14.7.3 Rules

In generated Verilog, a designer might want to include a comment on rule scheduling signals (such as `CAN_FIRE_` and `WILL_FIRE_` signals), to say something about the actions that are performed when that rule is executed. This can be achieved with a `doc` attribute attached to a BSV rule declaration or rules expression.

The `doc` attribute can be attached to any `rule..endrule` or `rules...endrules` statement. Example:

```
(* doc = "This rule is important" *)
 rule do_something (b);
    x <= !x;
 endrule
```

If any scheduling signals for the rule are explicit in the Verilog output, their definition will be preceded by the comment. Example:

```
// rule RL_do_something
//   This rule is important
assign CAN_FIRE_RL_do_something = b ;
assign WILL_FIRE_RL_do_something = CAN_FIRE_RL_do_something ;
```

If the signals have been inlined or otherwise optimized away and thus do not appear in the Verilog, then there is no place to attach the comments. In that case, the comments are carried to the top module's header. Example:

```
// ...
// Comments on the inlined rule ‘RL_do_something’:
//   This rule is important
//
module mkTop(...);
```

The designer can ensure that the signals will exist in the Verilog by using an appropriate compiler flag, the `-keep-fires` flag which is documented in the *Bluespec Compiler (BSC) User Guide*.

The `doc` attribute can be attached to any `rule..endrule` expression, such as inside a function or inside a for-loop.

As with comments on submodules, if the comments on several rules are the same, and those comments are carried to the top-level module header, the comment is only displayed once.

```
// ...
// Comments on the inlined rules ‘RL_do_something_2’, ‘RL_do_something_1’,
// ‘RL_do_something’:
//   This rule is important
//
module mkTop(...);
```

15 Embedding RTL in a BSV design

This section describes how to embed existing RTL modules, Verilog or VHDL, in a BSV module. The `import "BVI"` statement is used to utilize existing components, utilize components generated by other tools, or to define a custom set of primitives. One example is the definition of BSV primitives (registers, FIFOs, etc.), which are implemented through import of Verilog modules. The `import "BVI"` statement creates a BSV wrapper around the RTL module so that it looks like a BSV module. Instead of ports, the wrapped module has methods and interfaces.

The `import "BVI"` statement can be used to wrap Verilog or VHDL modules. Throughout this section Verilog will be used to refer to either Verilog or VHDL. (One limitatation for VHDL is that BSV does not support two dimensional ports.)

externModuleImport ::= `import "BVI"` [*identifier* `=`] *moduleProto*
  { *moduleStmt* }
  { *importBVIStmt* }
  `endmodule` [`:` *identifier*]

The body consists of a sequence of *importBVIStmts*:

importBVIStmt ::= *parameterBVIStmt*
  | *methodBVIStmt*
  | *portBVIStmt*
  | *inputClockBVIStmt*
  | *defaultClockBVIStmt*
  | *outputClockBVIStmt*
  | *inputResetBVIStmt*
  | *defaultResetBVIStmt*
  | *noResetBVIStmt*
  | *outputResetBVIStmt*
  | *ancestorBVIStmt*
  | *sameFamilyBVIStmt*
  | *scheduleBVIStmt*
  | *pathBVIStmt*
  | *interfaceBVIStmt*
  | *inoutBVIStmt*

The optional *identifier* immediately following the `"BVI"` is the name of the Verilog module to be imported. This will usually be found in a Verilog file of the same name (*identifier.v*). If this *identifier* is excluded, it is assumed that the Verilog module name is the same as the BSV name of the module.

The *moduleProto* is the first line in the module definition as described in Section 5.3.

The BSV wrapper returns an interface. All arguments and return values must be in the `Bits` class or be of type `Clock`, `Reset`, `Inout`, or a subinterface which meets these requirements. Note that the BSV module's parameters have no inherent relationship to the Verilog module's parameters. The BSV wrapper is used to connect the Verilog ports to the BSV parameters, performing any data conversion, such as packs or unpacks, as necessary.

Example of the header of a BVI import statement:

```
import "BVI" RWire =
   module RWire (VRWire#(a))
      provisos (Bits#(a,sa));
   ...
   endmodule: vMkRWire
```

Since the Verilog module's name matches the BSV name, the header could be also written as:

```
import "BVI"
   module RWire (VRWire#(a))
      provisos (Bits#(a,sa));
   ...
   endmodule: vMkRWire
```

The module body may contain both *moduleStmt*s and *importBVIStmt*s. Typically when including a Verilog module, the only module statements would be a few local definitions. However, all module statements, except for method definitions, subinterface definitions, and return statements, are valid, though most are rarely used in this instance. Only the statements specific to *importBVIStmt* bodies are described in this section.

The *importBVIStmt*s must occur at the end of the body, after the *moduleStmt*s. They may be written in any order.

The following is an example of embedding a Verilog SRAM model in BSV. The Verilog file is shown after the BSV wrapper.

```
import "BVI" mkVerilog_SRAM_model =
  module mkSRAM #(String filename) (SRAM_Ifc #(addr_t, data_t))
  provisos(Bits#(addr_t, addr_width),
           Bits#(data_t, data_width));
     parameter FILENAME      = filename;
     parameter ADDRESS_WIDTH = valueOf(addr_width);
     parameter DATA_WIDTH    = valueof(data_width);
     method request (v_in_address, v_in_data, v_in_write_not_read)
                     enable (v_in_enable);
     method v_out_data  read_response;
     default_clock clk(clk, (*unused*) clk_gate);
     default_reset no_reset;
     schedule (read_response) SB (request);
endmodule
```

This is the Verilog module being wrapped in the above BVI import statement.

```
module mkVerilog_SRAM_model (clk,
                             v_in_address, v_in_data,
                             v_in_write_not_read,
                             v_in_enable,
                             v_out_data);
  parameter FILENAME      = "Verilog_SRAM_model.data";
  parameter ADDRESS_WIDTH = 10;
  parameter DATA_WIDTH    = 8;
  parameter NWORDS        = (1 << ADDRESS_WIDTH);

  input                        clk;
  input  [ADDRESS_WIDTH-1:0]   v_in_address;
  input  [DATA_WIDTH-1:0]      v_in_data;
  input                        v_in_write_not_read;
  input                        v_in_enable;

  output [DATA_WIDTH-1:0]      v_out_data;
  ...
endmodule
```

15.1 Parameter

The parameter statement specifies the parameter values which will be used by the Verilog module.

parameterBVIStmt ::= `parameter` *identifier* = *expression* ;

The value of *expression* is supplied to the Verilog module as the parameter named *identifier*. The *expression* must be a compile-time constant. The valid types for parameters are `String`, `Integer` and `Bit#(`*n*`)`. Example:

```
import "BVI" ClockGen =
module vAbsoluteClock#(Integer start, Integer period)
                      ( ClockGenIfc );
    let halfPeriod =  period/2 ;
    parameter initDelay  = start;              //the parameters start,
    parameter v1Width = halfPeriod ;           //halfPeriod and period
    parameter v2Width = period - halfPeriod ;  //must be compile-time constants
...
endmodule
```

15.2 Method

The `method` statement is used to connect methods in a BSV interface to the appropriate Verilog wires. The syntax imitates a function prototype in that it doesn't define, but only declares. In the case of the `method` statement, instead of declaring types, it declares ports.

methodBVIStmt ::= `method` [*portId*] *identifer* [`(` [*portId* { `,` *portId* }] `)`]
[`enable` `(`*portId* `)`] [`ready` `(` *portId*`)`]
[`clocked_by` `(` *clockId*`)`] [`reset_by` `(` *resetId*`)`] `;`

The first *portId* is the output port for the method, and is only used when the method has a return value. The *identifier* is the method's name according to the BSV interface definition. The parenthesized list is the input port names corresponding to the method's arguments, if there are any. There may follow up to four optional clauses (in any order): `enable` (for the enable input port if the method has an `Action` component), `ready` (for the ready output port), `clocked_by` (to indicate the clock of the method, otherwise the default clock will be assumed) and `reset_by` (for the associated reset signal, otherwise the default reset will be assumed). If no `ready` port is given, the constant value 1 is used meaning the method is always ready. The names `no_clock` and `no_reset` can be used in `clocked_by` and `reset_by` clauses indicating that there is no associated clock and no associated reset, respectively.

If the input port list is empty and none of the optional clauses are specified, the list and its parentheses may be omitted. If any of the optional clauses are specified, the empty list `()` must be shown. Example:

```
method CLOCKREADY_OUT clockready() clocked_by(clk);
```

If there was no `clocked_by` statement, the following would be allowed:

```
method CLOCKREADY_OUT clockready;
```

The BSV types of all the method's arguments and its result (if any) must all be in the `Bits` typeclass.

Any of the port names may have an attribute attached to them. The allowable attributes are `reg`, `const`, `unused`, and `inhigh`. The attributes are translated into port descriptions. Not all port attributes are allowed on all ports.

For the output ports, the ready port and the method return value, the properties `reg` and `const` are allowed. The `reg` attribute specifies that the value is coming directly from a register with no intermediate logic. The `const` attribute indicates that the value is hardwired to a constant value.

For the input ports, the input arguments and the enable port, `reg` and `unused` are allowed. In this context `reg` specifies that the value is immediately written to a register without intermediate logic. The attribute `unused` indicates that the port is not used inside the module; its value is ignored.

Additionally, for the method enable, there is the `inhigh` property, which indicates that the method is `always_enabled`, as described in Section 14.2.2. Inside the module, the value of the enable is assumed to be 1 and, as a result, the port doesn't exist. The user still gives a name for the port as a placeholder. Note that only `Action` or `ActionValue` methods can have an enable signal.

The following code fragment shows an attribute on a method enable:

```
    method load(flopA, flopB) enable((*inhigh*) EN);
```

The output ports may be shared across methods (and ready signals).

15.3 Port

The `port` statement declares an input port, which is not part of a method, along with the value to be passed to the port. While parameters must be compile-time constants, ports can be dynamic. The `port` statements are analogous to arguments to a BSV module, but are rarely needed, since BSV style is to interact and pass arguments through methods.

portBVIStmt ::= `port` *identifier* [`clocked_by` `(` *clockId* `)`]
  [`reset_by` `(` *resetId* `)`] `=` *expression* `;`

The defining operator `<-` or `=` may be used.

The value of *expression* is supplied to the Verilog port named *identifier*. The type of *expression* must be in the `Bits` typeclass. The *expression* may be dynamic (e.g. the `_read` method of a register instantiated elsewhere in the module body), which differentiates it from a parameter statement. The *bsc* compiler cannot check that the import has specified the same size as declared in the Verilog module. If the width of the value is not the same as that expected by the Verilog module, Verilog will truncate or zero-extend the value to fit.

Example - Setting port widths to a specific width:

```
    // Tie off the test ports
       Bit#(1) v = 0 ;
       port TM = v   ;  // This ties off the port TM to a 1 bit wide 0
       Bit#(w) z = 0 ;
       port TD = z   ;  // This ties off the port TD to w bit wide 0
```

The `clocked_by` clause is used to specify the clock domain that the port is associated with, named by *clockId*. Any clock in the domain may be used. The values `no_clock` and `default_clock`, as described in Section 15.5, may be used. If the clause is omitted, the associated clock is the default clock.

Example - BVI import statement including port statements

```
       port BUS_ID clocked_by (clk2) = busId ;
```

The `reset_by` clause is used to specify the reset the port is associated with, named by *resetId*. Any reset in the domain may be used. The values `no_reset` and `default_reset`, as described in Section 15.8 may be used. If the clause is omitted, the associated reset is the default reset.

15.4 Input clock

The `input_clock` statement specifies how an incoming clock to a module is connected. Typically, there are two ports, the oscillator and the gate, though the connection may use fewer ports.

inputClockBVIStmt	::=	`input_clock` [*identifier*] `(` [*portsDef*] `)` `=` *expression* `;`
portsDef	::=	*portId* [`,` [*attributeInstances*] *portId*]
portId	::=	*identifier*

The defining operator `=` or `<-` may be used.

The *identifier* is the clock name which may be used elsewhere in the import to associate the clock with resets and methods via a `clocked_by` clause, as described in Sections 15.7 and 15.2. The *portsDef* statement describes the ports that define the clock. The clock value which is being connected is given by *expression*.

If the *expression* is an identifier being assigned with `=`, and the user wishes this to be the name of the clock, then the *identifier* of the clock can be omitted and the *expression* will be assumed to be the name. The clock name can be omitted in other circumstances, but then no name is associated with the clock. An unamed clock cannot be referred to elsewhere, such as in a method or reset or other statement. Example:

```
    input_clock (OSC, GATE) = clk;
```

is equivalent to:

```
    input_clock clk (OSC, GATE) = clk;
```

The user may leave off the gate (one port) or the gate and the oscillator (no ports). It is the designer's responsibility to ensure that not connecting ports does not lead to incorrect behavior. For example, if the Verilog module is purely combinational, there is no requirement to connect a clock, though there may still be a need to associate its methods with a clock to ensure that they are in the correct clock domain. In this case, the *portsDef* would be omitted. Example of an input clock without any connection to the Verilog ports:

```
     input_clock ddClk() = dClk;
```

If the clock port is specified and the gate port is to be unconnected, an attribute, either **`unused`** or **`inhigh`**, describing the gate port should be specified. The attribute **`unused`** indicates that the submodule doesn't care what the unconnected gate is, while **`inhigh`** specifies the gate is assumed in the module to be logical 1. It is an error if a clock with a gate that is not logical 1 is connected to an input clock with an **`inhigh`** attribute. The default when a gate port is not specified is **`inhigh`**, though it is recommended style that the designer specify the attribute explicitly.

To add an attribute, the usual attribute syntax, `(* attribute_name *)` immediately preceding the object of the attribute, is used. For example, if a Verilog module has no internal transitions and responds only to method calls, it might be unnecessary to connect the gating signal, as the implicit condition mechanism will ensure that no method is invoked if its clock is off. So the second *portId*, for the gate port, would be marked unused.

```
     input_clock ddClk (OSC, (*unused*) UNUSED) = dClk;
```

The options for specifying the clock ports in the *portsDef* clause are:

```
    ( )                     // there are no Verilog ports
    (OSC, GATE)             // both an oscillator port and a gate port are specified
    (OSC, (*unused*)GATE)  // there is no gate port and it's unused
    (OSC, (*inhigh*)GATE)  // there is no gate port and it's required to be logical 1
    (OSC)                   // same as (OSC, (*inhigh*) GATE)
```

In an `input_clock` statement, it is an error if both the port names and the input clock name are omitted, as the clock is then unusable.

15.5 Default clock

In BSV, each module has an implicit clock (the *current clock*) which is used to clock all instantiated submodules unless otherwise specified with a `clocked_by` clause. Other clocks to submodules must be explicitly passed as input arguments.

Every BVI import module must declare which input clock (if any) is the default clock. This default clock is the implicit clock provided by the parent module, or explicitly given via a `clocked_by` clause. The default clock is also the clock associated with methods and resets in the BVI import when no `clocked_by` clause is specified.

The simplest definition for the default clock is:

defaultClockBVIStmt ::= `default_clock` *identifier* `;`

where the *identifier* specifies the name of an input clock which is designated as the default clock.

The default clock may be unused or not connected to any ports, but it must still be declared. Example:

```
default_clock no_clock;
```

This statement indicates the implicit clock from the parent module is ignored (and not connected). Consequently, the default clock for methods and resets becomes `no_clock`, meaning there is no associated clock.

To save typing, you can merge the `default_clock` and `input_clock` statements into a single line:

defaultClockBVIStmt ::= `default_clock` [*identifier*] [`(` *portsDef* `)`] [`=` *expression*] `;`

The defining operator `=` or `<-` may be used.

This is precisely equivalent to defining an input clock and then declaring that clock to be the default clock. Example:

```
default_clock clk_src (OSC, GATE) = sClkIn;
```

is equivalent to:

```
input_clock clk_src (OSC, GATE) = sClkIn;
default_clock clk_src;
```

If omitted, the `=` *expression* in the `default_clock` statement defaults to `<- exposeCurrentClock`. Example:

```
default_clock xclk (OSC, GATE);
```

is equivalent to:

```
default_clock xclk (OSC, GATE) <- exposeCurrentClock;
```

If the portnames are excluded, the names default to `CLK, CLK_GATE`. Example:

```
default_clock xclk = clk;
```

is equivalent to:

```
default_clock xclk (CLK, CLK_GATE) = clk;
```

Alternately, if the *expression* is an identifier being assigned with `=`, and the user wishes this to be the name of the default clock, then he can leave off the name of the default clock and *expression* will be assumed to be the name. Example:

```
default_clock (OSC, GATE) = clk;
```

is equivalent to:

```
default_clock clk (OSC, GATE) = clk;
```

If an expression is provided, both the ports and the name cannot be omitted.

However, omitting the entire statement is equivalent to:

```
default_clock (CLK, CLK_GATE) <- exposeCurrentClock;
```

specifying that the current clock is to be associated with all methods which do not specify otherwise.

15.6 Output clock

The `output_clock` statement gives the port connections for a clock provided in the module's interface.

outputClockBVIStmt ::= `output_clock` *identifier* `(` [*portsDef*] `)` `;`

The *identifier* defines the name of the output clock, which must match a clock declared in the module's interface. Example:

```
interface ClockGenIfc;
  interface Clock gen_clk;
endinterface

import "BVI" ClockGen =
module vMkAbsoluteClock #( Integer start,
                      Integer period
                     ) ( ClockGenIfc );
   ...
   output_clock gen_clk(CLK_OUT);
endmodule
```

It is an error for the same *identifier* to be declared by more than one `output_clock` statement.

15.7 Input reset

The `input_reset` statement defines how an incoming reset to the module is connected. Typically there is one port. BSV assumes that the reset is inverted (the reset is asserted with the value 0).

inputResetBVIStmt	::=	`input_reset` [*identifier*] [`(` *portId* `)`] [`clocked_by` `(` *clockId* `)`] `=` *expression* `;`
portId	::=	*identifier*
clockId	::=	*identifier*

where the `=` may be replaced by `<-`.

The reset given by *expression* is to be connected to the Verilog port specified by *portId*. The *identifier* is the name of the reset and may be used elsewhere in the import to associate the reset with methods via a `reset_by` clause.

The `clocked_by` clause is used to specify the clock domain that the reset is associated with, named by *clockId*. Any clock in the domain may be used. If the clause is omitted, the associated clock is the default clock. Example:

```
input_reset rst(sRST_N) = sRstIn;
```

is equivalent to:

```
input_reset rst(sRST_N) clocked_by(clk) = sRstIn;
```

where `clk` is the identifier named in the `default_clock` statement.

If the user doesn't care which clock domain is associated with the reset, `no_clock` may be used. In this case the compiler will not check that the connected reset is associated with the correct domain. Example

```
input_reset rst(sRST_N) clocked_by(no_clock) = sRstIn;
```

If the *expression* is an identifier being assigned with `=`, and the user wishes this to be the name of the reset, then he can leave off the *identifier* of the reset and the *expression* will be assumed to be the name. The reset name can be left off in other circumstances, but then no name is associated with the reset. An unamed reset cannot be referred to elsewhere, such as in a method or other statement.

In the cases where a parent module needs to associate a reset with methods, but the reset is not used internally, the statement may contain a name, but not specify a port. In this case, there is no port expected in the Verilog module. Example:

```
input_reset rst() clocked_by (clk_src) = sRstIn ;
```

Example of a BVI import statement containing an `input_reset` statement:

```
import "BVI" SyncReset =
module vSyncReset#(Integer stages ) ( Reset rstIn, ResetGenIfc rstOut ) ;
   ...
   // we don't care what the clock is of the input reset
   input_reset rst(IN_RST_N) clocked_by (no_clock) = rstIn ;
   ...
endmodule
```

15.8 Default reset

In BSV, when you define a module, it has an implicit reset (the *current reset*) which is used to reset all instantiated submodules (unless otherwise specifed via a `reset_by` clause). Other resets to submodules must be explicitly passed as input arguments.

Every BVI import module must declare which reset, if any, is the default reset. The default reset is the implicit reset provided by the parent module (or explicitly given with a `reset_by`). The default reset is also the reset associated with methods in the BVI import when no `reset_by` clause is specified.

The simplest definition for the default reset is:

defaultResetBVIStmt ::= `default_reset` *identifier* `;`

where *identifier* specifies the name of an input reset which is designated as the default reset.

The reset may be unused or not connected to a port, but it must still be declared. Example:

```
     default_reset no_reset;
```

The keyword `default_reset` may be omitted when declaring an unused reset. The above statement can thus be written as:

```
     no_reset;        // the default_reset keyword can be omitted
```

This statement declares that the implicit reset from the parent module is ignored (and not connected). In this case, the default reset for methods becomes `no_reset`, meaning there is no associated reset.

To save typing, you can merge the `default_reset` and `input_reset` statements into a single line:

defaultResetBVIStmt ::= `default_reset` [*identifier*] [`(` *portId* `)`] [`clocked_by` `(` *clockId* `)`]
[`=` *expression*] `;`

The defining operator `=` or `<-` may be used.

This is precisely equivalent to defining an input reset and then declaring that reset to be the default. Example:

```
    default_reset rst (RST_N) clocked_by (clk) = sRstIn;
```

is equivalent to:

```
    input_reset rst (RST_N) clocked_by (clk) = sRstIn;
    default_reset rst;
```

If omitted, = *expression* in the `default_reset` statement defaults to `<- exposeCurrentReset`. Example:

```
    default_reset rst (RST_N);
```

is equivalent to

```
    default_reset rst (RST_N) <- exposeCurrentReset;
```

The `clocked_by` clause is optional; if omitted, the reset is clocked by the default clock. Example:

```
default_reset rst (sRST_N) = sRstIn;
```

is equivalent to

```
default_reset rst (sRST_N) clocked_by(clk) = sRstIn;
```

where `clk` is the `default_clock`.

If `no_clock` is specified, the reset is not associated with any clock. Example:

```
input_reset rst (sRST_N) clocked_by(no_clock) = sRstIn;
```

If the *portId* is excluded, the reset port name defaults to `RST_N`. Example:

```
default_reset rstIn = rst;
```

is equivalent to:

```
default_reset rstIn (RST_N) = rst;
```

Alternatively, if the *expression* is an identifier being assigned with `=`, and the user wishes this to be the name of the default reset, then he can leave off the name of the default reset and *expression* will be assumed to be the name. Example:

```
default_reset (rstIn) = rst;
```

is equivalent to:

```
default_reset rst (rstIn) = rst;
```

Both the ports and the name cannot be omitted.

However, omitting the entire statement is equivalent to:

```
default_reset (RST_N) <- exposeCurrentReset;
```

specifying that the current reset is to be associated with all methods which do not specify otherwise.

15.9 Output reset

The `output_reset` statement gives the port connections for a reset provided in the module's interface.

outputResetBVIStmt ::= `output_reset` *identifier* [`(` *portId* `)`] [`clocked_by` `(` *clockId* `)`]`;`

The *identifier* defines the name of the output reset, which must match a reset declared in the module's interface. Example:

```
interface ResetGenIfc;
  interface Reset gen_rst;
endinterface

import "BVI" SyncReset =
module vSyncReset#(Integer stages ) ( Reset rstIn, ResetGenIfc rstOut ) ;
   ...
   output_reset gen_rst(OUT_RST_N) clocked_by(clk) ;
endmodule
```

It is an error for the same *identifier* to be declared by more than one `output_reset` statement.

15.10 Ancestor, same family

There are two statements for specifying the relationship between clocks: `ancestor` and `same_family`.

ancestorBVIStmt ::= `ancestor` `(` *clockId* `,` *clockId* `)` `;`

This statement indicates that the second named clock is an `ancestor` of the first named clock. To say that `clock1` is an `ancestor` of `clock2`, means that `clock2` is a gated version of `clock1`. This is written as:

```
ancestor (clock2, clock1);
```

For clocks which do not have an ancestor relationship, but do share a common ancestor, we have:

sameFamilyBVIStmt ::= `same_family` `(` *clockId* `,` *clockId* `)` `;`

This statement indicates that the clocks specified by the *clockIds* are in the same family (same clock domain). When two clocks are in the same family, they have the same oscillator with a different gate. To be in the same family, one does not have to be a gated version of the other, instead they may be gated versions of a common ancestor. Note that `ancestor` implies `same_family`, which then need not be explicitly stated. For example, a module which gates an input clock:

```
input_clock clk_in(CLK_IN, CLK_GATE_IN) = clk_in ;
output_clock new_clk(CLK_OUT, CLK_GATE_OUT);
ancestor(new_clk, clk_in);
```

15.11 Schedule

scheduleBVIStmt ::= `schedule` `(` *identifier* { `,` *identifier* } `)` *operatorId*
    `(` *identifier* { `,` *identifier* } `)``;`

operatorId ::= `CF`
  | `SB`
  | `SBR`
  | `C`

The `schedule` statement specifies the scheduling constraints between methods in an imported module. The operators relate two sets of methods; the specified relation is understood to hold for each pair of an element of the first set and an element of the second set. The order of the methods in the lists is unimportant and the parentheses may be omitted if there is only one name in the set.

The meanings of the operators are:

`CF`	conflict-free
`SB`	sequences before
`SBR`	sequences before, with range conflict (that is, not composable in parallel)
`C`	conflicts

It is an error to specify two relationships for the same pair of methods. It is an error to specify a scheduling annotation other than `CF` for methods clocked by unrelated clocks. For such methods, `CF` is the default; for methods clocked by related clocks the default is `C`. The compiler generates a warning if an annotation between a method pair is missing. Example:

```
import "BVI" FIFO2 =
module vFIFOF2_MC                   (  Clock sClkIn, Reset sRstIn,
                                   Clock dClkIn, Reset dRstIn,
                                   Clock realClock,  Reset realReset,
                                   FIFOF_MC#(a) ifc )
                                provisos (Bits#(a,sa));
    ...
    method          enq( D_IN ) enable(ENQ) clocked_by( clk_src ) reset_by( srst ) ;
    method FULL_N   notFull                 clocked_by( clk_src ) reset_by( srst ) ;

    method          deq()       enable(DEQ) clocked_by( clk_dst ) reset_by( drst ) ;
    method D_OUT    first                   clocked_by( clk_dst ) reset_by( drst ) ;
    method EMPTY_N  notEmpty                clocked_by( clk_dst ) reset_by( drst ) ;

    schedule (enq, notFull) CF (deq, first, notEmpty) ;
    schedule (first, notEmpty) CF (first, notEmpty) ;
    schedule (notFull) CF (notFull) ;
    // CF: conflict free - methods in the first list can be scheduled
    // in any order or any number of times,  with the methods in the
    // second list - there is no conflict between the methods.
    schedule first SB deq ;
    schedule (notEmpty) SB (deq) ;
    schedule (notFull) SB (enq) ;
    // SB indicates the order in which the methods must be scheduled
    // the methods in the first list must occur (be scheduled) before
    // the methods in the second list
    // SB allows these methods to be called from one rule but the
    // SBR relationship does not.
    schedule (enq) C (enq) ;
    schedule (deq) C (deq) ;
    // C: conflicts - methods in the first list conflict with the
    // methods in the second - they cannot be called in the same clock cycle.
    // if a method conflicts with itself, (enq and deq), it
    // cannot be called more than once in a clock cycle
  endmodule
```

15.12 Path

The `path` statement indicates that there is a combinational path from the first port to the second port.

pathBVIStmt ::= `path` (*portId* , *portId*) ;

It is an error to specify a path between ports that are connected to methods clocked by unrelated clocks. This would be, by definition, an unsafe clock domain crossing. Note that the compiler assumes that there will be a path from a value or `ActionValue` method's input parameters to its result, so this need not be specified explicitly.

The paths defined by the `path` statement are used in scheduling. A path may impact rule urgency by implying an order in how the methods are scheduled. The path is also used in checking for combinational cycles in a design. The compiler will report an error if it detects a cycle in a design. In the following example, there is a path declared between `WSET` and `WHAS`, as shown in figure 9.

```
    import "BVI" RWire0 =
       module vMkRWire0 (VRWire0);
          ...
          method wset() enable(WSET) ;
```

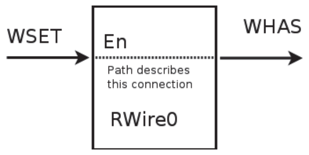

Figure 9: Path in the RWire0 Verilog module between WSET and WHAS ports

```
      method WHAS whas ;
      schedule whas CF whas ;
      schedule wset SB whas ;
      path (WSET, WHAS) ;
   endmodule: vMkRWire0
```

15.13 Interface

interfaceBVIStmt ::= `interface` *typeDefType* `;`
{ *interfaceBVIMembDecl* }
`endinterface` [`:` *typeIde*]

interfaceBVIMembDecl ::= *methodBVIStmt*
| *interfaceBVIStmt* `;`

An interface statement can contain two types of statements: method statements and subinterface declarations. The interface statement in BVI import is the same as any other interface statement (Section 5.2) with one difference: the method statements within the interface are BVI method statements (`methodBVIStmt` 15.2).

Example:

```
import "BVI" BRAM2 =
module vSyncBRAM2#(Integer memSize, Bool hasOutputRegister,
                   Clock clkA, Reset rstNA, Clock clkB, Reset rstNB
                    ) (BRAM_DUAL_PORT#(addr, data))
   provisos(Bits#(addr, addr_sz),
            Bits#(data, data_sz));
   ...

   interface BRAM_PORT a;
     method put(WEA, (*reg*)ADDRA, (*reg*)DIA) enable(ENA) clocked_by(clkA) reset_by(rstA);
     method DOA read() clocked_by(clkA) reset_by(rstA);
   endinterface: a

   interface BRAM_PORT b;
     method put(WEB, (*reg*)ADDRB, (*reg*)DIB) enable(ENB) clocked_by(clkB) reset_by(rstB);
     method DOB read() clocked_by(clkB) reset_by(rstB);
   endinterface: b
endmodule: vSyncBRAM2
```

Since a BVI wrapper module can only provide a single interface (`BRAM_DUAL_PORT` in this example), to provide multiple interfaces you have to create an interface hierarchy using interface statements.

The interface hierarchy provided in this example is:

```
interface BRAM_DUAL_PORT#(type addr, type data);
   interface BRAM_PORT#(addr, data) a;
   interface BRAM_PORT#(addr, data) b;
endinterface: BRAM_DUAL_PORT
```

where the subinterfaces, `a` and `b`, are defined as `interface` statements in the body of the `import "BVI"` statement.

15.14 Inout

The following statements describe how to pass an `inout` port from a wrapped Verilog module through a BSV module. These ports are represented in BSV by the type `Inout`. There are two ways that an `Inout` can appear in BSV modules: as an argument to the module or as a subinterface of the interface provided by the module. There are, therefore, two ways to declare an `Inout` port in a BVI import: the statement `inout` declares an argument of the current module; and the statement `ifc_inout` declares a subinterface of the provided interface.

inoutBVIStmt ::= `inout` *portId* [`clocked_by (` *clockId* `)`]
  [`reset_by (` *resetId* `)`] `=` *expression* `;`

The value of *portId* is the Verilog name of the `inout` port and *expression* is the name of an argument from the module.

inoutBVIStmt ::= `ifc_inout` *identifier* `(`*inoutId* `)` [`clocked_by (` *clockId* `)`]
  [`reset_by (` *resetId* `)`] `;`

Here, the *identifier* is the name of the subinterface of the provided interface and *portId* is, again, the Verilog name of the `inout` port.

The clock and reset associated with the `Inout` are assumed to be the default clock and default reset unless explicitly specified.

Example:

```
interface Q;
   interface Inout#(Bit#(13)) q_inout;
   interface Clock c_clock;
endinterface

import "BVI" Foo =
module mkFoo#(Bool b)(Inout#(int) x, Q ifc);
   default_clock ();
   no_reset;

   inout iport = x;

   ifc_inout q_inout(qport);
   output_clock c_clock(clockport);
endmodule
```

The wrapped Verilog module is:

```
module Foo (iport, clockport, qport);
   input cccport;
   inout [31:0] iport;
   inout [12:0] qport;
   ...
endmodule
```