterministic choice between the vectors [0,1] and [1,0] by writing either

```
[0,1] union [1,0]
```

Note that the following is not legal, however:

```
{[0,1], [1,0]}
```

As with other operators, vectors are padded with zeros to the same length before being unioned. The "in" operator may also be applied to vetor sets. In general:

```
[a,b] in ([c,d] union [e,f])
```

is equivalent to

```
([a,b] = [c,d]) | ([a,b] = [e,f])
```

10.7 Coercion of scalars to vectors

An integer expression is *coerced* to a vector expression whenever:

- it is combined with a vector by a binary operator or conditional, or
- it is assigned to a vector of signals

The length of the vector representation of an integer is 32 bits. The shorter of the two argument vectors is then padded with zeros before applying the opertion.

```
[0,1,1] + 17
```

yields a 32 bit vector representation of the number 20.

10.8 Explicit coercion operators

An vector expression may be explicitly coerced to a vector of a given length by applying the "bin" function. The expression

```
bin(n,val)
```

causes the vector "val" to be either shortened to length n, or padded with zeros to length n. If "val" is an integer, is is first coerced to a 32 bit vector, and then truncated or padded. Thus, for example

```
bin(3,17) = [0,0,1]
bin(4,17) = [1,0,0,1]
```

and

```
[0,1,1] + bin(4,17)
```

is equal to

```
  [0,0,1,1]
+ [1,0,0,1]
= [1,1,0,0]
```

Note that coercing a negative integer to longer than 32 bits will not produce the intuitively correct result, since "bin" treats its argument as an unsigned number. The "sbin" operator is equivalent to "bin", except that it sign extends rather than zero extending. Thus, for example "`sbin(64,-1)`" is a string of 64 ones.

10.9 Coercion of array variables to vectors

A reference to an variable of type "array" without a subscript will be converted to a vector. For example, if x is declared:

```
x : array 0..3 of boolean;
```

then the expression "x" is equivalent to

```
[x[0],x[1],x[2],x[3]]
```

Note that the elements of x occur in the order given by the type declaration. Thus, for example, if x is declared:

```
x : array 3..0 of boolean;
```

then the expression "x" is equivalent to

```
[x[3],x[2],x[1],x[0]]
```

This has consequences when combining "big endian" and "little endian" arrays. For example, if we have

```
x : array 3..0 of boolean;
y : array 0..3 of boolean;
```

Then the expression "x = y" is equivalent to

```
(x[3] = y[0]) & (x[2] = y[1]) & (x[1] = y[2]) & (x[0] = y[3])
```

The only difference between big-endian and little-endian binary numbers is the order in which they are converted to vectors (which determines which element of the array is considered most significant and least significant).

10.10 Array subranges

A array variable may be explicitly coerced to a vector by specifying a subrange of bit indices. For example, the expression

```
x[2..5]
```

is equivalent to

```
[x[2],x[3],x[4],x[5]]
```

Similarly,

```
x[5..2]
```

is equivalent to

```
[x[5],x[4],x[3],x[2]]
```

Subranges may be used, for example, to extract bitfields, or to reverse the declared order bits in an array.

10.11 Assignments to vectors

Assignments may also be made to vectors of signals. When a value is assigned to a vector, the following rules apply:

- An integer value on the right hand side is first coerced to a 32 bit vector.
- A vector value on the right hand side is padded or truncated to the same length as the left hand side of the assignment.

For example,

```
[x,y] := [1,1,0];
```

is equivalent to

```
x := 1;
y := 0;
```

That is, the leftmost (high order) bit is dropped to make the vectors the same length. On the other hand

```
[x,y] := 1;
```

is equivalent to

```
x := 0;
y := 1;
```

since the integer is coerced to a vector, and then truncated to length 2.

The assignment

```
[x,y,z] := [1,0];
```

is equivalent to

```
x := 0;
y := 1;
z := 0;
```

since the vector on the right-hand-side is zero-extended.

Important note: A vector of signals may not be assigned a nondeterministic value.

10.12 Assignments to arrays

An unsubscripted array reference on the left hand side of an assignment is converted to a vector (see above). This means that the result of the assignment depends on whether the vector is "big-endian" or "little-endian". For example, if:

```
x : array 0..1 of boolean;
```

then

```
x := [1,0];
```

is equivalent to

```
[x[0],x[1]] := [1,0];
```

which is equivalent to

```
x[0] := 1;
x[1] := 0;
```

On the other hand, if:

```
x : array 1..0 of boolean;
```

then

```
x := [1,0];
```

is equivalent to

```
[x[1],x[0]] := [1,0];
```

which is equivalent to

```
x[1] := 1;
x[0] := 0;
```

10.13 Vectors as inputs and outputs

The above rules regarding vector assignments have consequences when vectors are passed as parameters to modules. For example, suppose we have a module:

```
MODULE foo(x)
{
  INPUT x : array 1..0 of boolean;
  ...
}
```

Suppose we create an instance of "foo" as follows:

```
bar : foo(y);
```

This is equivalent to:

```
bar.x : array 1..0 of boolean;
bar.x := y;
...
```

The meaning of this depends on whether “y” is big-endian or little endian. If “y” is declared in the same order as “bar.x”:

```
y : array 1..0 of boolean;
```

then we have

```
bar.x[1] := y[1];
bar.x[0] := y[0];
```

On the other hand, if “y” is in the opposite order:

```
y : array 0..1 of boolean;
```

then

```
bar.x[1] := y[0];
bar.x[0] := y[1];
```

That is, passing a “big-endian” array to a “little-endian” parameter, results in a reversal of the index order of the elements. What remains constant is the value as a binary number.

Note that as a result of the above rules for vector assignment, inputs may be truncated, or zero-extended. For example, if we instantiate “foo” as follows:

```
bar : foo([0,1,0]);
```

the effect will be

```
bar.x[1] := 1;
bar.x[0] := 0;
```

since the vector [0,1,0] will be truncated to [1,0]. On the other hand,

```
bar : foo([1]);
```

will give us

```
bar.x[1] := 0;
bar.x[0] := 1;
```

since the integer 1 will be coerced to the vector [0,1].

The same remarks apply to outputs. That is, suppose we have a module

```
MODULE zip(y)
{
  OUTPUT y : array 1..0 of boolean;
  ...
}
```

An instance

```
bar : zip(x);
```

is equivalent to

```
bar.y : array 1..0 of boolean;
x := bar.y;
```

This means that if "x" has length shorter than "bar.y", then "x" will get the low order bits of "bar.y". Similarly, if "x" is longer, then it will get "bar.y" extended with zeros. If "x" is an array declared in the opposite order to "bar.y", then "x" will get "bar.y" reversed, and so on.

10.14 Iteratively constructing vectors

A vector may be constructed iteratively using the construction `[f(i):i=l..r]`, where `f(i)` is some expression containing the parameter `i`, and `l`, `r` are integers. This expands to the vector `[f(l),...,f(r)]`. For example, the expression:

```
[ i*i : i = 1..5 ]
```

is equivalent to:

```
[1,4,9,16,25]
```

On the other hand:

```
[ i*i : i = 5..1 ]
```

is equivalent to:

```
[25,16,9,4,1]
```

10.15 Reduction operators

The associative operators `::` (concatenation), `&`, `|`, `^`, `+`, `*` and `merge` may be applied to vectors as "reduction" operators . For example,

```
&[w,x,y,z]
```

is equivalent to

```
(w & x & y & z)
```

and so on. Reduction operators may be combined with the iterated vector constructor. For example, to compute the sum of the first five squares, we could write:

```
+[ i*i : i = 1..5]
```

Note that for the non-commutative operator `::`, the order of the range specification matters. For example

```
::[ i*i : i = 1..5]
```

produces `[1,4,9,16,25]`, while

```
::[ i*i : i = 5..1]
```

produces `[25,16,9,4,1]`. Also note that using `::` as a reduction operator makes it possible to construct a vector by repeating a pattern. For example,

```
::[ [0,1] : i = 1..3]
```

is equivalent to `[0,1,0,1,0,1]`.

Reduction operators do not coerce their arguments to vectors. A reduction operator applied to a scalar operand has no effect. Thus, `+3 = 3` (and not `2`, fortunately!).

10.16 Vectors as conditions

If a vector appears as the condition in a conditional expression or statement, then the logical "or" of the elements of the vector is taken as the condition. Thus, for example,

```
[x,y,z] ? foo : bar
```

is equivalent to

```
(x | y | z) ? foo : bar
```

In particular, this means that when a bit vector is used as a condition, it is considered to be true if and only if it is non-zero.

11 Assertions

An *assertion* is a condition that must hold true in every possible execution of the program. Assertions in SMV are written in a "linear time" temporal logic, that makes it possible to succinctly state propositions about the relation of events in time.

11.1 Temporal formulas

Temporal formulas may contain all of the usual expression operator of SMV, plus the temporal operators G, F, X and U. The meanings of these operators are as follows:

- `X p` is true at time t if `p` is true at time $t + 1$.
- `G p` is true at time t if `p` is true at all times $t' \geq t$.
- `F p` is true at time t if `p` is true at some time $t' \geq t$.
- `p U q` is true at time t if `q` is true at some time $t' \geq t$, and for all times $< t'$ but $\geq t$, `p` is true.

A temporal formula is true for a given exectution of the program if it is true at the initial time ($t = 0$).

11.2 The assert declaration

A declaration of the form

```
assert p;
```

where p is a temporal formula, means that every execution of the program must satisfy the formula p. An execution that does not satisfy the formula is called a *failure* of the program.

An assertion may be given a name. For example:

```
foo : assert p;
```

This does not change the semantics of the program, but provides an identifier "foo" for refering to the given assertion. The code

```
foo : assert p;
foo : assert q;
```

is equivalent to

```
foo : assert p & q;
```

11.3 Using. . . Prove declarations

A `using`. . . `prove` declaration tells the verification system to use one assertion as an assumption when verifying another. A declaration of the form

```
using foo prove bar;
```

where foo and bar are identifiers for assertions, tells the verification system to use assertion foo as an assumption when proving assertion bar. A list of assumptions may also be used:

```
using a1,a2,...,an prove bar;
```

Such a "proof" may not contain circular chains of reasoning. Thus, for example,

```
using foo prove bar;
using bar prove foo;
```

is illegal.

12 Refinements

N.B. This section is incomplete and under construction

The mechanism of "refinement" in SMV allows one model to represent the behavior of a design simultaneously at many levels of abstraction. It also allows one to verify in a compositional manner that each level of the design is a correct implementation of the level above.

The basic object in the refinement system is a "layer". A layer is a named collection of assignments. For example:

```
layer P : {
  x := y + z;
  next(z) := x;
}
```

represents a layer named P, which contains assignments to signals x and z. Within a layer the single assignment rule applies. That is, any given signal may be assigned only once. However, a signal may be assigned in more than one layer.

One layer may be declared to "refine" another. The syntax for this declaration is:

```
P refines Q;
```

where P and Q are names of layers. If P refines Q, then an assignment to any signal s in P supercedes the corresponding assignment to s in Q. For example, suppose that layer Q is defined as follows:

```
layer Q : {
  y := z;
  next(z) := 2 * y;
}
```

The net functional effect of these declarations is equivalent to:

```
x := y + z;
y := z;
next(z) := x;
```

That is, the assignment to z in P supercedes the assignment to z in Q, because P refines Q. Any assignment that is superceded in this way becomes a part of the specification. That is, in our example, every trace of the system must be consistent with

```
next(z) := 2 * y
```

at all times. This proposition is given the name "z//Q", meaning "the assignment to signal z in layer Q". Note that the property z//Q is true in the case of our example, since at all times

```
x  =  y+z  =  z+z  =  2*z
```

Thus, we can infer that every trace of our system is also a trace of the system consisting only of the layer Q. Put another way, our system satisfies specification Q (and also, trivially, specification P).

12.1 The refinement relation

The refinement relation between layers is by definition transitive. Thus if we have:

```
P refines Q;
Q refines R;
```

then by implication

```
P refines R;
```

The refinement relation may not be circular. Thus

```
P refines P
```

is an error. The *implementation* of a signal is the assignment to that signal whose layer is minimal with respect to the refinement relation. If no unique minimal assignment to a signal exists, the program is in error.

12.1.1 Circular assignments

A circularity error occurs if there is a cycle of zero-delay assignments amongst the union of all assignments in all layers. Thus for example, the following program:

```
layer Q : {
  x := y;
}
layer P : {
  y := x;
  next(x) := y;
}
P refines Q;
```

is erroneous, even though it is functionally equivalent to the non-circular program:

```
y := x;
next(x) := y;
```

12.2 Compositional verification

In order to verify a given assignment x//P, where x is a signal and P is layer, it is allowed to use any other assignment y//Q as an assumption (with one proviso, below). The syntax for this is:

```
using x//P prove y//Q;
```

In this case, x//P is refered to as the "assumption" and y//Q as the "guarantee". The one restriction on the use of this statement is that if x and y are identical, then P must refine Q. Other than this, any use is allowed, including circularities. For example, it is legitimate to write:

```
using x//P prove y//P;
using y//P prove x//P;
```

As a example, suppose we have:

```
layer P : {
  x := 0;
  y := 0;
}
layer Q : {
  init(x) := 0;
  next(x) := y;
  y := x;
}
Q refines P;
using x//P prove y//P;
using y//P prove x//P;
```

That is, in essence, the "using" declarations say that in order to prove that x is always zero, we can assume that y is always zero, and vice versa.

12.3 The using...prove declaration

This declaration has the form form

```
using
  p_1, p_2, ..., p_k
prove
  q_1, q_2, ..., q_m
;
```

where `p_1, p_2, ..., p_k` and q_1, q_2, ..., q_m are properties. The meaning of the declaration is that properties `p_1, p_2, ..., p_k` are to be taken as assumptions when proving the properties q_1, q_2, ..., q_m.

If assumptions are not declared for a given property, then that property inherits the assumptions of its parent. For example, if we wish to assume property `foo` when proving `bar.a`, `bar.b` and `bar.c`, it is sufficient to declare

```
using foo prove bar;
```

If there is no declaration of assumptions for `bar.a`, then it will inherit the assumption of `foo` from its parent, `bar`. Note, however, that if we include the declaration

```
using baz prove bar.a;
```

then only property `baz` is used to prove `bar.a`.

Similarly, if we wish to assume a collection of properties `foo.a`, `foo.b` and `foo.c` when proving a property `bar`, it is sufficient to declare:

```
using bar prove foo;
```

It is allowed to use several assignments to the same signal as assumptions. For example:

```
using x//P1, x//P2 prove y//P
```

In this case, the conjunction of the two assumptions is used.

12.4 Abstract signals

In some cases, it may be necessary to introduce auxiliary signals that are used as part of the specification, or part of the proof, but do not belong to the system being verified. Such signals are introduced by the keyword `abstract`, as follows:

```
abstract <signal> : <type>
```

The implementation of a non-abstract signal may not depend on an abstract signal.

13 Syntax

This section gives a BNF grammar for the SMV language.

13.1 Lexical tokens

A program is a sequence of *lexical tokens*, optionally separated by *whitespace*. A token is either an *atom*, a number, or any of the various keywords and punctation symbols that appear in `typewriter font` in the grammar expressions that follow.

An atom is

- A string consisting of alphanumeric characters and the charaters, and the characters "$" and "_", beginning with an alphabetic character, or
- A string containing any character except the space character, delimited by an initial backslash ("\") and a final space character. The delimiters do not count as part of the atom.

As an example "`foo_123`" is an atom. It is exactly equivalent to "`\foo_123 `" (note the terminating space character). Using backslash and space as delimiters allows any character (including backslash, but excluding space) to be included in an atom.

A number is a string of digits. Whitespace is any string of space characters, tab characters and newline characters.

13.2 Identifiers

The grammar rules for an *identifier* is as follows:

```
id::        atom
       |    id . atom
       |    id . [ expr ]
```

13.3 Expressions

The grammar rules for an *expression* are, in order of precedence, from high to low (note ϵ stands for the empty string):

```
expr::      id
       |    number
       |    { atom,...,atom }
       |    expr :: expr
       |    [-|+|*|&||^] expr
       |    expr ** expr
       |    expr [*|/|<<|>>] expr
       |    expr [+|-] expr
       |    expr mod expr
       |    expr in expr
       |    expr union expr
       |    expr [=| =|<|<=|>|>=] expr
       |    ~ expr
       |    expr & expr
```

```
        |   expr [|||^] expr
        |   expr <-> expr
        |   expr -> expr
        |   expr ? expr : expr
        |   expr .. expr
        |   ( expr )
        |   [ expr,...,expr ]
        |   [ expr,...,expr : atom = expr .. expr ]
        |   bin ( expr , expr )
```

All operators of the same precedence except "?:" associate to the left. For example, `a / b * c` is parsed as `(a / b) * c`. The ternary "?:" associates to the right. Thus

```
a ? b : c ? d : e
```

is parsed as

```
a ? b : (c ? d : e)
```

13.4 Types

The grammar rules for types are:

```
type::      boolean
        |   expr .. expr
        |   { atom,...,atom }
        |   array expr .. expr of type
        |   atom ( expr,...,expr )
```

13.5 Statements

The grammar rules for statements are:

```
stmt::      lhstup : type ;
        |   lhs [:=|<-] expr ;
        |   { block }
        |   if ( expr ) stmt
        |   case { cblk }
        |   switch ( tuple ) { cblk }
        |   [for|chain] ( atom = expr ; expr ; atom = expr ) stmt

lhs::       id
        |   next ( id )
        |   ( lhstup )
lhstup::    ε
        |   lhs
        |   lhstup lhs
```

block::		stmt
	|	block stmt
cblk::		expr : stmt
	|	cblk expr : stmt

13.6 Module definitions

The grammar rules for module definitions are:

module::		`module` atom (params) { block }
params::		ϵ
	|	atom
	|	params , atom

13.7 Programs

The grammar rules for programs are:

prog::		[stmt | module]
	|	prog [stmt | module]