A Guide to PERLILOG

Eli Billauer

`eli.billauer@gmail.com`

May 1, 2016

This guide applies to version 1.0 of PERLILOG .

Contents

1 Introduction

1.1 PERLILOG in a nutshell

1.1.1 What is PERLILOG ?

PERLILOG is a design tool, whose main target is easy integration of Verilog IP cores.

The philosophy behind Perilog is that an IP core should be like a black box. Fitting it for a certain purpose should be as easy as defining the desired requirements. Connecting the cores, to become a system, should be as easy as drawing a block diagram.

With plain Verilog, the reality couldn't be further away. But by using PERLILOG correctly, integration can be that simple.

PERLILOG introduces a new meaning to "IP core". It also introduces a different way to approaching the task of interfacing cores with each other.

PERLILOG was built to make core programming and integration intuitive tasks. As such, it is based on new, rather natural concepts, which one must get used to in order to gain the most of the tool.

PERLILOG is written in Perl, currently with no GUI. While the scripts, that the system consists of, are rather sophisticated, only plain Perl knowledge is needed to use its scripting capabilities.

1.1.2 Bye bye Verilog?

Absolutely not. Verilog is still the language to define the functionality of the core. The final output of a design, which incorporates PERLILOG , is perfectly normal Verilog files.

PERLILOG will do the following tasks instead of you:

- Instantiation of modules

- Connection between modules

- Setting up modules' attributes (word width, address mapping on buses etc.)

1.1.3   Other usability

Aside from its main purpose, PERLILOG can be used an excellent starting point for writing various scripts, that manipulate Verilog modules.

1.2   What's PERLILOG ? A broader look

We'll begin with what it's not: It's not a new language, and it doesn't come instead of either Perl or Verilog.

A Perl programmer will come up with the quickest answer: It's a Perl module. A Verilog HDL programmer may consider it as a scripting extension of good old Verilog.

And the truth lies somewhere in the middle: PERLILOG is a mixture, both in syntax and in spirit, of these two languages.

The interesting part is the mixture in spirits. Perl programming is natural and intuitive, while Verilog can be very technical. The main idea behind PERLILOG , is to free the HDL programmer from those moments, in which the human functions like a machine. One of the concepts that was highly adopted, was that an environment is made for humans and not for computers. This means that the language should be closer to how humans see things, than how they are actually working in the computer.

Using PERLILOG does not mean leaving the Verilog tools you're familiar with, or throwing away your Verilog coding skills. On the contrary. PERLILOG is something you run on a bunch of files you've written, some of which are very similar to plain Verilog, and get a set of Verilog files to feed your Verilog simulator or synthesizer with. If you want, it can be seen as a compiler from some loosely defined language to strict Verilog.

Unlike Perl, HDL programming is targeted towards efficiency, which can't be compromised in any HDL environment. Neither is there any tolerance for bugs in ASIC designs. PERLILOG was designed with thought of helping the human to avoid both lack of efficiency and buggy code. Much of this help comes from dismissing the programmer from performing boring tasks. A bored programmer is a dangerous programmer.

But like Perl, the only way to understand why it's different, is to learn it. The plain Perl language looks ugly and obscure until you get the hang of it – there is no way to know how powerful it is without using it. PERLILOG can't really be explained in a few words either. Like Perl, its strength lies in its spirit.

1.3   How PERLILOG is used

Figure 1 depicts the data flow using PERLILOG . To begin with, we note that the final target of the process is plain Verilog files, which can be fed into the well-known simulators or synthesizers with no hassle at all.

These target Verilog files are a complete system, which comprises of several IP cores. What PERLILOG did, was to glue a group of cores (or just Verilog modules) into this system. This effort involved connecting the cores, and setting them up to work together.

The actual execution of a PERLILOG process, is running a Perl script, which uses the PERLILOG module (by literally saying `use Perlilog`). This main script is in general straightforward to write. Its function is to describe how the cores are related to each other, and set up their attributes.

The cores can be given to PERLILOG in several ways, which can be divided into two forms: Writing Verilog-like files, or writing small Perl scripts.

Writing Verilog-like files includes the possibility to use existing Verilog code without needing to change a single line of it, but there are several advantages in making slight adaptions to PERLILOG . Once the principles are known, this is a quick and rewarding process.

Creating a core as a Perl script is preferred when the we want the Verilog code to be primarily created by a Perl script. This is an advanced feature, which we discuss later on.

All in all, PERLILOG is primarily an integration tool. It is intended to make reuse and integration of cores as simple as drawing the targeted system's block diagram.
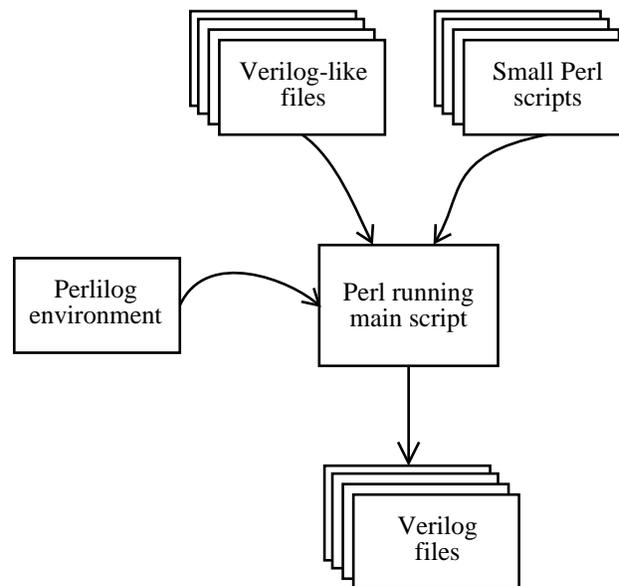
Figure 1: Data flow with PERLILOG

1.4   The key features of PERLILOG

This is an attempt to summarize some goodies about PERLILOG :

- Easy connection of IP cores: Instead of connecting the cores in Verilog instantiation, wire by wire, PERLILOG allows to group wires and classify this group. This means not only that possible confusion of which wire goes to which is avoided, but also that misunderstandings of the wires' definitions are spared.

- Reusability of cores: Many cores, especially those who fit the category of "peripherals" can be written independently of the bus that they are going to be mapped to.

- The interfacing between cores, even if it's complicated, can be done automatically by prewritten and well-tested scripts. Bus controllers can be upgraded and configured separately, by simple means.

- Teamwork: PERLILOG IP cores are easy to encapsulate for trivial use by others. In fact, highly skilled programmers may create cores, that less advanced personell may integrate into a system.

- Easy configurability: IP cores fit in as objects in an object-oriented environment, and can be configured by virtue of the object's properties. This, together with simple Perl scripting, allows the IP core to be configured by attributes that humans understand. Since Perl scripts interpret the properties, sanity checks are easy to perform. In addition, one single property may affect several aspects of the Verilog code, in ways that are very difficult to achieve with Verilog compiler directives.

- Configuration files for simulators and synthesizers, as well as reports are easy to generate in PERLILOG .

- The environment is easily customized to meet specific needs. It's easy to add extra scripts to make slight changes in an already functioning design.

- Once the system is adapted by some design group, and is up and running for a while, it's easy for someone new to get started with it (a new employee, for example). No need to be an expert to use the system.

- PERLILOG is GUI-ready in the sense that a GUI-based application will interface naturally with PERLILOG , and that the concept of the environment are easily depicted in graphics.

The word "easy" appears a lot of times, and after trying to do a few things with PERLILOG I stand behind every one of these.

The real difficulty is to get used to the concepts that makes it all this easy. So getting started is a difficult, but yet rewarding process.

1.5   This guide's outline

This is a guide, not a reference manual.

Meaning, that the information is not organized by topic, but in an order which is sensible to learn by. Still, the sections are heavily cross-referenced to allow a quick jump to related issues, so it should be fairly easy to find specific information.

Sections 2 and 3 teach how to use PERLILOG in its simplest way – with template files. This is what you need to know if you know Verilog and some basic Perl, and you'd like to use PERLILOG to connect modules.

This section will show examples, and the next one completes the examples with some insight about what really happened. Reading and understanding the examples will probably be enough to try a few things of your own.

Section 4 is short, and gives a few tips for using the environment a bit better.

Next, section 5 explains how objects are used in PERLILOG . This is another stage in understanding what actually happens in the system, and it opens some additional possibilities.

And here comes the turnpoint. Sections 6 and 7 teach how to enrich PERLILOG . This means knowing how to write new classes (which generate objects) and then how to write classes that connect modules in interesting ways. It does not require to read the source of PERLILOG or anything like that, but a good feeling of the ideas, on which it is based, is nothing you can do without at this stage. Once the ideas are clear, writing classes is really easy.

After that, we have several sections which are written as an API reference but unlike the classic way of writing references, there are plently of usage examples there too, so it's possible to understand something from that part as well.

At the time of writing this, PERLILOG has very few classes, which makes it far less useful than it can be. My hope is that others (you?) will enrich the system with some real-life classes, which will make it a powerful tool as it can be.

2   A jump into the water

2.1   A jump into what?

PERLILOG is very Perlish. That means, among others, that if you read some code written PERLILOG , you'll probably get a good idea about what the whole thing is about.

This is the reason why we begin with several examples. Actually, trying to explain the ideas behind PERLILOG without seeing examples first would make this guide look like an academic paper. And be as hard to read.

The examples do not respresent the abilities of PERLILOG , but rather demonstrate a few concepts. They should be viewed merely as a slow drive in the parking lot, but they are nevertheless a crucial element of this guide.

Slow drive or not, this is what the examples are going to show:

- A connection of several wires in one chunk

- Wishbone SoC bus master and slave connection

- Connecting two Wishbone bus slaves with one master, without needing to write the bus controller

- Connecting a module, with no bus interface logic, to a Wishbone bus

- A module with embedded Perl script, which writes parts of the Verilog

But the examples are not enough. Following conventions is very important in PERLILOG , because it's an environment where many people's code melt together. Writing code that works is nice, but if wasn't written according to some rules it will soon make trouble.

2.2   Installing and running PERLILOG

2.2.1   Automatic installation

PERLILOG was written under ActivePerl 5.6.1, but should work properly on any Perl version above 5.004.

PERLILOG can be installed as any CPAN Perl module. Simply uncompress `perlilog-xx.tar.gz` into a local directory, and perform the following commands at prompt:

```
perl Makefile.PL
make
make install
```

Note that you must be superuser (root) to perform `make install`.

2.2.2 Manual installation

There are several cases in which the automatic installation is not possible. The common ones are lack of GNU make (on non-UNIX systems) or lack of enough priviliges on the computer to install the files. Since the automatic installation procedure is so simple, it's always recommended to try it before installing manually.

Manual installation is performed by copying the files and directories in the `lib` subdirectory to any directory, but preferably to one of the directories that appear in `@INC`, as can be shown with

```
perl -e "print join ',', @INC"
```

Note, that in any case, the `lib` directory itself is not copied, but its content.

If the files are installed in one of the `@INC` directories, the command line for running Perl will be shorter, and the following guidelines are recommended:

- A directory path, in which the word `site` appears somehow, is usually preferred, just to keep the files organized in your system.

- If the computer has multiple users, the files should be owned by the superuser or the administrator, and should be read-only to regular users (this is a regular security measure).

After putting the files in some directory, the following command can be used to run scripts (may depend on your Perl flavor). We've taken `try1.pl` as the example of the script to run:

- `perl -w try1.pl` if the installation was done into one of the directories mentioned in `@INC`.

- `perl -w -IC:\perlilog try1.pl` on Windows-based systems, assuming that `Perlilog.pm` and the subdirectories were put in `C:\perlilog\`.

- `perl -w -I/home/myself/perlilog try1.pl` on UNIX/Linux systems, assuming that `Perlilog.pm` and the subdirectories were put in `/home/myself/perlilog/`.

The `-w` flag is recommended, since it makes Perl supply warnings if something looks bad.

2.3 First example: A simple connection

2.3.1 The template files

We now see the first example of a trivial connection between two modules. There are three files involved, which can be found in the `examples/ex1` directory.

The first file, `myff.pt` is:

```
port ffport vars clk:clock, rst:reset, d:data, q:register;

module myff_template(clock, reset, data, register);
   input  clock, reset, data;
   output register;

   reg    register;

   always @(posedge clock or posedge reset)
      if (reset)
         register <= #1 0;
      else
         register <= #1 data;
endmodule
```

The `.pt` extension stands for "PERLILOG template".

Except for the first line, this is no more than a Verilog implementation of a D-flipflop. The port declaration in the first line will be discussed later on. We now look on the second file, `mytest.pt`:

```
port testport vars clk:clk, rst:rst, d:d, q:q;

module mytest_template(clk, rst, d, q);
   input   q;
   output  clk, rst, d;
   reg     clk, rst;


   always #5 clk = ~clk;

   assign d = ~q;

   initial
```

```
            begin
                clk = 0; rst = 0;

                #1; rst = 1;
                #1; rst = 0;

                repeat (100) @(posedge clk);
                $stop;
            end
        endmodule
```

Again, we have something familiar, except for the first line. This time it's a basic stimulus generator (some will call this a test bench).

2.3.2 The Perl script

We now view a Perl script, which connects these two modules. It's in `try1.pl`:

```
use Perlilog;
init;

$top = verilog->new(name => 'top');

$ff = template->new(name => 'flipflop',
                    tfile => 'myff.pt',
                    parent => $top);

$test = template->new(name => 'test',
                      tfile => 'mytest.pt',
                      parent => $top);

interface($ff->getport('ffport'), $test->getport('testport'));

execute;

silos->new(name => 'silos_configfile_creator')->makesilosfile;
```

The script begins with `use Perlilog` which activates the PERLILOG module. The `init` command means that we've finished extra configurations (none), and the system may initialize.

Next we create an object of class `verilog`, give it the name `top`, and put its handle in `$top`. As we shall see later on, this will result in a Verilog module, named `top` and put in a file named `top.v`.

As a matter of fact, the `name => 'top'` in the declaration actually means that the `name` property should have the string value `top`. In general, `=>` pairs mean initial values of properties when we call the `new()` method.

Now we create two other objects, both of class `template`, which is a class derived from `verilog`. The first class is given the name `flipflop`, and is assigned the template file `myff.pt`, which we saw earlier. This will end up with a Verilog module named `flipflop` in `flipflop.v`, which will look very much like `myff.pt`.

In quite a similar way, will we have a module named `test`, which will look like `mytest.pt`.

Both of these objects have the `parent` property set to `$top` which is the handle of the first object we generated. This will result in both of these object being instantiated in the first object. Figure 2 depicts the relation between the objects.

The next line, which is the `interface()` call is where we connect between these two modules. It will be explained in section 2.3.4.

The `execute` command that follows kicks off the generation of Verilog code, along with almost any other task that needs to be performed. In fact, all we did until this point was to prepare the environment for execution. It's only when we reached this point, that something really happens.

Even so, we're left with a small post-execution task, which is done in the last line of the script: We generate another object, of class `silos`, and call its `makesilosfile()` method. This will create a project file for Silos III, which makes it easy to use this simulator. More about this in section 2.3.6.

2.3.3   The output Verilog files

As a result of running the Perl script above, a new directory, `PLverilog` is generated (if it existed already, it was deleted along with anything in it – beware).

Three Verilog files are generated, and one Silos `.prj` file.

We shall now view each of them. We start with `flipflop.v`:
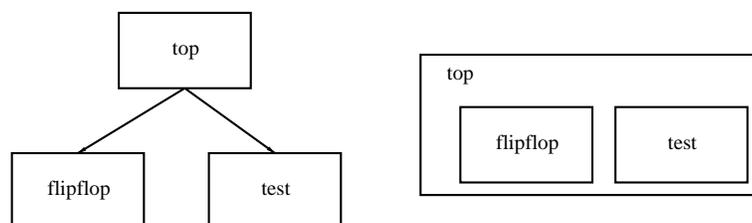
Figure 2: Instantiation tree (left) and module illustration (right)

```
'timescale 1ns / 10ps

module flipflop(clock, reset, data, register);

  input   clock;
  input   reset;
  input   data;
  output  register;
  reg   register;

   always @(posedge clock or posedge reset)
      if (reset)
         register <= #1 0;
      else
         register <= #1 data;

endmodule
```

We see that the module has changed name, and that there are some cosmetical
changes in the declaration part of the module. Nothing interesting here. Same thing
with test.v:

```
'timescale 1ns / 10ps

module test(q, clk, rst, d);

  input   q;
  output  clk;
  output  rst;
  output  d;
  reg   clk;
  reg   rst;

   always #5 clk = ~clk;

   assign d = ~q;

   initial
      begin
         clk = 0; rst = 0;

         #1; rst = 1;
         #1; rst = 0;

         repeat (100) @(posedge clk);
         $stop;
```

```
        end

endmodule
```

So let's have a look on the on `top.v`:

```
'timescale 1ns / 10ps

module top;

  wire  clk_via;
  wire  rst_via;
  wire  d_via;
  wire  register_via;

  flipflop  flipflop_ins(.clock(clk_via), .reset(rst_via), .data(d_via),
    .register(register_via));

  test  test_ins(.q(register_via), .clk(clk_via), .rst(rst_via), .d(d_via));

endmodule
```

This module is interesting for the simple reason that it was generated automatically. The two modules were instantiated, and four Verilog wires were created to connect them. Surprisingly enough, the connection between the modules makes sense. We now explain how PERLILOG knew how to connect these modules.

2.3.4 Ports and interface

We now go back to the template files, and look on the first line of each. In `myff.pt` we had:

```
port ffport vars clk:clock, rst:reset, d:data, q:register;
```

This line declared a PERLILOG *port*. It is given the name `ffport` and is of type `vars`. Then we have a list of pairings between *labels* and their respective variables.

The best way to understand ports is to imagine them as electrical connectors, that you plug into your computer, for example. The port's type is like the physical format of the connector, and that tells you what kind of adaptor you need to connect with another connector.

Each of the Verilog wires (or registers) are labeled with a name, which is like we put a label on each of the wires of the connector, describing what function the wire has. We disregard the fact that a Verilog wire or register can have more than one bit – it's a wire to us.

In the port above, we have four labels: `clk`, `rst`, `d` and `q`, each of them is associated with a Verilog wire or register in the current Verilog module.

Finally, the name (`ffport`) is just the identifier of the entire port (connector?) so we can relate to it in the Perl script.

Now to the port declaration in `mytest.pt`:

```
port testport vars clk:clk, rst:rst, d:d, q:q;
```

This is very similar to the one in `myff.pt`, only the port's name is different, and the names of Verilog wires and registers are different (they relate to the Verilog module of `mytest.pt`).

Note that the labels themselves are exactly the same on the two ports. This is how PERLILOG knows what to connect with what.

Generally, the `vars` type of port is the simplest kind. It's used to take some wires and registers, view them as a group, and give each an arbitrarily named label. When two `vars` ports are interfaced, wires and registers with the same label are connected. We shall see later on how interconnection can be made much smarter.

We now get to the line we didn't explain in the Perl script,

```
interface($ff->getport('ffport'), $test->getport('testport'));
```

The expression `$ff -> getport('ffport')` is a request from the flip-flop object (whose handle is kept in `$ff`) to give us the handle of its port, named `ffport`. Doing the same with `testport`, we have two handles of ports, which we pass to the `interface()` routine.

This is actually the point where we ask the system to "connect" the two ports, or if you like, to get the plug and jacket together.

Note that we can interface any ports from any modules, regardless of where they are instantiated.

2.3.5 "via" wires

The modules `flipflop` and `test` are not directly connected, but are both instantiated in the `top` module. But as a result of the connection of `ffport` with `testport`, wires need to be drawn between the two modules. This can only be done by creating extra wires in `top`. These are called *via* wires, or *vias*.

In this case, PERLILOG generated `clk_via`, `rst_via`, `d_via` and `register_via` in `top.v`.

The `_via` suffix marks the wires as vias. If better control of the names is desired, the `viasource` property can be set. See section 11.2.6.

2.3.6 The Silos project file output

We finish this example with showing the Silos project file, `perlilog.spj`:

```
[Files]
0=top.v
1=flipflop.v
2=test.v

[Settings]
Mode=Debug
```

The nice thing about this file is that when we give it to the Silos III simulator, we're ready to run a simulation right away. In other words, the PERLILOG environment prepared a file with all the Verilog files listed in it, so the human user doesn't need to bother which Verilog files were generated.

This demonstrates the possibility to generate configuration files and scripts to go along with the Verilog files automatically. The Perl language is highly suitable for this kind of task, and the relevant data is easy to access within the PERLILOG environment.

2.3.7 Two files instead of three

A small variant of the previous example, is to instantiate the flip-flop module in the test bench module, rather than having them both instantiated in a common top-level module.

To do this, we only change the Perl script slightly. We don't generate a `top` object, and we choose the test object as the flip-flop object's parent (see figure 3). The Perl file for this, `try2.pl`, is:

```
use Perlilog;
init;
```
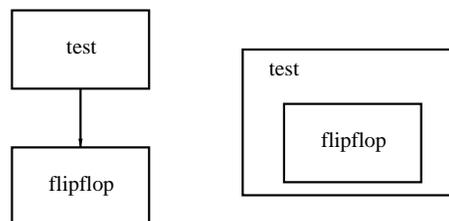

Figure 3: Instantiation tree (left) and module illustration (right)

```
$test = template->new(name => 'test',
                      tfile => 'mytest.pt');

$ff = template->new(name => 'flipflop',
                    tfile => 'myff.pt',
                    parent => $test);

interface($ff->getport('ffport'), $test->getport('testport'));

execute;

silos->new(name => 'silos_configfile_creator')->makesilosfile;
```

Note that we also changed the order in which the two object are created, so when the flip-flop object is generated, we have `$test` as a handle to use as the parent.

Running this script, `flipflop.v` remains the same, but we don't get a `top.v` file, and `test.v` will look as follows:

```
'timescale 1ns / 10ps

module test;

  reg  clk;
  reg  rst;
  wire q;
  wire d;

   always #5 clk = ~clk;

   assign d = ~q;

   initial
     begin
        clk = 0; rst = 0;

        #1; rst = 1;
        #1; rst = 0;

        repeat (100) @(posedge clk);
        $stop;
     end

  flipflop  flipflop_ins(.clock(clk), .reset(rst), .data(d),
                         .register(q));
```

```
endmodule
```

Which is the natural way to do it. Note that the input/output declarations that appear in `myff.pt` have vanished, and we have only registers and wires.

This variation shows how simple it is to move the place where a module is instantiated. It remains this simple even if the modules are placed in the most obscure manners: PERLILOG will draw vias as needed to fulfill the requested port connection.

2.4   Other examples: Wishbone connections

2.4.1   Purpose

We now move on to see how PERLILOG makes connecting Wishbone-compliant IP cores easily. The focus is not on the Wishbone standard itself, but rather on bus connection in general. Therefore, we are going to work with the simplest possible Wishbone interface standards, in order to minimize the amount of Wishbone-related details.

The Wishbone-related example files are kept in the `examples/ex2` directory.

2.4.2   A simple master-slave connection

Connecting a Wishbone master and slave involves, surprisingly enough, *three* elements: We have one module functioning as a master, one as a slave, and one module which supplies the clock and reset signals – both the master and the slave accept these signals as input, according to the Wishbone spec.

In this example, the slave will be a simple ROM, and the master will be a test-bench like module which reads from the ROM and displays the read data.

We begin with looking at the template file for the ROM, `simple_rom.pt`:

```
port wbport wbs clk_i:wb_clk_i, rst_i:wb_rst_i, cyc_i:wb_cyc_i,
                stb_i:wb_stb_i, we_i:wb_we_i, ack_o:wb_ack_o,
                adr_i:wb_adr_i, dat_i:wb_dat_i, dat_o:wb_dat_o;

module rom(wb_clk_i, wb_rst_i, rst_i, wb_adr_i, wb_dat_i, wb_dat_o,
           wb_we_i, wb_stb_i, wb_cyc_i, wb_ack_o);

   input        wb_clk_i, wb_rst_i, wb_we_i, wb_stb_i, wb_cyc_i;
   input [:]    wb_adr_i;
   input [:]    wb_dat_i;
   output [:]   wb_dat_o;
   reg [:]      wb_dat_o;
```

```
            output         wb_ack_o;

            assign wb_ack_o = wb_cyc_i && wb_stb_i; // Always single clock cycles

            always @(wb_adr_i)
               case (wb_adr_i)
                 3'd0: wb_dat_o = 8'd69;
                 3'd1: wb_dat_o = 8'd108;
                 3'd2: wb_dat_o = 8'd105;
                 3'd3: wb_dat_o = 8'd10;
                 default: wb_dat_o = 8'd0;
               endcase
endmodule
```

We see in the port declaration that this port is of type `wbs`, Wishbone Slave. The well-known Wishbone signals are given in the label list, and are associated with Verilog variables.

Some of the variables are declared with no bit range. For example, `wb_dat_i` has a non-Verilog declaration of `[:]` where a Verilog compiler would expect something like `[7:0]`. The `[:]` declaration tells PERLILOG to choose the bit range according to context. More precisely, PERLILOG will copy the bitrange from the Verilog variable which `wb_dat_i` will be connected with.

This trick makes use of the fact that PERLILOG copies bit ranges between Verilog variables when they are connected with each other. We mention as a side remark, that when PERLILOG connects two Verilog variables as part of some port interfacing, the variables must have the same bit range, or an error is reported.

The logics which implement the ROM is not in our scope, and is quite trivial anyhow. So we skip that part.

Next, we have a partial look on the Wishbone master, `wb_master.pt`:

```
port wbport wbm clk_i:clk, rst_i:rst, cyc_o:cyc, stb_o:stb,
                we_o:we, ack_i:ack, adr_o:adr, dat_i:din, dat_o:dout;

module wb_master(clk, rst, adr, din, dout,
                 we, stb, cyc, ack);
   input [7:0]  din;
   input        ack, clk, rst;
   output       cyc, stb, we;
   output [7:0] adr, dout;

   reg [7:0]    adr, dout;
   reg          cyc, stb;
   reg          we;
```

```
reg [7:0]    q;

integer      i;
```

```
[... module body omitted ...]
```

```
endmodule
```

The port declared is a Wishbone master (`wbm`). Note that the labels are those of a Wishbone master, and as expected, they are stringwise slightly different than the slave's. Unlike the `vars` ports, the labels are mated by their meaning and not by being identical.

As we can see, the bit ranges here are defined explicitly. After all, they have to be defined somewhere...

And now the top-level module's template file, `top.tp`:

```
module top;

   reg    clk;
   reg    rst;
   port   clkrst vars clk:clk, rst:rst;

   always #5 clk = ~clk;

   initial
      begin
         // initial values
         clk = 0;

         // reset system
         rst = 1'b0; // negate reset
         #2;
         rst = 1'b1; // assert reset
         repeat(4) @(posedge clk);
         rst = 1'b0; // negate reset
      end
endmodule
```

The only essence about this module is that it generates the clock and reset. The port declaration is not in the beginning of the file, but somewhere in the middle – it doesn't matter to PERLILOG .

The port is of type `vars`. We shall see how this port is used in the script. For now, we note that this port has two labels, `clk` and `rst`.

And now we go to the main script, `trywb1.pl`:

```
use Perlilog;
init;

$top = template->new(name => 'top',
                     tfile => 'top.pt');

$test = template->new(name => 'test',
                      tfile => 'wb_master.pt',
                      parent => $top);

$rom = template->new(name => 'rom',
                     tfile => 'simple_rom.pt',
                     parent => $top);

interface($rom->getport('wbport'), $test->getport('wbport'),
          $top->getport('clkrst'));

execute;

silos->new(name => 'silos_configfile_creator')->makesilosfile;
```

Here the top module is taken from a template file, and not an empty Verilog object as before. Besides from that, there are few new things about this script: Three objects are generated. The first one, `top` is the parent of the two others, and hence these two are intantiated in `top`.

The `interface()` here involves the two Wishbone ports and the `vars` port, which is how it should be done: The clock and reset for the Wishbone bus are donated by some `vars` port, which is expected to have labels named `clk` and `rst`, with obvious meanings. As we can see, these two signals are explicitly chosen (they are not global in any way).

It is recommended to run `trywb1.pl` and view the results. The resulting Verilog modules are quite long and are more or less what one should expect, so we shall not go over them here. Still, seeing it working is worth the effort.

2.4.3   Connecting three Wishbone ports

By the nature of the Wishbone bus (like other SoC buses), connecting a master and slave is significantly easier than connecting two slaves to one master. We shall now see how this is done in PERLILOG , and how the dirty work is done for us.

There is another file in the directory, named `simple_rom2.pt`, which is almost identical with `simple_rom.pt`, except that the values of the ROM have been changed, so that the we can see which ROM is accessed as we run the test.

The following script, which is kept in `trywb2.pl` shows how two Wishbone slaves,

---

ROMs in this case, are connected to a single bus:

```
use Perlilog;
init;

$top = template->new(name => 'top',
                     tfile => 'top.pt');

$test = template->new(name => 'test',
                      tfile => 'wb_master.pt',
                      parent => $top);

$rom1 = template->new(name => 'rom1',
                      tfile => 'simple_rom.pt',
                      parent => $top);

$rom1->getport('wbport')->const('wb_adr_bits', 2);
$rom1->getport('wbport')->const('wb_adr_select', 0);

$rom2 = template->new(name => 'rom2',
                      tfile => 'simple_rom2.pt',
                      parent => $top);

$rom2->getport('wbport')->const('wb_adr_bits', 2);
$rom2->getport('wbport')->const('wb_adr_select', 1);


interface($rom1->getport('wbport'), $rom2->getport('wbport'),
          $test->getport('wbport'), $top->getport('clkrst'));

execute;

silos->new(name => 'silos_configfile_creator')->makesilosfile;
```

This script is very much like the previous one, except for one important difference: We assign values to two properties of each of the Wishbone slaves.

Before explaining those properties, we recall that our Wishbone master has 8 bits in adr_o. This is the entire address space, which is now shared between two slaves.

This sharing is implemented by a simple bus controller, which is generated automatically. It is responsible for activating the right slave according to the address. And once a slave is activated, there are a number of bits, taken from the LSB and up, which are passed to both slaves, to give each slave some address space. We tell the bus controller how to split the address space with these properties, as follows.

The wb_adr_bits property is set to the number of bits that we want the slave's adr_i to have. We chose 2 bits, so we have an address space of 4 cells. This was done

equally for both ports.

Now we have the bits [7:2] of the master's `adr_o` which don't reach the slaves, but are used to determine which of the slaves should be active. The `wb_adr_select` property tells the bus controller to activate the respective port, when the value of these bits equals the value of this property.

The setting of the ports' properties makes the two ROMs cover the addresses 0-7, in two non-overlapping segments 0-3 and 4-7.

When running the script in `trywb2.pl`, we get an extra module, that we didn't define explicitly, saved in `single_master_wb_controller.v`. This module contains the bus controller.

We may now see the advantage in not setting the bit range of `wb_adr_i` explicitly in either of the ROM modules (`simple_rom.pt` and `simple_rom2.pt`): When the ROM module was connected to the master directly in a one-on-one connection, all 8 bits of the master's `adr_o` went to the slaves, and hence the slave's address input was 8 bits wide as well.

But when a bus controller got involved, it needed to share the address space between two slaves, and the number of bits of the address input went down to 2. Indeed, if we view the Verilog files of the ROMs, we'll see that the input is declared differently, as needed by the environment.

And again, it is recommended to run the script and view the resulting files.

2.4.4   Multiple instances of the same module

One of the advantages of scripted Verilog generation, is that multiple instances of the same module are easily created.

In PERLILOG every instance of a module is backed up by a distinct object. By default, each object creates a Verilog file of its own, so each Verilog module is instantiated once. This is good as long as we don't have the same module repeating itself too many times. But in cases where the exactly same module is needed many times, the single instantiation model becomes cumbersome.

The philosophy of PERLILOG for this is: If *exactly* the same Verilog file is going to be written twice (except for the module's name), let's write the module file once, and instantiate it twice.

Let's take a quick example. We have another example script, `trymulti.pl`, which differs from `trywb2.pl` by the following added lines:

```
$rom3 = template->new(name => 'rom3',
      tfile => 'simple_rom2.pt',
      parent => $top);

$rom3->getport('wbport')->const('wb_adr_bits', 2);
```

```
$rom3->getport('wbport')->const('wb_adr_select', 2);

$rom3->equivalent($rom2);

interface($rom1->getport('wbport'), $rom2->getport('wbport'),
  $rom3->getport('wbport'),
  $test->getport('wbport'), $top->getport('clkrst'));
```

where the `interface()` command has replaced the previous one.

What this extra snippet does, is adding another object, `rom3` to the system. The special thing here is the `$rom3->equivalent($rom2)` command. It tells the system, that `$rom3` shouldn't create a file, since its Verilog code would be identical to `$rom2`'s anyhow. The system will trust this, and make an instantiation of `$rom2`'s module, instead of `$rom3`.

The system will also verify that the Verilog that `$rom2` generates is indeed identical to `$rom3`'s and issue a loud warning if this is not the case. Should such a warning be ignored, the Verilog code bundle may be buggy or may not even compile.

The equivalence dependency is central: If several objects are mutually equal, all equivalence declaration must be with respect to a single object (as opposite to chaining the dependency). An error message will be issued otherwise.

2.4.5 Mapping a non-Wishbone port to the bus

Many times we want to connect simple peripherals to the bus, and find ourselves dealing with the bus interface more than with the specific logic that we need.

We shall now see two examples of how a `vars` port can be mapped to a bus. The first one will be an interesting implementation of the ROM. This is `fake_rom2.pt`:

```
port fakeport vars r0:zero, r1:one, r2:two, r3:three;

module fake_rom(zero, one, two, three);

   output [:] zero, one, two, three;

   assign zero  = 8'd65;
   assign one   = 8'd66;
   assign two   = 8'd67;
   assign three = 8'd10;
endmodule
```

As a Verilog module, what we have here is weird. This module has four outputs only, whose bit range is unknown. All four outputs have constant values.

The interesting thing here is that we can connect `fakeport` to a Wishbone bus, even
though the port itself is a plain `vars` port. The label `r0` means that the Verilog variable
it's associated with should be mapped on the bus for read-only at address 0. `r1`, `r2`
and `r3` work in a similar way for mapping to higher addresses. We may thus use the
following script, `trywb3.pl`, which differs from `trywb2.pl` in only two respects:

- `simple_rom2.pt` is replaced with `fake_rom2.pt`.

- `wbport` is replaced with `fakeport`.

So in effect we're using `fake_rom2.pt` exactly like we used `simple_rom2.pt`. The
`trywb3.pl` is listed here only for the sceptics among us (feel free to skip it):

```
use Perlilog;
init;

$top = template->new(name => 'top',
                     tfile => 'top.pt');

$test = template->new(name => 'test',
                      tfile => 'wb_master.pt',
                      parent => $top);

$rom1 = template->new(name => 'rom1',
                      tfile => 'simple_rom.pt',
                      parent => $top);

$rom1->getport('wbport')->const('wb_adr_bits', 2);
$rom1->getport('wbport')->const('wb_adr_select', 0);

$rom2 = template->new(name => 'rom2',
                      tfile => 'fake_rom2.pt',
                      parent => $top);

$rom2->getport('fakeport')->const('wb_adr_bits', 2);
$rom2->getport('fakeport')->const('wb_adr_select', 1);


interface($rom1->getport('wbport'), $rom2->getport('fakeport'),
          $test->getport('wbport'), $top->getport('clkrst'));

execute;

silos->new(name => 'silos_configfile_creator')->makesilosfile;
```

What is more interesting to see, is how `rom2.v` turned out:

```
'timescale 1ns / 10ps

module rom2(wb_clk_i, wb_rst_i, wb_adr_i, wb_dat_i, wb_dat_o, wb_we_i,
  wb_stb_i, wb_cyc_i, wb_ack_o);

  input   wb_clk_i;
  input   wb_rst_i;
  input [1:0] wb_adr_i;
  input [7:0] wb_dat_i;
  input   wb_we_i;
  input   wb_stb_i;
  input   wb_cyc_i;
  output [7:0] wb_dat_o;
  output  wb_ack_o;
  reg [7:0] wb_dat_o;
  wire [7:0] zero;
  wire [7:0] one;
  wire [7:0] two;
  wire [7:0] three;

   assign zero  = 8'd65;
   assign one   = 8'd66;
   assign two   = 8'd67;
   assign three = 8'd10;

  assign wb_ack_o = wb_cyc_i && wb_stb_i;
  always @(wb_adr_i or zero or one or two or three)
    case (wb_adr_i)
      0: wb_dat_o = zero;
      1: wb_dat_o = one;
      2: wb_dat_o = two;
      3: wb_dat_o = three;
      default: wb_dat_o = 0;
    endcase

endmodule
```

The only piece that remained from the original `fake_rom2.pt` in this module, is the four assignments to `zero`, `one`, `two` and `three`. All the rest was automatically added because the `fakeport` was interfaced with an Wishbone bus. The inputs and output has changed to conform to Wishbone spec, and some logic has been added as well.

We don't explain this module in detail, because there is a similar example just coming up, and we are going to be quite detailed about that one.

For now, The most important point about all this, is that `fake_rom2.pt` is truly portable. It presents only the essence of logic (in this case – none), without any dependency on

the bus it is going to be connected to.

So far we only dealt with reading on the bus. We now move on to an example of similar `vars` port connection to the bus, for read and write.

2.4.6 Non-Wishbone port: A nicer example

For the following example, we work with the files at `examples/ex3`. The toplevel file, `top.pt` remains the same, and the Wishbone master file, `wb_master.pt` has been altered to perform write cycles as well.

We look at `adder.pt`:

```
port theport vars rw0:data1, rw1:data2, r2:sum, r3:diff;

  input  [:]  data1, data2;
  output [:]  sum, diff;

  assign sum = data1 + data2;
  assign diff = data1 - data2;
```

This is a complete template file. You may have wondered what function the module header (beginning with the Verilog keyword `module`) had in the previous examples. The answer is that it has no function at all – both the module name and the variable list are simply ignored.

To be honest, the module header is not completely ignored: Its position in the file is noted, and two module headers are not allowed in a single template file. But otherwise the template file parser simply wipes the module header and footer (`endmodule`) away.

So when writing a template file from scratch, there is no reason to have a module header, as in `adder.pt`. The support of module headers is there to allow easy adaption of Verilog files. And the previous examples included module headers merely to make Verilog programmers feel at home.

Once mapped on a bus, you can write a value on address 0, another on address 1, and read their sum on address 2 and their difference on address 3. The values that were written to addresses 0 and 1 can be read back. This was assured by using a `rw` label for `data1` and `data2`, rather than a `w` label (e.g. `rw0` and not `w0`).

The main script, `trywb4.pl` is very similar to the previous ones we've seen. Only names have changed:

```
use Perlilog;
init;

$top = template->new(name => 'top',
                     tfile => 'top.pt');
```

```
$test = template->new(name => 'test',
                      tfile => 'wb_master.pt',
                      parent => $top);

$unit1 = template->new(name => 'adder',
                       tfile => 'adder.pt',
                       parent => $top);

$unit1->getport('theport')->const('wb_adr_bits', 2);
$unit1->getport('theport')->const('wb_adr_select', 0);

$unit2 = template->new(name => 'logic',
                       tfile => 'logic.pt',
                       parent => $top);

$unit2->getport('theport')->const('wb_adr_bits', 2);
$unit2->getport('theport')->const('wb_adr_select', 1);


interface($unit1->getport('theport'), $unit2->getport('theport'),
          $test->getport('wbport'), $top->getport('clkrst'));

execute;

silos->new(name => 'silos_configfile_creator')->makesilosfile;
```

The template file `logic.pt` is more or less the same as `adder.pt`, so we won't deal with it here.

The Verilog module `adder.v`, which unfolds from `adder.pt` is a bit long, so we shall take it piece by piece. It begins like any Wishbone slave:

```
'timescale 1ns / 10ps

module adder(wb_clk_i, wb_rst_i, wb_adr_i, wb_dat_i, wb_dat_o, wb_we_i,
  wb_stb_i, wb_cyc_i, wb_ack_o);

  input  wb_clk_i;
  input  wb_rst_i;
  input [1:0] wb_adr_i;
  input [7:0] wb_dat_i;
  input  wb_we_i;
  input  wb_stb_i;
  input  wb_cyc_i;
  output [7:0] wb_dat_o;
  output  wb_ack_o;
```

```
reg [7:0] wb_dat_o;
```

Next, we have have some application-specific variables declared.

```
reg [7:0] data1_reg;
reg [7:0] data2_reg;
wire [7:0] data1;
wire [7:0] data2;
wire [7:0] sum;
wire [7:0] diff;
```

Note that two registers, `data1_reg` and `data2_reg` are declared. These are necessary to hold the values which are written to the module from the bus.

And here come a couple of lines that are familiar from the template file (this is what the module is doing, after all...):

```
assign sum = data1 + data2;
assign diff = data1 - data2;
```

The acknowledge is set up to respond for single-clock cycles:

```
assign wb_ack_o = wb_cyc_i && wb_stb_i;
```

Next the module allows reading by assigning a value to `wb_dat_o` according to the address:

```
always @(wb_adr_i or data1_reg or data2_reg or sum or diff)
  case (wb_adr_i)
    0: wb_dat_o = data1_reg;
    1: wb_dat_o = data2_reg;
    2: wb_dat_o = sum;
    3: wb_dat_o = diff;
    default: wb_dat_o = 0;
  endcase
```

Note that addresses 0 and 1 are readable, and give the value that was written to the same address.

Our next task is to respond correctly to bus writes. In our case, it's writing to our internal registers:

```
always @(posedge wb_clk_i or posedge wb_rst_i)
  if (wb_rst_i)
    begin
      data1_reg <= #1 0;
```

```
        data2_reg <= #1 0;
    end
  else if (wb_cyc_i && wb_stb_i && wb_we_i)
    case (wb_adr_i)
      0: data1_reg <= #1 wb_dat_i;
      1: data2_reg <= #1 wb_dat_i;
    endcase
```

Some may not like the fact that the bus' reset functions as the module's reset. But simplicity governs in these examples.

Finally, we make the connection between the registers and the variables that they are supposed to be there for:

```
assign data1 = data1_reg;
assign data2 = data2_reg;
```

2.5   Last example: Embedded Perl code

Our last example demonstrates how Perl code can be embedded into template files. A small script will write some of the Verilog code, which allows the module to be configured via its properties as an object, and not with "defines". It is hard to find a short example of when this is truly useful, because the main disadvantage of using "defines" is when the module is very complex, and the different configurations have several implications of the implementation of the module.

Instead, we shall see how our ROM module's data values can be set up from the main script. Those who know the powers of Perl will easily see how this can be extended to solve real-life problems.

2.5.1   The main script

We go back to the examples/ex2 directory. There we find the trywb5.pl script, which is identical to trywb2.pl, except:

- Both ROMs are created from a single template file, perl_rom.pt, rather than simple_rom.pt and simple_rom2.pt.

- The romdata property is set for both objects upon creation.

Here is the script itself:

```
use Perlilog;
init;
```

```
$top = template->new(name => 'top',
                     tfile => 'top.pt');

$test = template->new(name => 'test',
                      tfile => 'wb_master.pt',
                      parent => $top);

$rom1 = template->new(name => 'rom1',
                      tfile => 'perl_rom.pt',
                      parent => $top,
                      romdata => [69, 108, 105, 10]);

$rom1->getport('wbport')->const('wb_adr_bits', 2);
$rom1->getport('wbport')->const('wb_adr_select', 0);

$rom2 = template->new(name => 'rom2',
                      tfile => 'perl_rom.pt',
                      parent => $top,
                      romdata => [65, 66, 67, 10]);

$rom2->getport('wbport')->const('wb_adr_bits', 2);
$rom2->getport('wbport')->const('wb_adr_select', 1);


interface($rom1->getport('wbport'), $rom2->getport('wbport'),
          $test->getport('wbport'), $top->getport('clkrst'));

execute;

silos->new(name => 'silos_configfile_creator')->makesilosfile;
```

What happens here, is that two almost identical objects are generated. They differ in
their names and in the `romdata` property. As one can guess, `romdata` is the ROM's
data, so we can set the data as required from the main script.

2.5.2   The template file

We move on to view the template file `perl_rom.pt`:

```
port wbport wbs clk_i:wb_clk_i, rst_i:wb_rst_i, cyc_i:wb_cyc_i,
                stb_i:wb_stb_i, we_i:wb_we_i, ack_o:wb_ack_o,
                adr_i:wb_adr_i, dat_i:wb_dat_i, dat_o:wb_dat_o;

module rom(wb_clk_i, wb_rst_i, rst_i, wb_adr_i, wb_dat_i, wb_dat_o,
           wb_we_i, wb_stb_i, wb_cyc_i, wb_ack_o);
```

```
    input           wb_clk_i, wb_rst_i, wb_we_i, wb_stb_i, wb_cyc_i;
    input [:]    wb_adr_i;      // lower address bits
    input [:]    wb_dat_i;
    output [:] wb_dat_o;
    reg [:]        wb_dat_o;

    output          wb_ack_o;

    assign wb_ack_o = wb_cyc_i && wb_stb_i; // Always single clock cycles

    always @(wb_adr_i)
        case (wb_adr_i)
CASES
          default: wb_dat_o = 0;
        endcase
endmodule

perl
  $c = 0;
  $cases = "";

  foreach ($self->get('romdata')) {
    $val = $_ + 0;
    $cases.="            $c: wb_dat_o = $val;\n";
    $c++;
  }
  chomp $cases;
  $code = $self->get('verilog');
  $code =~ s/CASES/$cases/;
  $self->set('verilog', $code);
endperl
```

It differs from from `simple_rom.pt` from section 2.4.2 only in the two following re-
spects:

- The Verilog `case` clause has been emptied from its cases, and the string `CASES`
  has been put insted.

- Some Perl code has been added in a clause surrounded by the keywords `perl`
  and `endperl`.

Quite expectedly, what happens here is that the Perl code generates the heart of the
`case` clause, and replaces the `CASES` string with something meaningful.

These are the steps taken by the Perl snippet:

1. The `romdata` property is read and a `foreach` loop is engaged on its items.

2. The relevant Verilog line is appended to `$cases` for each item looped upon.

3. The object's `verilog` property is read into `$code`. Its value is the Verilog code that we would find in the Verilog file, if it wasn't for this script snippet.

4. By using a classic Perl substitute command, we wipe the `CASES` away, and put the value of `$cases` instead. All this happens on the string kept in `$code`.

5. `$code` is written back to the object's `verilog` property.

The beauty about this is that the module is configured via the object's property, and it's a Perl script that interprets its meaning, and takes action. This is a powerful tool, which makes it possible to configure the module with intuitive parameters, and not necessarily those that were needed to make the "ifdefs" work.

3   Template objects

3.1   About this section

The examples in section 2 demostrated some of things that can be done with template objects. With good common sense (which most Perl programmers have) it's possible to make some use of template objects quite successfully. Except for a few features (which will be presented in this section), trial and error is enough to get the most out of this specific tool.

But template objects are also a gateway to start learning the concepts of PERLILOG . Starting to work with template objects in the spirit of the environment, is a good preparation for more advanced use of the system.

Also, it's nice to know what we're doing. This section gives some insight to things that were slightly neglected previously.

This section will contain a lot of forward references. These references are not necessary to understand this section in a basic level, but they should encourage to gradually get to know the entire environment. Even though template objects is the simplest level of using PERLILOG , it offers access to most of the advanced features via its embedded scripting capabilities. Knowing the environment well is therefore rewarding in terms of what you can do with PERLILOG , even if you stick to template objects.

3.2   How it works

When a template object is created with `new()`, the template file is read and parsed. The parsing mechanism is not very sophisticated, and it's definitely not covering the complete Verilog syntax. The parser's main mission is to recognize the parts that should be interpreted by the template object, and to set up the internal data structures accordingly. This includes:

- Verilog variables (wires, registers, inputs and outputs) declarations

- Port declarations

- Embedded perl clauses

- The module header and footer

The module header and footer are ignored. Only their position within the read file is marked. The rest of the elements are handled by the template object, and do not appear in the final Verilog file.

Anything that isn't included in the categories mentioned above, or couldn't be parsed for some reason, is stored to be written directly to the final Verilog file.

Another way to look at the first stage, is that the parser eliminates, or filters away any part that it is able to parse in the way from the input template file to the output Verilog file.

It may come as a surprise that the variables are eliminated as well, as they are needed in the Verilog file. They are indeed eliminated, but they are also written back, possibly in a modified format, at some later stage in the PERLILOG execution. Altering the variable's attributes is one of the most important features of PERLILOG , and it's done quite easily, thanks to the way they are stored, during execution cycle of PERLILOG .

After the file has been parsed there is a rather complicated chain of events, part of which is described in this guide. Little of this is specific to template objects, so we may summarize it all with the fact that the standard Verilog headers are added to the module, along with the variables, and that the Verilog file is written.

3.3   What to expect from template objects

The purpose of template objects is to allow the Verilog programmer to concentrate on writing productive logic as much as possible, and hassle with anything else as little as possible.

The programmer is required to write some pseudo-Verilog code, which supplies some basic variable definitions, port declarations and the core Verilog code. Embedded Perl is allowed to make life even nicer.

As an extra feature, the template class was written to function as a simple importer for prewritten Verilog modules. This makes it possible to take some Verilog module, add some port declarations, and use it as a template file. If this works nice, it allows all the features of a "normal" template file.

It should be pointed out, that this is not the only solution to import prewritten Verilog modules. The static objects reads a Verilog module from a file and leaves it untouched. This is a safer, but yet less powerful way to import modules. See section 6.9.

Some precautions to take when importing an existing module with the template class are mentioned in section 3.8.

3.4   A template object as an object

After seeing `template -> new(...)` several times in the examples, it should be no surprise that PERLILOG is highly object-oriented. Section 5 goes into the issue of how to use objects in PERLILOG in detail, and until we reach that, common sense will be the best tool to follow things.

In the first main script (section 2.3.2) we had:

```
$ff = template->new(name => 'flipflop',
                    tfile => 'myff.pt',
                    parent => $top);
```

Just to have it cleared out, we explain the meaning of this again:

This creates a new object of class `template`, and puts its reference in `$ff`. The object is initialized with three properties, `name`, `tfile` and `parent`.

The `name` property is the name of the to-be Verilog module and also the name of the file to be written (appended with a `.v` suffix). It's also considered to be the object's name.

All objects in PERLILOG must have this property initialized upon creation (except for interface objects). The object's name is a unique identifier. For this reason, two objects must not have the same name (use `suggestname()` in case of doubt, see section 9.1.11).

The `tfile` property is a complete path (from the current directory) to the file which functions a template file for this object.

Last, `parent` is the reference of the "parent object", which is the object in which the resulting module will be instantiated. If this property is left undefined, the module will not be instantiated, which is the right thing to do on the top-level module.

Additional properties may be set up and read by embedded Perl scripts, as was shown in section 2.5.

3.5   Variable declarations

3.5.1   Same syntax, different meaning

In the examples we've seen that Verilog varibles (wires, registers, inputs and outputs) were declared as in regular Verilog (in fact, we only saw inputs and outputs).

The template file's syntax looks the same, but the underlying meaning is somewhat different. Variable declarations in a template file are not going to appear as they are in the final Verilog file, but they should be seen as declarations to the template objects. In some cases, this declaration will end up as a very similar line in the final Verilog file, so it won't make any difference.

But the variable declaration can change dramatically, for example if a Verilog `wire` is connected outside the module, which can make it change to an `output`. PERLILOG takes the freedom to make such changes as a normal action.

The variable declaration in template files is Verilog compatible, but the syntax is slightly extended, as we shall see next.

3.5.2   Undeclared bit range `[:]`

As we saw in the example of section 2.4.2, variables may be declared without specifying the bit range, as in

```
input  [:]     wb_dat_i;
output [:]     wb_dat_o;
reg    [:]     wb_dat_o;
```

Syntactically speaking, the empty bit range specifier, `[:]`, takes the exact place of its classic Verilog sibling, which looks something like `[15:0]`.

The empty bit range specifier means that the bit range is yet unknown, and should be taken from the context. It does *not* imply that the final bit range should be of some indexed format. In other words, a single bit, which is declared by not putting a bit range specifier, may be chosen by the system as the final bit range, instead of `[:]`.

3.5.3   Verilog `input` and `output`

Even though the declaration of Verilog `input` and `output` variables is legal in a template file, it may be misleading: A variable will appear as input or output as needed by connections with other modules, which is determined by the main script. The key factors here, are which port was connected to which, and the parental relations between the objects, both factors being unknown in the context of the template file.

Thus, variables that appear as `input` in the template file may be transformed into `wire` in the final Verilog file. `output` may be turned into `wire` or `reg`. Functionally there is nothing wrong with that, but it may be somewhat confusing.

In section 6.7.3 (and on) the issue of variable driving is discussed. For future reference, we only state here that Verilog `input` variables are interpreted as "driven" variables, and are assigned the type `in` (and not `input`). Verilog `output` are considered "driving" variables, and they are of type `out` or `reg` (and not `output`).

In essence this means that PERLILOG responds to the `input` and `output` keywords in the template file as an indication whether the value of these variables are assigned in the Verilog logic part. If the a variable was chosen to be `output` by the Verilog programmer, its value was surely assigned, otherwise it would be floating. And the opposite goes for `input`: If this was chosen, then the value is only read in the module,

surely it's not assigned a value. And this happens to be exactly as much information as the PERLILOG environment needs to have from the template file.

3.5.4 `wire` **and** `iwire`

For those who want to use template files without confusing themselves, there are two synonyms for `input` and `output`, which are `iwire` and `wire`. These synonyms can be used interchangably.

Variables that are declared as `reg` don't need another `wire` declaration, but it's not an error to have one.

3.5.5 **Variable rules summary**

It all boils down to the following list of rules:

- Write your Verilog module normally. If you want to write the `module` header and `endmodule` footer, fine, just keep in mind that they will be ignored.

- Declared `reg` and `wire` variables just as you would do in Verilog.

- If you don't assign the value of some variable within the module, because it should come from "the outside world", use `iwire`. This is the parallel to declaring the variable as an `input` in Verilog, or binding the variable to another modules `output` as part of an instantiation.

- If you want the variable's bit range to be flexible, use `[:]` where you would normally put the bit range. Just make sure that the bit range is defined explicitly on some of the variables to be connected.

We begin with looking at the template file for the ROM, `simple_rom.pt`:

3.5.6 `simple_rom.pt` **revisited**

We return to `simple_rom.pt` of section 2.4.2, with a few technical changes. This is merely a slight clean-up of the previous version, but it shows how a template file, which was written originally as a template file will look like:

```
port wbport wbs clk_i:wb_clk, rst_i:wb_rst, cyc_i:wb_cyc,
                stb_i:wb_stb, we_i:wb_we, ack_o:wb_ack,
                adr_i:wb_adr, dat_i:wb_dat_i, dat_o:wb_dat_o;

    iwire       wb_clk, wb_rst, wb_we, wb_stb, wb_cyc;
    iwire [:]   wb_adr;
```

```
iwire [:]     wb_dat_i;
reg [:]       wb_dat_o;

wire          wb_ack;

assign wb_ack = wb_cyc && wb_stb; // Always single clock cycles

always @(wb_adr)
   case (wb_adr)
     3'd0: wb_dat_o = 8'd69;
     3'd1: wb_dat_o = 8'd108;
     3'd2: wb_dat_o = 8'd105;
     3'd3: wb_dat_o = 8'd10;
     default: wb_dat_o = 8'd0;
   endcase
```

Functionally, this template is identical to `simple_rom.pt`.

3.6 Declaring ports

The `port` declaration within template files is just a convenient way to generate a port object, and make the current template object its parent.

The deeper discussion of ports can be found in section 7.1. For now, we rely on the examples to give an impression of what ports are and what can be done with them. But keep in mind that the concept of ports goes far beyond this.

A port declaration has always the from of:

`port` *port-name class assignments*;

It may go on for several lines, and is always terminated by a semicolon.

For example, we shall go back to port declaration we saw in section 2.3.4:

`port ffport vars clk:clock, rst:reset, d:data, q:register;`

The `port` is a special template file keyword. It is followed by the port's name. It's important to note that this name is not the port's name as an object, but rather a local name, which is used with `getport()`. Unlike object names, the local name is known only within the the object who owns the port, so the uniqueness needs not to be kept beyond the single object. This allows a template file to be used to create more than one object, since object names must be unique in a global context.

The *class* is the name of the PERLILOG class, which should generate this port. In this example we chose `vars`.

Then we have a list of assignments, which are separated by commas. There are two types of assignments:

- property=value assignments. Each assignment sets a property of the port object as a constant. The value is a scalar, even though PERLILOG supports lists as properties (the template class goes for simplicity).

- label:variable binding. This makes the variable known by a label name. In fact, all the label declarations end up as a single property in the object, `labels`.

The order of the assignments has no significance.

At the time of writing these lines, the ports that are supported are no more than those presented in section 2. For real-life application it's probably desired to extend the set of ports.

This task is quite easy, once the PERLILOG structure is well understood. Unfortunately, reaching that stage takes some effort, but the reward makes it worth. Section 7 is dedicated to explaining how to extend the set of ports, but it's based on everything until that point.

3.7   Embedded Perl

3.7.1   How it works

The idea behind embedded Perl is that since we have a Perl interpreter running anyhow, why not let it run some Perl snippets inside the template file?

The template file parser recognizes two kind of clauses:

- `perl` to `endperl`
- `perlonload` to `endperl`

Both clauses contain normal Perl code. They differ in the timing of the execution.

- `perl` clauses are executed as late as possible. Namely, as the last task of the implementation of the `epilogue` method (this method is explained in section 6.5). When these clauses run, the system is just before flushing to Verilog files.

- `perlonload` clauses are executed as the last thing done in the `new()` method of the template class, which is rather early. This means that when they run, the file has been parsed completely, all ports that are declared are set up, and the varibles are set up in the internal property structure. But the Verilog code, as well as the object's properties probably have a long way to go.

As we can see, neither is executed immediately on parsing, but the code is stored and executed at well-defined times. If there are several clauses of the same kind, they are executed in the same order as they appear in the template file (top-to-bottom).

3.7.2 `perl` **or** `perlonload`**?**

The choice between `perl` or `perlonload` depends on the operation we want to perform. In section 2.5 we saw how embedded Perl was used to implant a piece of Verilog code into the `verilog property`.

The choice of doing it late (`perl` clause) is preferred, because the Verilog content may be split up into three properties, `prehead`, `extrahead` and `verilog` itself during the early stages. This split-up is necessary to support the issue described in section 3.8.2. At any rate, when a `perl` clause is executed, `verilog` is just about to become a Verilog file, so it's complete with all headers and instantiations. So stringwise brush-ups are done best late.

But sometimes it's better to be early. For example if we want to set up some variable's bit range according to some property, it has to be done early, because the system needs this information before wrapping up the Verilogs. In general, setting up attribute-like properties should be done early.

A nice example is if we want to copy some properties from the object itself to its port:

```
perlonload
    $port = $self->getport('wbport');
    $port->const('wb_adr_bits', $self->get('wb_adr_bits'));
    $port->const('wb_adr_select', $self->get('wb_adr_select'));
endperl
```

With a template file including this code, we don't need to set the port's `wb_adr_bits` and `wb_adr_select` directly, but we set the properties on the object's property space.

3.7.3 Rules for writing embedded scripts

In general, the embedded scripts are just like any Perl script, which runs in a separate namespace (actually, a separate package).

This means that the embedded scripts may generate Perl variables freely, without worrying about name collisions.

The namespace is initialized with two variables. Neither of these should be written to.

- `$self` is the handle (reference) of the current object. It's useful to run methods with (like `get()`, `set()` and `const()` to access properties).

- `$errorcrawl` is a system variable which should not be touched. It controls the way some errors are reported.

Besides, there are some basic restrictions to follow:

- Predefined, system global variables should not be changed. This includes `%ENV`, `%SIG` and friends, as well as `$[`-like variables. If you must change them, use `local` to avoid environmental pollution.

- If the namespace is changed (with Perl's `package`), or `$Foo::Bar`-like access to variables is used, the reasonable precautions should be taken, as always when visiting your neighbour's namespace.

- The main script runs under package `main`. Don't trust any other package name to remain in place. The package names may change unexpectedly.

All of the embedded Perl clauses (`perl` and `perlonload`) are executed in the same namespace, but different object's Perl clauses run in different namespaces. This allows passing information among the clauses simply by putting it in a Perl variable, including from a `perlonload` clause to a `perl` clause.

As for errors and warnings, use `warn()` and `die()` like you would in a normal script[1].

3.8   Importing prewritten modules

3.8.1   Verilog compatability

The template class can be used as an importer for prewritten Verilog modules. Since covering all possible tricks in Verilog would mean writing a complete Verilog compiler, obviously there are restrictions of what the template class can accept.

We shall now review how the template class handles various possible elements in classic Verilog code.

3.8.2   The Verilog `module` declaration

When the parser reaches the Verilog `module` declaration, the following happens:

- The parser scans through the file until it reaches a semicolon (`;`), which signifies the end of the module declaration. The module header, that was just scanned, is ignored and its data is trashed.

- The parser verifies that this is the first time that a module declaration is encountered, and issues an error otherwise.

- Any lines that were meant to be sent to the final Verilog file until this point is moved to the `prehead` property. This chunk is considered "what came before the header", and as such it will be put before the header of the final Verilog file as well. This is merely intended to put comments in the right place relative to the header.

---

[1]Obvious, right? Well, it won't be this obvious in code generating classes... Been at section 6.10 yet?

`endmodule` is simply ignored.

3.8.3 Unhandled Verilog variables

In general, lines that begin with one of the "keywords" `reg`, `wire`, `iwire`, `input`, `output`, `inout`, `parameter`, `integer`, `real` or `time` are considered to be variable declarations.

Variables that can fit into some internal PERLILOG representation will be kept as such, and the respective declaration will not be passed on to the final Verilog output (it will reappear at a later stage of the PERLILOG execution, possibly somewhat altered).

Otherwise, the declaration is passed on, and will appear as is in the Verilog output file. In the latter case, the parser will at least save the names of the variables involved, so that the PERLILOG environment will not create another variable with the same name, in the same module. Saving the names will happen grossly in the following cases:

- If the variables is of type `parameter`, `integer`, `real` or `time`.

- If the variable is a multi-dimensional array and it's not an `input`, `output` nor `inout`.

- If the variable has a bit range which consist of anything else than plain numbers, and it's not an `input`, `output` nor `inout`.

- If the variable is initialized upon declaration with a "=" sign.

Finally, there are some situations, where the variable names will not be saved. A warning will then be issued. The reasons are more or less as follows:

- If the variable is a multi-dimensional array and is an `input`, `output` or `inout`.

- If the variable has a bit range which consist of anything else than plain numbers, and is an `input`, `output` or `inout`.

- If the parser got messed up with the syntax for some reason.

3.8.4 Verilog `task` and `function`

The template class handles `task` and `function` by simply flushing to the final Verilog output until `endtask` or `endfunction` is reached. Functions and tasks must not be nested in any way.

This bypass is helpful when importing test benches, which often make use of these.

3.8.5 Comments in Verilog

Multi-line comments with /* and */ are not supported. This means that these tokens will not be recognized as comments, so the lines inbetween will be taken in as usual. If it's plain text inbetween, it will probably not make any difference, but if there is anything there to confuse the parser, it will.

Another thing to note, is that one-line // comments are accepted, so

```
reg    myregister, // This is my register
       hisregister;
```

will be parsed just nicely, but

```
reg    myregister, /* This is not good */
       hisregister;
```

will make the comment appear to be some kind of variable.

4   Writing main scripts

4.1   Introduction

The "main script" is the script that is named in the command line which runs Perl. Its purpose is to choose the relevant objects, to configure them and to connect them. The main script is the part of PERLILOG which is closest to classic scripting.

Main scripts feel like regular Perl scripts: In particular, they are by default run under a `no strict` mode (unlike PERLILOG classes).

There is little to say about the main scripts, because most of the issues are covered elsewhere: How to get a main script together can be understood from the examples, and advanced use requires some knowledge about the object environment.

So we shall look at two issues: What happens during the `init()` call, and how to write add-ons to it. Then we'll present a nice debugiing tool, which dumps the objects in a readable format.

4.2   The `init()` call

In all the script examples, we had a call to `init()`. This call must be made after new classes have been added to the system by the main script, but before it starts to create objects. Why is explained in section 6.4.2.

When `init()` is called, PERLILOG reads the `site_init.pl` file in the `sysclasses` directory. It may seem paradoxal that this file is in this directory, but there are technical reasons for this.

The file is expected to have one single subroutine, `init()`, which is called. This routine performs the following tasks:

- Declaring the `template`, `static` and `silos` classes.

- Declaring the `wbm`, `wbs` and `vars` classes (ports).

- Declaring the `vars2vars`, `wbsimple` and `vars2wbm` and `wbsingmaster` classes as interface classes (section 7 deals with ports and interface classes).

- The global object is generated (explained in section 5.4.5).

- The system object is generated.

- If an `init.pl` file exists in the main script's current directory, it is executed.

The really interesting part about this is that a per-project initialization file can be generated, since the user's `init.pl` is looked for in the project home directory (unless the home directory is changed by the main script prior to `init()`).

Both `site_init.pl` and `init.pl` are executed by a call to `init()`, which must be written in a PERLILOG class style. How to write classes is widely explained in section 6.4, but the following rules should be enough to get a descent `init.pl` file:

- The initialization script is written in a subroutine named `init()`.

- All variables that are defined in the subroutine must be localized with `my`[2]. The subroutine (unlike the main script) is run in a `use strict` mode.

- Calls to PERLILOG -specific routines must be done with explicit package name, as in

```
&Perlilog::inherit('myclass', "myclass.pl", 'interface');
&Perlilog::interfaceclass('myclass');
```

4.3 The object dumper

A nice method of the `root` class, is the object dumper, which is intended for debugging. When called, it prints human-readable information about all or some of the objects. This information consists of the object's name, a brief description, its class and most important, all it properties. The data is presented in a format that is practical for human reading, so information is truncated occasionally where it would take too much space.

See section 9.1.10 for a complete description of this method. But trying

```
globalobj->objdump();
```

at the end of some main script should make things clear.

---

[2]If you have to, use complete `$Foo::Bar`-like identifiers for global variables

5 PERLILOG objects

5.1 Background

Even though the plain Perl environment supports objects as a natural part of its core, the syntax for using objects in Perl is quite cumbersome. This is probably a result of not wanting to make objects a special creature in Perl, but a natural follower of packages, references and hashes. The result is a very powerful and flexible mechanism that supports all the goodies that you could expect from object-oriented programming, and a few more tricks that are beyond the tradition. On the other hand, the path to object-oriented programming includes learning a few advanced topics in Perl, which is probably the reason why few scripts actually incorporate objects in Perl for real-life applications.

PERLILOG includes an environment for object-oriented programming which is, in fact, a wrapper of the native environment. It makes use of the powerful features that Perl gives in this field to create a very simple syntax to define classes, generate objects, call methods and access properties. Some features were added to support functionalities special for PERLILOG .

The PERLILOG object environment frees the programmer from awareness to references and blessed such in particular, packages and the need to know how to write Perl modules. The knowledge needed to use PERLILOG classes, as well as generating them, includes only plain Perl programming and the acquaintance with some additional functions, that are described in the sequel. Knowledge of general object-oriented programming techniques and concepts are needed as well, of course.

5.2 An example

Before getting into the complicated explanations, let's consider a simple example, which shows the basics of using objects in PERLILOG .

Let's assume that we have a file named `myclass.pl` which is:

```
sub sayit {
  my $self = shift;
  my $what = shift;
  print "I was told to say $what\n";
}
```

This file is used to declare a class that is used in the following script:

```
use Perlilog;

inherit('myclass','myclass.pl','root');
init;

$object = myclass->new(name => 'MyObject');
$object->sayit('hello');
```

We now explain this script briefly, only to give the general picture:

The first line in the script, `use Perlilog;`, makes Perl load the PERLILOG environment.

Then we have an `inherit` command. This tells the PERLILOG environment, that a new class, with the name `myclass` should be declared, the methods are defined in `myclass.pl` (listed just above), and that the class should be derived from the PERLILOG basic class, `root`.

We may now view `myclass.pl`, and see that it consists exactly of a subroutine definition, in well-known Perl syntax. In effect, this subroutine will become the method `sayit` in the `myclass` class. Note that the subroutine's first argument goes to a variable named `$self`, which is a handle to the called object. The argument that is given by the caller will reach the variable `$what`.

We now return to the main script. After the `inherit` declaration, PERLILOG is called with `init`. As this function implies, it causes the PERLILOG environment to initialize.

Having done this, we generate a new object of class `myclass`, give it the name `MyObject`, and put its handle (actually reference) in `$object`.

Finally, we call this object's method `sayit`, with the argument `'hello'`. As a result, the text `I was told to say hello` will be printed.

5.3 Properties

5.3.1 The basics

The properties in PERLILOG are divided into two types: Settable and constant. Settable properties are just like any variable: They can be assigned a value at any time, and the value can be changed as often as desired.

Constant properties, as the name implies, are assigned a value only once. Any attempt to change their value will result in a fatal error. This restriction is useful whenever changing the property's value will violate some basic assumption. For example, the `name` property is constant for good reasons: It is the object's identifier, and as a such, the connection between the name and the object's handle (reference) is recorded in the system.

The properties are accessed mainly with three basic methods (defined in the `root` class): `get` for retrieving properties' values, `set` for setting the values of settable properties, and `const` to assign values to constant properties. It goes something like:

```
$object->set('property', 'value');
$value = $object->get('property');
```

Properties are also assigned values (as constants) when creating a new object using `new`. See section 5.4 for more about this.

The properties in PERLILOG are in general lists. Scalars are considered as a list with one item. `get` behaves in such a way, that you'll get back exactly what you gave `set` or `const`, no matter if it was a list or a scalar.

Or a bit more detailed: If a property is assigned a scalar, and `get` is called in list context, you simply get back a list with one element, which is the value assigned. But if a property is assigned a list, and `get` is called in scalar context, only the first element is returned.

A detailed example is shown in section 9.1.3.

5.3.2 Property names

Property names behave like (and are, in fact) hash keys: They are case-sensitive, and all characters are allowed. It is recommended not to have newlines (`\n`) in the names, since these have a special meaning in the internal structure, and may cause strange property collisions.

For normal, practical purposes, case-sensitive alphanumeric names is the right thing to do. Don't make the names too long either, only as a matter of convenience.

See section 5.3.6 regarding the directory-like structure, in which the properties can be organized.

5.3.3 Undefs and empty lists

Sometimes a property is not set, and sometimes we want to set its value to "nothing". The PERLILOG API attempts to return a sensible value, depending on the situation and the context.

If a property has not been set (or has been removed), `get` will return an undefined value (a.k.a. `undef`) if it is called in scalar context, and an empty list if it's called in list context. It is perfectly proper (no warnings) to call `get` on an undefined value.

For example, if we assume that the property `'property'` was not defined when the following code snipped was run,

```
$scalar = $test -> get('property');
```

```
@list = $test -> get('property');

$elementscount = $#list + 1;
print "Was not defined and had $elementscount elements\n"
  unless (defined $scalar);
```

it will print "Was not defined and had 0 elements".

We may also remove a property by assigning it with either an undef value, an empty list, or a list containing only one undef value. The property will be removed, and will for all purposes behave as if it was never assigned a value. Naturally, a constant property can't be removed, but using `const` with an undefined value on an undefined property, will simply have no effect.

Here are a few examples:

```
$test -> set('property', ()); # Will remove 'property'
$test -> set('property', undef); # Will remove 'property'
$test -> set('property'); # Will remove 'property'
$test -> set('property', (undef)); # Will remove 'property'

$test -> set('property', (undef, undef)); # 'property' is set to a value!
```

5.3.4 More about constant properties

Constant properties differ from settable properties in two respects:

- Constant properties can't be changed after they are set

- A callback subroutine may be triggered when the constant is assigned a value.

The second issue of callbacks is described in the next section (sec. 5.3.5). We now focus on the issue of the exact meaning of the value being "constant".

If a property is not assigned a value, there is, of course, no problem with assigning a such. If the property has been assigned a value with `const`, PERLILOG allows `const` to be called again for the same property, only if the value to assign the property is "the same". We shall see what "the same" means below. It may not seem much of a privilege to be allowed to call `const` more than once with the same value, but this turns out to be useful in callback routines.

We now define what we mean with "the same": We recall that scalars are considered by the system as a list with one item, so we may reduce the entire discussion to lists. Two lists are "the same" if (and only if) they have the same number of elements, and each of the elements in the first list is Perl `eq` with the corresponding element in the second.

The choice of `eq` may not fit all properties. For some properties, a different criterion may be in place, in spirit of what means "the same" for that certain property. In that case, the `seteq` method (see section 9.1.13 for a description and example) may be used to substitute `eq` comparison with some other criterion.

When `const` is called for an already assigned property, it will never affect the existing value, and no magic callbacks will occur, even if the new and existing value are different (this statement is relevant when `seteq` was used to soften the criterions for "the same").

5.3.5 "Magic" callbacks

"Magic" callbacks are possible only on constant properies. The motivation behind this mechanism is to force some properties to have the same value. These properties may belong to the same object, or to different objects.

Suppose that we want property A and B to have the same value. One way of doing this is to look for all places in the script where A and B are set, and make sure that if one of the properties is set, so is the other one.

Another, safer and more elegant way to accomplish this, is to tell the environment, that we want some piece of script to be executed, as soon as A is assigned a value. This script will then copy the value of A to B. Then we do the opposite thing with B, so that the value of B is copied to A.

A "magic" callback request (done with the `addmagic()` method), is precisely asking PERLILOG to run some codepiece, as soon as some constant property is assigned a value. Naturally, if the property already has a value, the callback will be fired off immediately.

The technical details of this mechanism are widely described in section 9.1.12.

A few noteworthy points about "magic" callbacks:

- "Magic" callbacks are never executed more than once, due to the nature of constant properties.

- When using "magic" callbacks for its original purpose, the callback script `get`'s the value of A and `const`'s B with that value. And vice versa.

- Usually, a callback will be set for both A and B, so that they are mutually dependent. So if A is assigned a value, B will be assigned a value by virtue of a callback. As a result, B's callback script will be fired off, resulting in A being `const`'ed again. If all is done properly, A will be re-`const`'ed with the same value, so nothing will actually happen on this second callback.

- If property A and B are tied equal with a pair of callbacks, and B and C are tied in the same way, all three are, in fact, forced to have the same value: As soon as one of them is assigned a value, it will spread to all three by a chain effect.

- "Magic" callbacks may be used to force relations between properties, that are not necessarily equality. Complex relations between values of properties may be forced by writing the script snippets to maintain these relations.

- "Magic" callbacks are not necessarily used to force values on other properties, but can be used whenever we want a notification about some property's assignment.

5.3.6 The property path

Similar to paths of directories and subdirectories in filesystems, there are paths to the properties in PERLILOG . When referring to a property by a string (like in `$object -> get('property')`), we relate to a property at root level. We may access properties at a deeper level by something like `$object -> get(['my_things', 'property'])`. In this example, `'my_things'` functions as a "directory", and `'property'` is a property within that directory.

Property paths may be used in all places, where a property name is expected.

There is nothing special about putting properties into deeper paths, except that it lessens the chances to collide property names. There is no method to get a list of properties within a path[3]. You're supposed to know what you're doing.

In fact, the entire path structure can be viewed as a multidimensional array, in which the key consist of a number of scalars rather than one. A property is recognized by the exact sequence in the list, which defines its "name". Viewing this structure as directory-like is merely a recommendation to have the properties organized better.

We may thus conclude the following facts:

- `$object -> get('property')` and `$object -> get(['property'])` are exactly the same thing.

- It is OK to have a property with the name of what appears to be a "directory". We may thus call `$object -> set('like_dir', $value1)` and after that `$object -> set(['like_dir', 'like_node'], $value2)` and get two distinct and legal properties. This may be a confusing, but yet legal setting.

Some Perl programmers recognize that the squared brackets, [and], create a reference to a list, so the `\@`-type of reference can also be used, but this is usually not needed – it is enough to know that the path is a list of nodes to walk in the tree, with square brackets instead of round brackets. This list can be arbitrarily long, and it always begins from the root, and describes the nodes on the way to the desired property.

As with simple properties, it's proper to `get` undefined properties in unknown paths. An `undef` or empty list is returned, and no more fuss is made about that.

---

[3]except for the `objdump()` method, which is a debugging tool. See section 4.3

5.3.7   Methods for lists

Since the properties are considered as lists, there are four methods in the `root` class, which allows some easier access to the values: `pshift`, `punshift`, `ppush` and `ppop`. These methods are described in sections 9.1.15–9.1.18. In general, they behave very similar to their plain-Perl siblings. It is worth to mention, that these methods can't be used on constant properties, since they change the property's value.

5.4   Creating and using objects

5.4.1   The formalilties

Objects are created by a call of the format:
class -> `new`(*Initial properties*);
Such a call returns a handle (if you care, a blessed reference to a hash), which is usually kept in a scalar variable for future calls to the object.

When creating a new object in PERLILOG , the `name` property must always be set, or a fatal error will be issued. The object's name property is its identifier in the system, and must therefore be case-insensitively unique (the `suggestname` method is useful to create unique names, see section 9.1.11).

The initial properties are set by passing a hash, where the keys are the property names, and the values are their values. If we want to set a property to a list, the value of the relevant key in the passed hash should be a reference to a list, containing these values (possibly an anonymous list).

Note that properties that are assigned values with `new` are constant properties. See section 5.3 for more about constant properties and properties in general.

In PERLILOG as in common Perl, there is nothing really special about the word `new`, except that it happens to be the name of a method, that creates new objects. All classes in the system should create their objects by inheriting the `new` method from the `root` class. Extensions are allowed by using the `SUPER::` prefix on overriding methods of `new` (see section 6.4.5).

5.4.2   An example

While the rules above may appear complicated, this is how it really is:

```
$object = root->new(name => 'MyObject',
                    theKey => 'TheValue',
                    myList => ['One', 'Two', 'Three'],
                    Five => 5);
```

In this example, we create a new object of the `root` class, and we set four properties: `name`, `theKey`, `myList` and `Five`. It is important to note that the name of the properties don't need quotation mark (even though they are allowed), but the properties' values are indeed quoted, or given as any literal value in Perl. Moreover, note the square brackets in giving the value of `myList`. This property will assigned a plain list with the elements given, not a reference to a list.

For sake of clarity, if we continued this section's example with:

```
print $object->get('theKey')."\n";
@list=$object->get('myList');
print join(',', @list)."\n";
```

We would get:

```
TheValue
One,Two,Three
```

5.4.3 Property paths in `new`

In section 5.3.6 we mentioned a directory-like structure of properties. It is possible to initialize properties deeper than the root level with `new` by using references to hashes. Some Perl programmers may recall, that an anonymous reference to a hash can be created by using curly brackets, { and }, so the following example should make it clear:

```
$object = myclass->new(name => 'MyObject',
                       MyDir => {MyKey => 'TheValue',
                                 MyOther => 'TheOther'}
                      );
```

In this case, `$object -> get(['MyDir', 'MyKey'])` returns `TheValue`.

The rule is that if the value is a reference to a hash, then the key becomes a "directory", and the hash is interpreted as pairs of property names and their values, within that "directory". This may be recursively repeated to achieve unlimited depth in the path.

5.4.4 Common object types

We may divide the objects in PERLILOG into the following categories:

- Verilog objects – As their name implies, these object's purpose is to generate Verilog code. In most of the cases, these objects are associated with a Verilog file, to which the resulting Verilog code is written.

- Port objects – These objects are there more for their properties than their methods. A port in PERLILOG has a feeling of a connector (as in real-life cables). The port objects hold the information regarding what kind of port ("connector") it is, and what wires (Verilog variables in effect) it is connecting. See section 7.1 for more. At any rate, they have a very basic method set, and don't generate any Verilog of themselves.

- Interface objects – These objects are created automatically by the system, as a result of connecting ports. These objects are responsible for making the connection come true, in terms of declaring the connection of Verilog inputs and outputs and adding Verilog code snippets to modules as necessary to make the interfaces match. The interface objects are transparent to the main script programmer, except sometimes, when they generate independent Verilog files.

- The Global Object – This one and only object is generated by the system, and serves as a container for "global variables". In addition, it performs some tasks in the code generation process that need to be done once in the entire system.

- Other objects – PERLILOG doesn't restrict the classes or objects to fit into the categories mentioned above. In particular, classes that are derived from the `root` class can readily be used for general purposes.

5.4.5 The global object

The global object is useful when:

- We want to keep global information somewhere – the global object's property table is easily accessed.

- We want to run some basic method, but we don't have an object at hand. The global object is derived from the `global` class, which is derived from the `codegen` class, so it supports quite a few methods.

There is a special function to get its handle from the main script, `globalobj`. Within method declarations, `$self -> globalobj` should be used instead. See section 4.3 for an example of using the global object to reach a method.

The Global Object's properties function as the system's global variables. Different classes may add properties as necessary to this object. Care should be taken to avoid name collisions between property names: Use class-specific and unique names in the Global Object.

For example, if objects of some class need to be aware of each other, a property in the Global Object may be a list of handles to objects of that certain class. To do this, each object of that certain class will need to add itself to the list when it's created.

6   Creating code-generating classes

6.1   How this section is organized

The purpose of this section is to supply the knowledge which is necessary to write code generating classes, and set the basis for writing interface classes.

We begin with a general overview of what code generating classes are, and what they are good for. We then go on with showing the Perl code of a simple code generating class. The purpose of the example is to give an idea of what the code looks like, and the reader is defenitely not expected to understand it at once.

Section 6.4 is a long and rather tedious, but nevertheless important tutorial on writing PERLILOG classes in general. It is a continuation of section 5, which described how to use classes to create objects, and how to use the objects. Now we learn how classes should be written. Section 6.4 deals with any kind of class, so the information is useful for extending basic classes as well.

Section 6.5 deals with how the code generating classes fit into the environment. It details when and how the various possible methods of the classes are called.

Then we have a mini-section 6.6, which tips off regarding methods that are useful when writing a class.

We then reach the conceptually most important part in code generating classes, and especially in interface classes: The PERLILOG variables. Section 6.7 attempts to explain the relation between the Verilog variables (wires, registers, inputs, outputs) and the way they are represented within PERLILOG . When writing interface classes, the understanding of this issue is the true difference between an endless frustration and a piece of cake. Worth the time.

We spend a few words about how Verilog modules are instantiated in section 6.8. This issue is rather unimportant, unless we insist on generating instantiations explicitly in Verilog code.

Section 6.9 is a slight diversion from the main subject, as it deals with static objects, with a small discussion of classes. This piece would better fit into section 5, but little could be understood from it at that stage. Static objects is a significant tool for importing Verilog modules as-is.

Finally, section 6.10 presents how errors should be reported in from within a class. In a complex system, this may help the user to get a clue of what really went wrong.

A very short section 6.11 offers a checklist summary of the main things to keep in

mind when writing a class.

6.2 Overview

6.2.1 What's a code generating class?

Until this point, we've only seen two types of classes: The `template` class, which was used to parse template files and generate Verilog code from them, and the `verilog` class, which was used once to generate an empty top-level module. The `template` class is derived from the `verilog` class, and `verilog` is derived from `codegen`.

Technically speaking, any class which is derived (directly or indirectly) from the `codegen` class, is a code generating class.

Less formally, these are classes whose objects have the capabilities to serve the PERLILOG environment in creating Verilog code. In section 6.5 we shall see exactly what this means in terms of supported methods.

But to give a definition to bite on: Code generating classes are there to generate Verilog code. Most of them also create an independent file with Verilog in it.

6.2.2 Why bother learning this

With the template class at hand, it may seem useless to learn writing code generating classes. And indeed it may be the case. The template class is a powerful tool that allows embedded scripting, so what good will it make to write a new class?

There are a few motivations:

- Interface classes happen to be code generating classes. If you want to teach the system how to interface between port X and Y, you may get away with hacking an existing class. But odds are that you'll need to know what you're doing. Knowing how to write a code generating class is an essential step.

- Many of the techniques we shall see in the context of code generating classes can be used in template files, as embedded Perl. These are advanced, but yet useful tools, which broaden the possibilities of the template objects significantly.

- Some basic functional modules, such as memories, timers and counters are better implemented as code generating classes. By using this tool wisely, it's possible to create classes which adapt both their Verilog code and their configuration files to the target architecture. For example, a class for memories may be required to create an object which presents some certain amount of RAM space. The main script programmer will not need to consider how this memory should be allocated in the specific FPGA or ASIC: It will all be handled by the class.

6.2.3 PERLILOG objects and Verilog code

The Code generating object has a property, `verilog`, in which the to-be Verilog code is kept as a single scalar string. This property's value changes through the process, and is typically generated from the inside out: It usually starts with the object's essence code, put there by the object itself. Interface object may then add code snippets in order to make a correct matching of ports. When all is done, instantiations and headers are added. It is also possible to make some last-minute touch-ups after this stage.

As can be understood from this, the Verilog object itself is by far not the only one to access and modify its own Verilog code, but it creates the core of it. Other objects affect it as well.

A code generating object may or may not generate a Verilog file. See section 7.11.3 for more about this.

6.2.4 Instantiation and the object tree

For sake of simplicity, each Verilog module (that is, a Verilog file) is instantiated once only. To be more accurate, each Verilog object has a `parent` property, whose value is the handle of its parent object. This relationship means, that the parent object's Verilog code will include the instantiation of the child's Verilog module. Since each Verilog object can have only one parent, it can be instantiated only once (This is a waste of files where the same module could have been instantiated many times, as is currently not allowed in PERLILOG . But this will be fixed in the future).

All this results in an object tree, which represents the relations of instantiation between Verilog modules. The root of the tree holds the modules which is not instantiated anywhere.

6.3   An example

We shall now see yet another way to implement the ROM module, which we saw a lot of in the beginning of this document.

This time we'll create a special class called `rom`, rather than using the `template` class on a template file.

We start this example with presenting our main script `tryclass.pl`, which can be found at `examples/ex4`. It resembles `trywb5.pl`, which was presented in section 2.5.

```
use Perlilog;
inherit('rom', 'rom.pl', 'verilog');
init;
```

```
$top = template->new(name => 'top',
                     tfile => 'top.pt');


$test = template->new(name => 'test',
                      tfile => 'wb_master.pt',
                      parent => $top);


$rom1 = rom->new(name => 'rom1',
                 parent => $top,
                 romdata => [69, 108, 105, 10]);


$rom1->getport('wbport')->const('wb_adr_bits', 2);
$rom1->getport('wbport')->const('wb_adr_select', 0);


$rom2 = rom->new(name => 'rom2',
                 parent => $top,
                 romdata => [65, 66, 67, 10]);


$rom2->getport('wbport')->const('wb_adr_bits', 2);
$rom2->getport('wbport')->const('wb_adr_select', 1);



interface($rom1->getport('wbport'), $rom2->getport('wbport'),
          $test->getport('wbport'), $top->getport('clkrst'));


execute;


silos->new(name => 'silos_configfile_creator')->makesilosfile;
```

These are the differences between the script above and `trywb5.pl`:

- The `rom` class is declared with an `inherit()` statement. This tells the PERLILOG environment that the `rom` class should be read from `rom.pl`, and that methods should be inherited from the the `verilog` class. More about this issue in section 6.4.1.

- The two ROM objects are generated with the `rom` class, and not `template`. This is evident by calling `rom -> new()` instead of `template -> new()`.

- The `tfile` property is not assigned here, since there is no template file involved.

Basically, we use the `rom` class like we used the `template` class with the template file `perl_rom.pt`. The differences in the script above reflect exactly this.

So we move on now to see what we have in our class script, `rom.pl`. If viewed just as a normal Perl script, it appears to consist only of two declarations of subroutines,

---

with nothing beyond this. So if this file is executed directly with Perl, no errors will be reported[4], but nothing else will happen either – the subroutines are never called.

These subroutines are in fact method declarations, and there are two methods declared in this class: `new()` and `generate()`.

We shall now view `rom.pl` in pieces, in the order that they appear in the file. This will allow brief explanations in the middle, which are intended to give no more than a general picture. You may not understand the code now as you read it, but things will become clearer until the end of this section.

The file begins a `sub` declaration and a call to the inherited `new()` method, which generates a new object, and returns its reference, which is kept in `$self`.

```
sub new {
  my $this = shift;
  my $self = $this->SUPER::new(@_);
```

Then we continue with declaring some Verilog variables, which should exist in the final Verilog code:

```
  $self->addvar('wb_clk_i', 'wire', 'in');
  $self->addvar('wb_rst_i', 'wire', 'in');
  $self->addvar('wb_adr_i', 'wire', 'in');
  $self->addvar('wb_dat_i', 'wire', 'in');
  $self->addvar('wb_dat_o', 'reg', 'out');
  $self->addvar('wb_we_i', 'wire', 'in');
  $self->addvar('wb_stb_i', 'wire', 'in');
  $self->addvar('wb_cyc_i', 'wire', 'in');
  $self->addvar('wb_ack_o', 'wire', 'out');
```

This makes the PERLILOG environment recognize the variables, and assures that they will be declared in the Verilog code. More about this in section 6.7.

The next step is to create the object's Wishbone slave port.

```
  my $port = wbs->new(name => $self->suggestname('Wishbone_slave_port'),
                      parent => $self,
                      labels => [ clk_i => 'wb_clk_i',
                                  rst_i => 'wb_rst_i',
                                  cyc_i => 'wb_cyc_i',
                                  stb_i => 'wb_stb_i',
                                  we_i => 'wb_we_i',
                                  ack_o => 'wb_ack_o',
                                  adr_i => 'wb_adr_i',
                                  dat_i => 'wb_dat_i',
```

---

[4]If Perl is run with `-w`, an unquoted bareword warning will is issued

```
                                                 dat_o => 'wb_dat_o' ]
                          );
```

Note that the ports `name` property is not the name we use in the main script to find it. Also we can see that the `parent` property is set to the ROM object, since a port's parent is actually its "owner". Finally, the labels are set up in a similar way to what we saw in the template files.

The name we used in the main script's `getport()` is given now:

```
  $self->const(['user_port_names', 'wbport'], $port);
```

After this, we may find this port as `wbport` by calling `getport()` on the ROM object.

Finally, we make sure that the `romdata` property is set up. Just to be safe:

```
  wrong("The \'romdata\' property is not defined on ".
        $self->who()."\n")
    unless (defined $self->get('romdata'));

  return $self;
}
```

And we return a reference to ourselves (`$self`), as should any `new()` method.

So far with the `new()` method. We now go on to `generate()`. It starts in the typical way:

```
sub generate {
  my $self = shift;
```

Then, a pseudo-Verilog code snippet is stored in `$code`:

```
  my $code = <<'ENDOFCODE';
   assign wb_ack_o = wb_cyc_i && wb_stb_i; // Always single clock cycles

   always @(wb_adr_i)
      case (wb_adr_i)
CASES
        default: wb_dat_o = 0;
      endcase
ENDOFCODE
```

This reminds of the template file shown in section 2.5. It is the heart of the Verilog code, with the string `CASES` put where the the `case` lines should be.

It is then no surprise that the following Perl code looks quite the same as the one that was embedded in section 2.5:

```
              my $c = 0;
              my $cases = "";
              my $val = 0;
              foreach ($self->get('romdata')) {
                $val = $_ + 0;
                $cases.="             $c: wb_dat_o = $val;\n";
                $c++;
              }
              chomp $cases;
              $code =~ s/CASES/$cases/;
              $self->append($code);
            }
```

There is only one main difference between this code and the one that was embedded, and that is how `$code` is initialized and written to the `verilog` property.

In the embedded version, the `verilog` property was read into `$code`, altered, and then written back to the property. Here, `$code` is initialized with a constant string, altered, and then appended to the existing `verilog` property using the `append()` method. The `verilog` property may well be an empty string at this stage, but we can't know for sure in an environment where the classes' relationships are so dynamic.

We summarize this example as follows: We had two methods, `new()` and `generate()`. `new()` called the "original" `new()`, after which it made some initializations of its own: Verilog variables and the object's port was created and initialized.

`generate()` actually generated some Verilog code and appended it to the `verilog` property.

6.4   Classes and inheritance

6.4.1   Source files and classes

In PERLILOG , a class is defined by telling the system to read a file, and to consider the subroutine definitions in it as the methods of the named class. In the example, `inherit('myclass', 'myclass.pl','root')` means to read the source file named `myclass.pl`, and create a new class, `myclass`. Methods are inherited from the `root` class, as specified by the third argument.

Note that there is no necessary connection between the name of the file and the name of the class. Even more important, it is not specified in the file from what class `myclass` should inherit methods. The file is only a list of subroutine (in fact, method) declarations, whose object-oriented context is given by the script.

As is seen in the example, the word `myclass` may then be used as `myclass -> new(...)` in order to generate a new object.

It's possible to declare a class tree in one command, rather than a single class.

This is done with `inheritdir`, which is described in section 8.1.3. The trick about `inheritdir` is that the directory tree of files reflects the inheritance tree of classes.

An even more important feature of this special class definition scheme, is the possibility to enrich an existing class with new methods, without changing its name. This is done with the `override` function. This function can be used with either two or three arguments. In the two-argument case, we have something like `override ('myclass', 'otherfile.pl')`, assuming that we have already declared the `myclass` class with `inherit`. As a result of calling `override`, the given file (`otherfile.pl` in this example) will be read. If new methods are encountered, they are simply added to the class. If methods that collide with existing methods are found, those in the given file will override those that were defined before.

Any class in the system, including the `root` class may be enriched with `override`, which opens the ability make changes in the basic classes, without needing to "tell" the other classes, and still have our new code executed by them.

In some cases, the class we want to override may not exist. Since the purpose of overriding an existing class is merely to make sure that some specific methods are supported in this class, it makes sense to create a new class with the same name if the class doesn't exist already. This is the purpose of the three-argument format of `override`: If `override` is called with three arguments, and the class mentioned doesn't exist, `override` behaves exactly like `inherit` with the same arguments. Hence, it declares a new class, with the third argument as the class to inherit from.

If `override` is called with only two arguments, and the overridden class isn't defined, an error is issued.

As can be expected from a proper object-oriented environment, overriding doesn't necessarily mean cancelling out the previous functionality of the overridden method. By using the `SUPER::` prefix, we call the original method, if it exists (this syntax is the common Perl way to reach the overridden method). See section 6.4.3.

See sections 8.1.2 and 8.1.4 for specific information about these two functions, as well as examples of using them. Also see 6.4.7 for a graphic depiction of the class relations of the built-in classes.

A third function `underride` works opposite to `override`. Its purpose is to catch calls to methods that are not already defined in the class, or calls via the `SUPER::` prefix within the class. As the name of this function implies, all existing methods in the class will override those that appear in the file given by `underride`.

6.4.2   The phases of the object system

The inheritance relations between classes are set during the execution, and is not predefined in the classes themselves. Setting up the class tree during run-time makes the definition of classes more flexible, but altering the class definition for existing objects could also be an opening for exotic bugs. Because of this (and also due to the way the classes are loaded), there is a clear distinction between three phases in the

main script:

- Declaration of classes and their relationships. During this phase, `inherit` and `override` are called in order to set up the class tree. No objects should be generated during this phase.

- A call to `init`. Some internal variables are set up during the execution of this function, and the Global Object is generated.

- Creation and use of objects – during this phase, anything except attempt to alter the class tree is allowed.

In essence, this means that the classes and their position in the inheritance tree are declared before any of them is used. This is enough freedom to allow the system to choose one set of classes over another, or fine-tune the functionality of some methods, depending on run-time parameters. A PERLILOG script might thus build a different class structure depending on the target device, the synthesis tool or whether some other external tools should be used.

But these manipulations can be done until the first object is created, which happens when `init` is called.

Even though it is possible to stretch these limits a bit without getting error messages, it is highly recommended to comply with this phase structure.

6.4.3   Class definition

All PERLILOG classes, including the "built-in" classes, are defined by source files, in which the class' methods are defined. These files are perfectly readable as plain Perl files, but special rules apply to make them useful method declarations.

The most strongest rule, is that all variables inside the subroutines must be declared "locally", that is, using Perl's `my`. The source files are read in a "use strict" context, so if a variable is mentioned without being explicitly `my`'ed, a fatal error will be issued at the loading of the class.

This is more than a technical rule for programming: It is strongly recommended not to have any global variables other than properties of the Global Object. Even though a global variable in the source file (the package, actually) might be a tempting shortcut, it may cause strange bugs as the PERLILOG environment plays freely with namespaces. A conservative "local variable" approach ensures steady functionality.

In short, a class defintions consists of method definitions, one after the other, where each method definition is a subroutine definition with no use of Perl global variables.

We now divide the method definitions to two groups: "Normal" methods and methods overriding `new`. Their distinction is in the fact that "normal" methods are called in association with an exisiting object, while methods that override `new` are called in order to create one, so there isn't necessarily any associated object.

6.4.4 "Normal" methods

In the example in section 5.2, a method called `say` was declared and used within an object. The general format of a method declaration is as follows:

```
sub name of method {
  my $self = shift;
  Your code here...
}
```

Writing a method is very much like writing a regular subroutine. The main difference is that a handle (reference) to the called objects is given as the first argument. It is customary to put this handle in the scalar `$self`, usually with `$self = shift`. There is nothing special about the name `$self`, except that it's easier to read the code when it follows this convention. After this `shift` operation, `@_` is the list of arguments, so the rest of the code can be written exactly like a normal subroutine. There are plenty of examples in this guide.

One important choice we have to make for each method we write, is if we want our method to override the functionality of a possibly inherited method, or if it should extend it. In PERLILOG as in plain Perl, defining a method means that it comes instead of an already defined, possibly inherited method, if it existed. If we want the method to do more than it possibly did before, we call the inherited method with a `SUPER::` prefix.

To demonstrate this, assume that we want the `donothing` method to be defined (in order to avoid an undefined method error), but we don't want to make any changes if it was already defined in the class we inherit methods from. The following method declaration will do the job:

```
sub donothing {
  my $self = shift;
  $self -> SUPER::donothing(@_);
}
```

Note that we explicitly call the inherited method. As opposed to normal method calls, a `SUPER` call does not generate any error if it doesn't exist (that is, if we actually didn't override any method, but declared it for the first time).

It is very important to be careful about the arguments that we pass in the SUPER call. We must always `shift` away the first argument, and thus make the inherited method see exactly the same arguments as ours (the "self" handle will be added again by Perl due to the call). On the other hand, we must be sure not to alter `@_` before making this call, or make a copy of the argument list before changing it (it is very popular to read arguments by using `shift`, which changes `@_`). All this is true, of course, if we don't want to intentionally change the argument list before passing it on.

Since we control the stage in which the inherited method is called, we may perform this call either before or after doing some functionality of our own method. Where it doesn't seem to matter, it is highly recommended to call the inherited method as soon

as possible (immeditately after `$self = shift`). This will minimize the chances to generate odd bugs as a result of mistakenly changing `@_`.

See section 8.1.2 for a more extensive example of SUPER calls.

A last remark about writing methods, which goes for any object-oriented programming: In places where you would write a subroutine in normal Perl, write another method in the class, and use it as a method by calling it via the `$self` object. Don't be paranoid about someone else changing the way your class will work. It's a good feature, and there are plenty of ways to screw things up anyhow.

6.4.5 Methods overriding `new`

The `root` class supplies a `new` method, which creates a new object of a given class. This method must never be overridden with any other method, but it may be extended. This is useful to add special properties to any new object, or any other operation that is needed whenever a new object comes to life.

This is done by having a piece of code as follows in the respective class source file:

```
sub new {
  my $this = shift;
  my $self = $this->SUPER::new(@_);
  Your code here...
  return $self;
}
```

Note, that unlike the "normal" methods, we don't get `$self` by `shift`'ing `@_`, but rather by calling the inherited `new`, which generates a new object for us. With this `$self` at hand, we may use it exactly like in any other method. `@_` is also exactly like in a normal method after `shift`'ing off `$this`, but it will be useless in many cases, since it consists of the initial properties, which will be initialized by `root`'s `new` method.

A word about `$this`: In almost all cases, `$this` will hold the name of the class. This is true when `new` is called in the *class* -> `new(...)` format. But it is also allowed to call `new` from an existing object, which will result in a new object of the same class.

For this reason, don't use `$this` as the class' name, but rather `ref($self)`. If you want to know the class' name before calling `new`, use `$class = ref($this) || $this`

The `new()` method usually returns `$self`, so except for in rare cases, this should be done by the extension as well.

In an interface class, the `new()` method must be written with special care. See section 7.10 for more about this.

6.4.6 The autoloading mechanism

The PERLILOG environment attempts to read source files as late as possible. In effect, each source file is read when an object, whose methods are based on the class, is created. The source file is read once, when it is needed for the first time.

This mechanism allows the declaration of more classes than are needed for execution, with minimal overhead for unused classes.

The autoload mechanism is transparent to the programmer. The only thing that needs to be taken into consideration, is that a source file may contain bugs or syntax errors, that will not be detected until the class is actually used. If a PERLILOG script is executed successfully, it does not indicate that all source files are free from syntax errors, but only those who were used.

6.4.7 The built-in class tree

A drawing of the built-in class tree is shown in figure 4. It includes the following
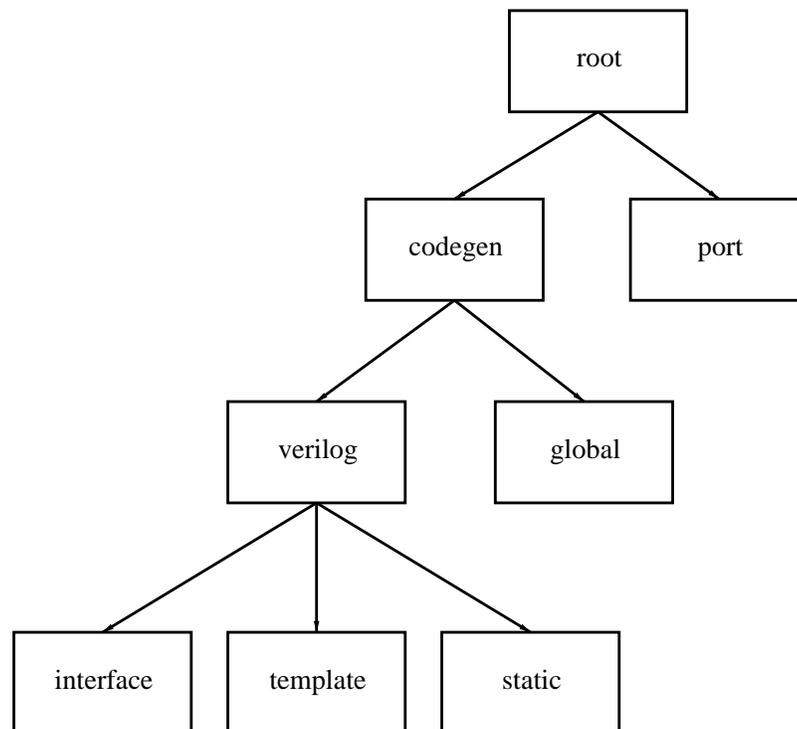
Figure 4: The built-in class tree

classes:

- The `root` class is the grandfather of all classes. It includes mainly general-purpose methods for handling properties and other system functionalities.

- The `codegen` class (Code Generating class) includes methods that are needed by any object that might deal with Verilog code. Despite its name, the Code Generating class is not intended to create Verilog code, but rather deal with other objects that do. It inherits methods directly from the `root` class.

- The `verilog` class is intended to actively generate Verilog code, either by creating its own file, or by adding code to another file. It supplies methods that deal with variables and other Verilog elements and functionalities. It inherits methods from the `codegen` class.

- The `interface` class inherits methods from the `verilog` class, and overrides a few methods, that make the class behave like an interface object.

- The `port` class overrides a few methods of those it inherits from the `root` class. Port objects are not supposed to execute much themselves, but rather hold properties, which is the reason why this object type has a very basic method set.

- The `global` class is derived from the `codegen` class. This class is used by the system to generate the global object. Hence this class should never be used by a script. It includes a few methods that are suitable for its position as the one and only global object.

The capabilities and relationships between classes reflect the situation before any add-ons by calls to `inherit` and `override`.

6.4.8 PERLILOG objects vs. plain Perl objects

If you happen to know object programming in plain Perl, then you have more knowledge than is actually needed, and you may possibly make some mistakes because of that. This section is here to help you avoid them.

Otherwise, you may skip this section with no worries.

The class source files are not read directly by the Perl interpreter. Rather, their content is `eval()`'ed after adding a header (this is done in memory, while the source file is left untouched). This header includes the well-known declarations for a proper Perl class (the Perl `package` pragma, and setting `@ISA`, if you care). These are transparent to both class writers and users. In particular, the package name may be different from the class name (even though the class name is used in the common way to generate new objects of the class).

The PERLILOG object environment is merely a wrapper for the common Perl objects, so the behavior is very similar. The main pitfall for experienced Perl object program-

mers, is to manipulate variables that the system assumes that the user is not aware of.

In general, the class source files should be as plain as possible: A list of subroutines, nothing else in the file. Every diversion from this is an opening to strange behaviours, and that shouldn't be needed except for rare cases.

So here are a few don't:

- Do not use the `package` pragma, or attempt to access a package by its name, using the `package::name` format. PERLILOG feels free to change package names without any notice, so you don't know what package name your class source file will get, even if you access the methods with a known class name. If is forgivable to access the `Perlilog` or `PLerror` namespaces directly, but this should be avoided, even at the cost of slower execution.

- Never access an object's properties directly. In other words, never make use of the fact that an object is actually a reference to a hash. It's not only that the way the data is stored may change without notice in future versions, but you may also corrupt the data structure.

- Never use `bless`. The objects that PERLILOG creates are properly blessed, and there should be no reason to rebless them. It is heavily assumed that all objects in the system were generated by the `root` class' `new` or one of its derivatives. Don't make a `new` of your own.

- Don't declare global variables within the package (that is, your class source file). It will work nicely at first, but will mess up things quite soon.

- Don't hassle with `@ISA`, `@EXPORT` and friends. These are global variables, so the previous "don't" should have been enough, but playing with these is even worse.

- Avoid `use` statements (except for `use Perlilog` once in the main script). Specifically, `use` should not be used as a pragma (like `use warnings`), while reading modules with `use` should be done very carefully.

6.5   The execution stages

6.5.1   The concept

In the example of section 6.3 we saw, that the ROM class merely consist of two methods, that both do relevant things, but we still have the missing piece of how and who calls the methods of the class.

PERLILOG was designed to be modular. That means that pieces of software can be added to existing classes, that may have been written by someone else, without compromising the intergrity of the code. One of the means for reaching this goal, was

to divide the process, which starts with the creation of an object and ends with the writing of Verilog files, into stages.

The technique is as follows: When a code generating object is created with `new()` it is listed in an internal system variable. This registration is done with the `registerobject()` method, which is called as part of the code generating object's `new()` implementation (see section 9.1.14 for more about this method).

When `execute` is called from the main script, the PERLILOG environment invokes the *multi-stage execution*. There is a preknown list of methods, which every registered object is called with. We shall go through this list soon, and explain each of them.

The order of the method calls is kept stricly: All registered objects are called with the same method before going to the next one. The order of the method calls is fixed, of course. As for the order, in which the objects are called, the rule of thumb is that the objects are called in the order they were registered, which is the order they were created. There are exceptions for this, as `registerobject()` allows to register an object for early or late execution. This is used by system objects, so simple objects should and do stick to this rule.

What we get is all objects progressing in parallel, which is helpful when one object's task depends on the progress of another's.

It is important to keep in mind, that objects that are registered during the multi-stage execution will *not* be included in the list of objects that are currently called. In other words, in order to have an object called to as a code generating object, it must be created before calling `execute`, and definitely not in the implementation of one of the multi-stage methods.

To the programmer of code generating classes, this means that the functional tasks should be written as methods, so that the method's spirit is kept in the implementation. This makes it easier to create later add-ons, which can rely on some maturity of the generation process.

6.5.2 The methods called

These are the methods that each of the code generating objects are called during the multi-stage execution, in this order:

1. `complete()` – This method is intended to give each object a chance to complete any missing information. This may involve reading some other object's property table, and copy pieces of information as needed.

2. `sanity()` – This method allows the object to run a sanity check on itself or other objects. This is useful for creating informative error messages when things will go wrong anyhow, only the error message is about to weird.

3. `generate()` – The mother of PERLILOG methods. The implementation of this method is expected to the smart work and create some Verilog code. Upon re-

turn from this method, the `verilog` property should have something substantial in it, but defenitely it's not the final thing to write to a file.

4. `instantiate()` – This method is intended to be executed by the system's implementation and should rarely be touched at all. At most, it can be extended. During this call, the objects are instantiated in their parent's Verilog code.

5. `headers()` – This shouldn't be tampered with too much either. The system's implementation creates module headers and Verilog variable declarations. When this method returns, the `verilog` property will look like a complete Verilog file. But it's still soon to save it to a file...

6. `epilogue()` – This method is there to allow things to be done just before saving the `verilog` property to a file. It may contain final brush-ups or anything that should be done in the last minute.

7. `files()` – Another method, which is meant for the system to handle. This is merely saving the contents of the `verilog` property to a file.

The method that wasn't mentioned in this list is `new()`, because it is called when the object is created, and not within the multi-stage execution. It's implementation should include anything that can't wait until the multi-stage implementation. For example, if some object creates other code generating objects, this must be done during the implementation of `new()`, because doing this during any of the multi-stage calls will be too late.

A final word: Despite the variety of methods, it may probably be enough to implement only the `generate()` method, and possible extend `new()`. Methods that are not overridden nor extended need not to be mentioned in the class file, so it can be really compact.

6.6   Useful methods

There is a comprehensive listing of methods at the end of this manual. What follows is merely a tip-off regarding methods that is good to know about, and weren't mentioned yet.

- The `who()` and `safewho()` methods are good to get a short and concise string that describes the current object in a way that humans understand. This is useful when creating error messages. Also, when writing a class, consider to override this method with something that will describe the objects better, but be sure to make something similar to the existing `who()` methods (overriding `who()` is enough). See sections 9.1.6 and 9.1.7 for more about this.

- The `globalobj()` method is useful to find the global object of the system. It is useful within method-defining code, since the common `globalobj()` command doesn't exist in the relevant namespace. Thus, a self-call to this method is the solution. See section 9.1.5.

- The `isobject()` method can be used to verify if some scalar is a handle to an object. This verification should be done if there is any doubt about this, before attempting to call a method of the alleged object. Otherwise an ugly Perl error may occur as a result of trying to use a non-object item as an object. See section 9.1.8.

6.7 PERLILOG variables and Verilog variables

6.7.1 Introduction

There is no way to understand PERLILOG variables without realizing how differently they are viewed, compared with Verilog.

In Verilog, a variable is a wire, register, input, output or some other creature, whose definition is closely related to its function within the module it lives in. The variable's definitions and the logics that come afterwards are one integral piece, which is called a Verilog module.

In PERLILOG , variable definitions and the logic code are two different worlds. Code generating classes write the logic code directly to the `verilog` property, but the variable definitions is the territory of the PERLILOG system. If we want to have a variable (say, a Verilog `wire`) in some module, we ask the PERLILOG environment to create it for us. The environment will then be responsible for creating the necessary `wire` statement in the Verilog file.

There are two reasons why every Verilog variable must at least be known to the PERLILOG environment:

- This allows the PERLILOG environment to make connections of modules, and make various manipulations on the variables.

- In order to avoid name collisions between existing variables, and those that PERLILOG may generate, all variable names must be known to the system.

All this fuss about variables has the purpose of allowing a set of useful features regarding the variable's attributes and connection. Easy connection between variables, easy Verilog inplanting of logic code into an foreign object and the ability to equalize bit ranges between variables are some of the advantages that we get.

6.7.2 Identifying a variable in PERLILOG

In PERLILOG there are two ways to say which variable we are talking about:

- The object which owns it, and the name of the variable in that object. Or, equivalently, the Verilog module in which is defined, and the name to use in Verilog

for the variable. This uniqueness is assured simply because Verilog considers modules as separate name spaces.

- A number, called the *variable's ID*. The ID is unique per Perl execution, and in particular, there is no context of Verilog module to access a variable via its ID.

There are various built-in methods in PERLILOG that deal with varibles. Some use the first type of identification, some use the second. The rationale is that if all variables involved live in the same Verilog mode (or equivalently, in the same object), they are conveniently accessed by their name in the module. On the other hand, when the operation may involve variables from different objects, the variable's ID is in place.

Note that by using variable IDs, we break the borders between the modules. We relate to the variable regardless of in which module it lives, which is very useful when connecting variables.

Two methods exist for change from one way of identification to another:

- `($obj, $var) = $self -> IDvar($ID)` converts a variable ID in `$ID` into a the object owning the variable (given in `$obj`) and the Verilog name of the variable (in `$var`). In scalar context we have `$var = $self -> IDvar($ID)`, and only the variable's *name* is returned. `$self` can be any code generating object, it doesn't affect the returned value (the ID is beyond objects). More in section 7.5.8.

- `$ID = $obj -> getID($var)` does the opposite. It returns the ID of the variable named `$var` in the object `$obj`. Note that it *does* matter which object we use to call the `getID()` with. More in section 10.1.4.

6.7.3   Variable drive and connection

Suppose that we wrote, in plain Verilog, something like:

```
assign driven = driving;
```

where `driven` is a Verilog `wire`, and both variables are of same size.

For most practical purposes, `driven` becomes an alias of `driving`: The value that we get from `driven` is always the same as the one of `driving`.

Let's try to look at this in terms of plain electronics: `driving` can be seen as some point on say, an electronic board. Just a wire or some test point where the voltage or logic level is set by some circuit's output. The `assign` statement above can then be seen as if we took a piece of wire, and connected the `driving` point to another point on the electronic board, that we chose to call `driven`. Disregarding impedance or fan-out issues, this is the electronic parallel to a Verilog `assign`.

But what if we now wrote:

```
assign subdriven = driven;
```

This is like taking another piece of wire, and connecting the point we called `driven` to another one, that we called `subdriven`. If we again disregard impedances and other real-life imperfections, we've just copied the voltage which originated in `driving` to `subdriven`. In other words, writing

```
assign subdriven = driving;
```

would be the same.

In a nutshell, this is the entire philosophy behind connecting variables in PERLILOG . Connection of variables in PERLILOG means creating groups of variables that have the same value all the time. One variable is actually getting its value from some Verilog expression, and that variable is the *driving variable*. All other variables (possibly one) passively get their value from the driving variable, and are called *driven variables*.

Or, equivalently: If a variable gets its value due to a connection with some other variable, by using the PERLILOG connection mechanism, it's a *driven* variable. Driven variables can be recognized by the simple rule, that they will be "floating" if we don't connect them with `attach()` properly (`attach()` is discussed in section 6.7.4).

On the other hand, if the variable is assigned a value in some Verilog statement, it's a *driving* variable. And to be accurate, here we refer to Verilog code which appears in the `verilog` property at the end of the `generate` method call, which does not, at that stage, include the module's inputs and outputs, nor instantiations that are generated by PERLILOG .

In the example of section 6.3, only `wb_dat_o` and `wb_ack_o` are driving variables. The rest are driven. If we look at the Verilog snippet, we see that only these two get their values in Verilog.

Register variables (Verilog `reg`) are always driving variables, by convention. The reason is that if we bothered to make them registers, then we probably want to assign them a value in the core Verilog code.

6.7.4   Connecting variables with `attach()`

The `attach()` method is an embodiment of the idea just presented: It works like a piece of wire connecting between two points in a circuit[5]. It's typically used in a statement like:

```
$self->attach($first, $second);
```

`$first` and `$second` are the variable IDs of the two variables we wish to connect. `$self` is a reference to any code generating object – it doesn't matter which. Neither

---

[5]In fact, the words "connect" or "bind" suit this method's name better than "attach", only the two former words are names of plain Perl functions...

does it matter if we swap the order of $first and $second in the call.

Neither $first nor $second needs to be the ID of a driving variable. The attach() method forces them to have the same value, and exactly as with the electronic board, both may take their value from some third variable, which drives them both. In fact, attach() merges *groups* of variables, and forces them to have the same value. The statement above actually means to force the same value on all variables that belong to the same group as $first and those who belong to the group of $second. It may sound complicated, but it's exactly what we do when connecting two points on an electronic board with a piece of wire: We force the same voltage on all points that were connected to either of the two points.

There is only one rule to keep strictly: In the end (actually, before the instantiate() execution stage is reached) each group of variables that have been merged with attach() must have *one and only one driving variable*.

If there is no driving variable in the group, we have a group of floating variables. If there are two or more, this is a collision. Both cases cause a fatal error during the instantiate() execution stage.

The only exception to this rule, is variables which have not been connected at all. In other words, a group of variables which consist of one variable is always legal.

There is no restriction at all regarding which objects the two variables belong to. The PERLILOG environment will make the necessary add-ons to the Verilog code to implement the grouping in an optimized manner. This includes:

- Converting Verilog wires and registers into inputs and outputs as necessary to connect them across modules

- Connecting instantiated modules' variables as necessary

- Adding Verilog assign statements when two or more variables of one group belong to one module.

In short, attach() simply means that the two variables are forced to have the same value, with no need to worry about how it is done. The PERLILOG system handles all this, including the verification of consistency, early during the instantiate() execution stage.

The attach() method is further described in section 10.1.2.

6.7.5 The variable's attributes

Each variable has three attributes. These are simply properties in the owning object's property tree (see section 11.2.1 for how to access them directly). The three attributes are:

- `type` – The variable's type, which is how the variable will be declared in the Verilog file (wire, register, output, output and register, input). More about this below.

- `drive` – either `in`, which means driven variable, or `out`, which means driving variable.

- `dim` – The dimension, or bitrange, of the variable. It's the dimension is a that will be given in the variable declaration in Verilog. Therefore, it has the format of `[3:0]`, or an empty string for a single-bit variable.

The `drive` attribute must be set when a new variable is created, and it can't be changed afterwards. The `dim` attribute may not be set until as late as the end of the `generate()` execution stage, but if it is assigned a value, it can't be changed afterwards (it's a `const`).

The `type` attribute must be set on new variables, but it may change. Even so, changing this attribute shouldn't be needed except for by the system during the `instantiate()` execution stage.

The variable `type` can be one of the following:

- `wire` – This is a Verilog `wire`. It may be either of drive `in` or `out`.

- `reg` – This is a Verilog register (`reg`). This format does not imply whether it is going to be used to generate a flip-flop or not during synthesis. The drive must me `out`.

- `input` – This is a Verilog `input`, and must be of drive `in`. Setting the type to `input` should be done only by the system,[6] converting a `wire` to `input` as appropriate when connecting it with a driving variable.

- `output` – This is a Verilog `output`, and must be of drive `out`. Setting the type to `output` should be done only by the system, converting a `wire` to `output` as appropriate when connecting it with a driven variable.

- `outreg` – Like `output`, only this is what we get when the system converts a Verilog register into an output. Always of type `out`.

We may conclude that to the code generating class programmers, there are only two possible values for the `type` attribute: `wire` and `reg`. If we chose `reg`, then the `drive` is always `out`. If we chose `wire`, then we'll choose the drive according to the simple question, of whether this variable will be floating if we don't `attach()` it at all. If the answer is yes, we go for drive `in`, otherwise we choose `out`.

---

[6]except static classes, which we shall discuss in 6.9

6.7.6 `attach()` **and the** `dim` **property**

An important side effect of the `attach()` method is that whenever two variables are connected with it, magic callbacks are set up mutually on the `dim` attribute (See section 5.3.5 for more about magic callbacks). In effect this means, that the `dim` attributes of both variables are forced to be equal. As soon as one has the attribute set (be it immediately), it is copied to the other variable.

The total effect of this is, that all variables that are grouped together (electronically "wired"), are also forced to have the same `dim` attribute. Any attempt, at any time, to give some variable another `dim` attribute will end up with a violation of the constant nature of the property, which holds this attribute. This is in fact the mechanism which filled in the unspecified `[:]` dimensions in the template file shown in section 2.4.2.

6.7.7   **Creating new variables**

There are three methods defined in the `verilog` class, which are relevant to generating new variables: `addvar()`, `suggestvar()` and `namevar()`. Both `addvar()` and `namevar()` create new variables. The variable's initial attributes are passed to these methods as arguments.

There is a small thing to be careful about when generating new variables: The variable's name must not collide with an existing name in the object (and hence the Verilog module). There are three ways of avoiding this:

- When we add variables of an object that we just generated, just keep track of the names. In this case we use `addvar()`. See section 11.1.2.

- The `suggestname()` method returns a name that is unique in the object it is called on. The name is based upon a suggestion that is given as argument, usually with a counting suffix when necessary. (See section 9.1.11. We then take the returned value, and use it as the name in `addvar()`.

- `namevar()` is the two-in-one solution: It calls `suggestname()` with the name it got, creates a new variable with `addvar()`, and returns both the name and the variable ID of the new variable (as opposed to `addvar()`, which returns only the ID). See section 11.1.3.

In some cases, we want to create Verilog code that uses some variable to read its value from ("read-only access"). If we have that variable's ID, `copyvar()` is useful, because it will create a new variable in the relevant object and `attach()` the new variable with the existing one, only if that is necessary. This is demonstrated in section 7.5.9.

Another special case is when a variable does not fit into any of the categories known to PERLILOG . For example, we may have Verilog `integer` variables, which PERLILOG doesn't know how to handle. Still, we must create a variable in order to prevent

PERLILOG from accidently creating another variable with the same name, in the same module.

This is done with a name-reserving `addvar()`: The method is called with only one argument, which is the variable's name. For example, `$obj -> addvar('i')` will allow us to use the variable `i` in our Verilog code without having PERLILOG messing things up (isn't `i` the variable we choose for loops?).

6.8 Automatic vs. explicit instantiation

6.8.1 Instantiation in general

The normal way in PERLILOG , is that modules are instantiated by the environment, according to their `parent` property. This happens automatically for all code generating objects, which have this property defined.

In some cases we need to perform an instantiation of either a module (see section 6.9) or some Verilog primitive elements such as gates, switches, or user-defined primitives (UDPs). This is called an explicit instantiation, because it is written directly to the `verilog` property either by some script, or it appeared in the template file.

6.8.2 Explicit instantiation

After getting used to PERLILOG doing all the instantiations for us, one may get the feeling that instantiation in general is the system's territory. But in fact, as long as we don't explicitly instantiate modules, which are going to be instantiated by the system (because they have the `parent` property defined), it's quite easy to stay out of trouble.

The only thing that really needs to be taken into account, is that the names of the instances must be unique. Instance names share the same namespace with plain Verilog variables (a separate namespace for each module).

There are two methods in the Verilog class, which help us keep track of the names: `addins()` and `suggestins()`, which are very similar to `addvar()` and `suggestvar()`. In fact, for this version of PERLILOG they do exactly the same, but `addins()` and `suggestins()` must be used for future compatability.

The following example shows how a Verilog primitive `nor` is instantiated:

```
my $ins = $obj->suggestins('nor_ins');
$obj->addins($ins, 'detached');
$obj->append("  nor $ins($out, $in1, $in2, $in3);\n");
```

Note that the `suggestins()` is used to avoid name clashes. `addins()` is called with a second argument (see section 11.1.6), which is mandatory on explicit instantiations.

If explicit instantiations appear in template files, exotic names are recommended. The current version of the template file parser doesn't reserve names of instances, so name collisions need to be avoided. It's not so clever to reserve the names with an embedded script either, since some future version of the parser may recognize instances and reserve the names.

6.9 Static objects

6.9.1 What it is

As we've seen (and will see more), the PERLILOG tampers with the Verilog code quite a lot. This is a blessing in most cases, but there are times when we want to know that the Verilog code will remain untouched. The straightforward example is when we use some prewritten module, which we want into the system ASCII-wise identical to what we got.

Static objects are intended for this exactly. These objects' `verilog` property isn't changed by any system routine, and in fact it's advisable to use the `const` method to write the desired Verilog module into it.

The formal definition of a static object, is a code generating object, whose `static` property is defined and has a value which Perl considers "true". As we shall see in the example of section 6.9.4, there is a `static` class, which loads a Verilog file, makes its content the static `verilog` property. It is advisable to base other classes, which need to be static, on this class.

Static objects are respected by many functions within the system, not only in the sense of not writing to the `verilog` property, but also by making it possible to use it as a normal object in many ways. The most important aspect is that variables can be defined, and ports can be attached to static objects, and then be interfaced in a normal manner. If the interfacing of some port would normally cause Verilog snippets to be injected into the `verilog` property, this will almost always be solved in an elegant way. In summary, static objects can be used like any other object, as long as we don't insist on changing its Verilog code.

6.9.2 How to instantiate static objects

There are two very different ways of handling static objects. The difference lies in whether the relevant module (or modules) is instantiated automatically by the system or not:

- A static object, whose `parent` property remains undefined, will not be instantiated by the PERLILOG environment. In fact, this holds true for any object, but it's useless for any object except static objects and the top-level object of the design. Uninstantiated static objects are useful when we want to write the in-

stantiation statement explicitly in some Verilog code. This could be done in the template file, some other static file, or possibly by some script, which appends the explicit instantiation to the Verilog code. The motivation of doing this may be that we have a completely written module, which we don't want to tamper with. If the module includes instantiations, it's easier to leave it as is, and incorporate the modules that are instantiated as static object. See section 6.8.2 for more about explicit instantiations.

- If the `parent` property is defined, the object will be instantiated automatically. In this case, the inputs and outputs of the module need to be declared to the PERLILOG environment, so that they can been connected like any other variables.

Note that if we want to instantiate the module of a static object more than once, we have to use explicit instantiation. The reason, among others, is that an object can't have more than one parent.

But from this point, we focus on static objects that are instantiated automatically.

6.9.3 Variables of static objects

Dealing with variables is a bit different with static objects. There are a couple of special rules:

- Unlike non-static objects, there is no need for PERLILOG to know about all variables that are defined in the Verilog module. Since PERLILOG is not going to add any variables of its own, there is no risk of name collisions. Accordingly, it's reasonable to call `addvar()` on the Verilog module's inputs and outputs only.

- On static objects, we define variables' `type` attribute exactly as it is in the Verilog module. For practical purposes, the `input` and `output` types are the only ones needed: For static objects it doesn't matter if the variable is of type `output` or `outreg`, and if it's a `wire` or `reg`, why bother telling PERLILOG about it anyway? Note that if we choose `wire` or `reg` for a variable that we use as an module I/O, PERLILOG will attempt to modify its type to the appropriate I/O type, fail, and report that failure to convert types.

6.9.4 Example of using a static object

We now demonstrate how the good old ROM can be created with a static Verilog object. The files are in the `examples/ex5` directory. In this directory, all `.pt` files are taken from the `ex2` and are identical to the original. Two files, that we shall show next, were added.

We begin with `therom.v`, which is a complete and legal Verilog file – nothing needs or should be changed:

```
'timescale 1ns / 10ps

module rom2(zero, one, two, three);

   output [7:0] zero, one, two, three;

   assign zero  = 8'd65;
   assign one   = 8'd66;
   assign two   = 8'd67;
   assign three = 8'd10;
endmodule
```

Next we see the `trystatic.pl` script, which is very much like `trywb2.pl` (see section 2.4.3).

The beginning of the file, which handles the `top`, `test` and `rom1` objects is identical to `trywb2.pl`, so we don't deal with it here.

The `rom2` object is going to be static. Note that the Verilog module already has the name `rom2`, because the PERLILOG environment can't change it to fit the module's name.

In the script we have:

```
$rom2 = static->new(name => 'rom2',
                    source => 'therom.v',
                    parent => $top);
```

Which creates a static object (by using the class `static`). The `source` property is the file to read the Verilog from.

Our next step is to make PERLILOG aware of the module's four outputs. We add the four variables to the object:

```
foreach $name ('zero', 'one', 'two', 'three') {
  $rom2->addvar($name, 'output', 'out', '[7:0]');
}
```

Note that the `type` attribute is `output`, and keep in mind that this would be a bad choice, hadn't this been a static object.

We now create a port of type `vars`, and associate the variables in the `labels` property:

```
$romport = vars->new(name => 'JustSomeName',
                     parent => $rom2,
                     labels => [ r0 => 'zero',
                                 r1 => 'one',
                                 r2 => 'two',
                                 r3 => 'three']);
```

The port's name, `JustSomeName`, only needs to be unique. We won't use it anyway, since we access the port with `getport()` and not with its name as an object.

But in order to make `getport()` work, we have to set a property:

```
$rom2->const(['user_port_names', 'wbport'], $romport);
```

The rest of the code is the same as in `trywb2.pl`. Actually we could have used `$romport` whenever we had `$rom2 -> getport('wbport')` in `trywb2.pl`, but this is merely a slight optimization.

It is really interesting to view the resulting Verilog files. In the original `trywb2.pl`, example, the `rom2.v` module had an entire Wishbone interface inplanted. This couldn't be the case of this example, since `rom2.v` comes from a static object. What we'll see, is that the interfacing code has moved to the bus controller module instead – it has to be somewhere.

6.9.5 Making a static ROM class

It makes sense to wrap all the extra code into a special `rom2` class. This has been done in the files at `examples/ex6`. We shall not go over them here, but it's recommended to look at the files if the subject is of interest.

In general, the code presented in section 6.9.4 took the form of an extension of the `new()` method. The `rom2` class suggested consist, in fact, only of this entension.

When reading the main script, note `inherit('rom2', 'rom2.pl', 'static')`, which makes `rom2` a `static` class with expansions.

6.10   Error reporting

6.10.1   Some philosophy

In a perfect world, errors tell the user or programmer what should be fixed. In the world where programmers rule, error messages don't exist at all, or they say what went wrong, which doesn't necessarily say anything about what should be fixed.

PERLILOG is intended to be an environment in which the code of many different people runs together. For this reason, it is important to supply the programmer with a variety of options to report that something is wrong, in a way that reflects the severeness of the error and also gives the reader of the error as much help as possible.

To begin with: `die` and `warn` should never be used within PERLILOG . Instead, other functions are supplied as follows. These functions can be used exactly in the way that `die` and `warn` would be used, both in the "main script" and in class definitions.

In particular: Unless the error message ends with a newline (`\n`), the line number on which the function was called will be appended to the error message.

The debug interface will hopefully develop way beyond this. But while writing both scripts and classes, it's compulsory to use the following functions for diagnostic reporting.

6.10.2 The list of functions

- `blow()` – This is similar to `die()`, and should be used when the scripts seem to be OK, but some unrecoverble error occured (such as failing to open a file). The error message should make sense to a user knowing nothing about the internals.

- `puke()` – This will work like `blow()`, but will also present a hopefully concise call stack dump. The ugly name of this function reflects what the output looks like and how graceful it is. To be used in reporting errors that occur only as a result of a bug. When the system `puke()`s on you, the error comes from the guts of the system. Accordingly, whoever will read the error message is expected to be either disgusted or having a good knowledge of the internals.

- `wiz()` – Use this function to throw warnings that will most probably not be understood by anyone else than yourself and a handfull of people. Use this mainly for testing your own class. These warning will be ignored in normal runs.

- `wizreport()` – This function should be used to detect conditions that you don't expect to happen, even after your class is in common use. Running this function will dump a call stack trace into a report file, and ask the user to send it to you. It still functions as a warning, so if execution must be stopped as a result of this condition, a call to `blow()` should take place after this one.

- `fishy()` – Generates a warning like the one you'd expect to get during the Sanity Check stage, but it may become before or after that stage. The contents of the warning text should be clear to a non-Perl user.

- `wrong()` – Like `fishy()`, but will set a flag to abort code generation, or abort immediately if in the middle of it. This is a way to report fatal conditions, and give other objects a chance to file their complaints before halting.

- `say()` – This is for general logging. This is basically reporting what you're doing, if you think it can interest someone.

- `hint()` – Like `say()`, but meant for more verbose information. The intention is to include information that may help debugging. It's plausable that these messages will be ignores unless the system is run in some verbose mode.

- `wink()` – This message is reserved for all of us who usually debug by putting meaningless `print`s in our code to mark that some milestone has been crossed. The messages will appear immeditely and visibly.

6.10.3 "Hidden" classes

Some of the error-reporting classes mentioned above make a call stack dump. There are cases, when we want to hide our class or package from this dump, in order to avoid confusing data from appearing. This applies mainly for system packages, and should not be used on "real" classes. If we want a class to be hidden from stack dumps, we define a global variable with `our $errorcrawl='skip'`, usually at the top of the file. This is a summary of possible values for `$errorcrawl`:

- `skip` – Causes the current package (or class) to be invisible in stack dumps. This was originally intended for the error-reporting package itself, so it wouldn't report its own internal calls.

- `halt` – Like `skip`, but applies also for any calls that the current class performed. In other words, the current class, and any calls it made are invisible.

- `system` – This causes the error message to be shortened slightly, by omitting the file name and line number, and saying "by System" instead. This is a slightly arrogant way to tell the reader of the error message, that the call was indeed performed, but it's useless to try find the error there.

Note that cleaning the error trace is nice as long as the information isn't needed, but it's very annoying if the error had to do with your code. Therefore, it is warmly recommended to avoid this kind of tricks unless you're absolutely confident about your code, and if it really fits the category of "system code".

6.11   Summary: How to write classes properly

This is a short checklist of things to keep in mind while writing a class in PERLILOG . There is nothing here that isn't mentioned elsewhere in this manual in detail.

- A class file should look like a clean list of subroutines. There should be nothing outside the subroutine blocks.

- Declare all variables with `my`. Don't use global variables.

- Use the global object where there is need for global variables.

- Always consider using the `SUPER::` prefix to call the overridden method. Make sure that all parameters are passed as is, unless intentionally doing otherwise. Make sure that the name of the SUPER'ed method is the same as your own.

- Use `blow()`, `puke()` and `wrong()` as appropriate instead of `die()`. Use `fishy()` and `wiz()` instead of `warn()`. In the error message, use `$self -> who()` to indentify your own object, and `$self -> safewho($other)` for some other object.

- Don't use direct subroutine calls, but method calls with as in `$self -> method()`.

- Access properties only with the standard methods (`get`, `set`, `const` and the built-in methods for list) or methods that make use of these. Never attempt to use the object as a hash.

- Be strict about the PERLILOG conventions. Always assume that your class must be able to work with other people's classes. Make no shortcuts.

7   Ports and creating interface classes

7.1   PERLILOG ports

7.1.1   An overview of ports

A crucial stage in understanding PERLILOG ports, it to realize that they have no formal definition, that describes what they actually are. They are what we want them to be – usually something to connect modules with.

Let's be more accurate: The formal definition of a port is an object, which is of some class that is derived from the `port` class, which is a very close relative to the `root` class. It is an object, which isn't originally intended to do more than to hold some properties. The port is there so we'll have something to connect, but actually it does nothing by itself. Its only purpose is to hold the information about its meaning, so that when we're running `interface()` on some ports it should be fairly clear what we want to happen.

There are two ways in which the port's information is held: One is the class it's made of as an object, and the rest is kept in the object's properties. Ports of different classes may be identical in their methods – the use of classes is merely to make strong distinctions between very different kinds of ports.

But in our minds a port should be "the thing that we connect". In section 2.3.4 the ports were compared with plugs in a computer, that we connect to the keyboard, the screen and so on. The feeling of a port should be something you connect to something, and whoops, it all works. It's much more important that a port will reflect something intuitive and straightforward, than represent the real down-to-earth action that connecting this port involves.

Another way to describe the idea of ports, is to imagine that we've just drawn a schematic diagram of our system to show to our boss. It's probably a few rectangles and arrows, that represent some flow of data or connection of signals. Now, let's see... Usually, we don't draw anything at the endpoints of these arrows, because we don't have anything interesting to write there. But if we have a set of PERLILOG ports, which fit in exactly into the end points of those arrows, then it's probably done right.

Naturally, the real-life meaning of connecting ports will end up to be some connection of Verilog modules via their inputs and outputs. While this will be true most of the time, there is no necessary connection between ports and Verilog wirework: Ports can be used for any purpose that means connection intuitively, and even more important, ports are not necessary to connect variables in Verilog. If all we wanted to do, is

to connect this variable in this module with some other variable in the other module, we're better of by using the `attach()` or `copyvar()` methods (see section 6.7 for an insight on the variable level).

In summary: PERLILOG ports are there to make the connection of modules an intuitive task. To make us, as humans, make these connections like we naturally think about them, and not necessarily like a computer likes it chewed down.

7.1.2 What ports must be

There are only a few rules that a port must keep in order to be OK with the system. These are:

- Ports must be generated from a class, which is derived from the `port` class (even indirectly).

- Ports must be proper PERLILOG objects, simply because they are objects. In particular, the existence and uniqueness of the `name` property must be kept as with any object[7].

- Ports must have the `parent` property set to the reference of the object which owns it upon creation.

Aside from this, ports can be set up as desired. It makes sense to keep attributes regarding the connection of the ports as the port's properties, but this is really a matter of implementation. In particular, ports may have properties which are not relevant to the port itself, but rather to the counterparts we intend to interface them with. For example, we saw in section 2.4.5 that a `vars` port was assigned the Wishbone-related properties `wb_adr_bits` and `wb_adr_select`, which were intended for the Wishbone bus controller.

7.1.3 The `labels` property

A port object is perfectly OK without the `labels` property set. On the other hand, this property is a comfortable way to point out the functional meanings of Verilog variables, as has been shown in the examples.

We have seen that the template class allows an easy set-up of this property via the port declaration. There are other advantages of using the convention of the `labels` property, such as methods in the basic `interface` class, which access this property easily.

The `labels` property is therefore a recommended way to map which variables are related to the port and how, but if it doesn't make sense to use it, there is no need to.

---

[7]We didn't say that the `name` property always exists. See section 7.10 on why port objects are often created in a special, transient way

So let's get to the details. The `labels` property is a list property. In particular, it can be read into a hash, and written from a hash, so

```
%mylabels = $port->get('labels');
$mylabels{'thelabel'} = 'TheVariable';
$port->set('labels', %mylabels);
```

is a perfectly sensible way to give `TheVariable` a meaning in the port `$port`, assuming, of course, that the `labels` property wasn't set up with `const()`.

When using a hash to read `labels` with, the keys of the hash are the labels, and the values are the variables. If the variables are numbers, they are the variable ID of the variable. Otherwise, its the name of the variable's name, as it appears in the port's owner's Verilog code.

Note that by using variable IDs, it's possible to refer to any variable in the system, and it's perfectly proper to take advantage of this. In section 7.3.5 we meet the `labelID()` method, which is a quick way to access the `labels` property, and get the data organized in variable IDs. This will prove useful.

7.1.4   New ports classes

If we want to add a new port type (in fact, a port *class*) to the system, there are two things to do:

- Making the port's class available

- Writing interface classes to support the port

There is a huge difference in the difficulty of these two tasks. The first task is a quick, formal thing to do. How to do this is explained in the paragraph to follow. To support the port with interface classes is a bit more of an artwork, but fortunately, the artistic side of doing this is to make it minimalistic. The good news are that it may be enough to write one or two not-so-complicated interface classes to get the port connected with any other port in the system. How to do that is shown in the rest of this section.

As for adding a new port class: Let's assume that we want a new class, named `myportclass`. First, we create a new file, which is the source file. It can actually be empty, but it's recommended to at least put a line of comment there. Ports are not supposed to do much, so there is no reason to add methods.

So we can create a file named `myportclass.pl`, which is exactly

```
# This is the myportclass -- no methods.
```

And then we must declare the class to the PERLILOG environment. We inherit from the `port` class, so we need the following statement to be executed before `init`:

```
inherit('myportclass', "myportclass.pl", 'port');
```

That's it. This was really easy. But now we go on to how to make `interface()` know what to do with this port class...

7.2 Interface classes overview

7.2.1 General

Interface objects perform whatever makes sense from the request to connect between a list of ports. In some cases, this merely means to `attach()` a few variables, as in the connection of two `vars` ports. In other cases, some interface code will also be added to the Verilog modules involved, like when a `vars` port was connected to a Wishbone bus. An independent Verilog module (such as a bus controller) may also be created when necessary. The definition of what interface modules do is as open as the definition of the ports they connect: It should make sense, that's all. In most cases, we expect some "wirework", though.

7.2.2 Approaching interface objects

Interface classes is probably the trickiest part is PERLILOG programming. It requires a good understanding of the interface object's place in the system, and also some good coding practice.

Many Perl programmers know that the quickest way to get good Perl code is copying someone else's code, and modifying it as necessary. This is probably true for the task of writing an interface class. For this reason, we shall start with seeing the Perl code of some interface classes. But unlike common Perl code, there are a few issues, which an interface class writer should be aware of, or terrible things happen when the classes are used. These issues are dealt with as they appear in the examples, so it's important to read them all. Then we approach some nevertheless important general issues, in the sections after the examples.

The general guidelines begin at section 7.8.

7.2.3 Assumptions to start with

Interface classes are plug-in-like creatures, which are never called by the main script directly. Rather, the main scripts requests a list of ports to be interfaced. As a result, the PERLILOG environment queries a list of classes for their willingness to take responsibility of the task. The query mechanism is described in section 7.9, but it is probably easier to grasp the entire picture by first looking closely at the interface classes themselves, and then see how they fit into the system.

In order to give the examples below any context, we make the following assumptions.

- The PERLILOG environment can call a method called `attempt()`, giving the method all the ports to be interfaced as arguments. The method will respond if the class, to which the method belongs, is ready to take responsibility of interfacing the given ports. In this case, it should create an object of its own class, and return its handle (reference).

- If a class agrees to take responsibility of interfacing, the PERLILOG environment *may* choose it to perform the task.

- In that case, the object that was created will function as a code generating object in the PERLILOG environment.

- Just like with any code generating object, will the PERLILOG environment call the `generate()` method in due time.

These assumptions are not accurate, and are only to start with – the exact picture will clear up as we go.

7.2.4 The Perl snippets presented

We shall now look on a few interface classes' Perl code. They are shown in snippets with explanations inbetween. Unless otherwise said, putting these snippets together, in the order that they appear here makes the respective class exactly (give or take remarks). Even when we view only parts of the Perl code, the pieces are shown in the same order in which they appear in the Perl files.

7.3 The `vars2vars` class in detail

We now look on the `vars2vars.pl` file, which is the simplest interface class. It normally resides in the `Perlilog/siteclasses` directory.

This class makes the interfacing between ports of type `vars`, by connecting variables with the same labels (as given in the `labels` property).

7.3.1 Example revisited

In section 2.3 we saw a simple connection between two `vars` ports. The outline of the interfacing is drawn in figure 5.

The `top` object is omitted from this drawing, because it isn't involved in the interfacing anyhow.

The box with dashed lines, which is marked `vars2vars`, represents the interface object, that was generated to interface between the two `vars` ports. The yellow-shaded area covers the ports, that this object interfaces.

Note that lines are drawn between the ports and their owners, but no lines are associated with the interfacing. This choice of notation will make sense as the drawings become complicated.

7.3.2 The `attempt()` method

The `vars2vars` class consists of two methods, `attempt()` and `generate()`. The first method is mandatory for interface objects:

```
sub attempt {
  my $this = shift;

  return undef
    if (grep {ref ne 'vars'} @_);

  my $self = $this->new(nick => 'vars_connection');
  return $self;
}
```

The purpose of this method is to check up if the current class has the ability to interface between the given ports. Note that it is similar to `new()` methods, in the sense that it's called before the object exists. In practice this means that the first argument is not the self reference (there is no object yet), but a string consisting of the class, or a reference to an object of the same class. In real life, only the first case apply.

This class considers itself competent to interface ports if and only if all of the ports are of type `vars`. The expression `grep {ref ne 'vars'} @_` searches for ports whose class is not `vars`, and if there are any, the method asserts the class to be inadequate. Note that in that case, no object was created at all.

Otherwise, an object is created of the current class, and its reference is returned. This is how the class signals that it agrees to make the interfacing.

Figure 5: Outline of interfacing two `vars` ports

7.3.3   The `nick` property

Note that the property `nick` is assigned, instead of `name`. The reason is that objects that are created as part of an `attempt()` call are considered "transient" by the system. Unlike any other case of object generation, these objects exist as Perl objects, but they are not recognized otherwise by the system, until a later stage, if at all.

If the object will be accepted by the system ("sustained"), the `name` property will be assigned with whatever the `suggestname` method will return, when called with `nick` as an argument. In other words, the object will end up with the `name` property set to a value equal to or similar to `nick`.

Note that using `nick` instead of `name` is only allowed with transient objects, and hence this should be used only in interface classes.

The issue of transient objects is widely discussed in section 7.10.

7.3.4   The `generate` method

The `generate` method is called upon just like any code generating object in the system. In this class, it begins with:

```
sub generate {
  my $self = shift;

  # Get the ports to connect...
  my @ports = $self->get('perlilog-ports-to-connect');
```

The opening line, `my $self = shift` is as usual – getting the self reference. Then a list of reference to the ports to connect is stored in `@ports`. The property `perlilog-ports-to-connect` is in fact set by the system to be an exact copy of the arguments (`@_`, the self-reference excluded) that were given to the `attempt()` method.

7.3.5   The `labelID()` method

Even though the `labels` property isn't mandatory on port objects, it's the recommended way to associate the names of variables within an object with their functional meaning. When writing an interface class, we are not interested in the variables' names, but their IDs. The `labelID()` method, defined in the code-generating class, does the work for us: We give it a reference to a port as an argument, it reads the `labels` property from the port object and resolves the variables' IDs. It then returns a hash, whose keys are the labels, and the values are ID's of the variables. (see section 10.1.7 for more about this).

The next lines of `generate()`:

```
my %conn = ();
foreach my $port (@ports) {
  my %h = $self->labelID($port);
  foreach my $label (sort keys %h) {
    $conn{$label} = join(',', (split(',', $conn{$label}), $h{$label}));
  }
}
```

The purpose of this code snippet, is to group all the variable IDs that are associated with each label. This is done by setting up a hash, %conn, whose keys are labels, and the values are the variable ID's attached to each label, as one string of comma-separated items.

7.3.6 Doing the actual work

Until this stage, we've played with our own internal variables. We now reach the stage where the class does some useful work. In our case, it is going through each of the labels, and `attach()` the variables in the group.

```
foreach my $label (sort keys %conn) {
  my @IDs = split(',', $conn{$label});
  my $first = shift(@IDs);
  fishy("The label \'$label\' has no counterpart while".
        " connecting the following ports:\n".
        join("\n", map {$self->safewho($_); } @ports)."\n")
    unless @IDs;
  foreach my $second (@IDs) {
    $self->attach($first, $second);
  }
}
}
```

In PERLILOG , the variables (wires) are `attach`'ed (connected) in pairs. In this example, all variables are `attach`'ed to the first variable in the list. The way `attach` works, it doesn't matter what pairing we make, as long as they are "electronically interconnected".

Another thing worth mentioning, is that the keys of %conn are scanned in a sorted manner (note the `foreach` declaration). Even though this may appear as a waste of CPU time, this assures that the items will be handled in the same order, no matter how Perl's `keys` shuffles them (which is platform-dependent). Even in cases where it appears not to make a difference, it's recommended to use `sort` in `foreach` loops.

That's it. Note that the complicated part about this class is actually related to the task of the class, and not the interface with the system.

We now go on to another class.

7.4   The `wbsimple` class

This class handles interfacing between a Wishbone master and slave. In essence it's very similar to the `vars2vars` class, only that it is sensitive to the meanings of the labels.

In section 2.4.2 we saw a connection between a Wishbone master and slave pair, along with a reset/clock port. The drawing, in terms of interfacing, is given in figure 6.
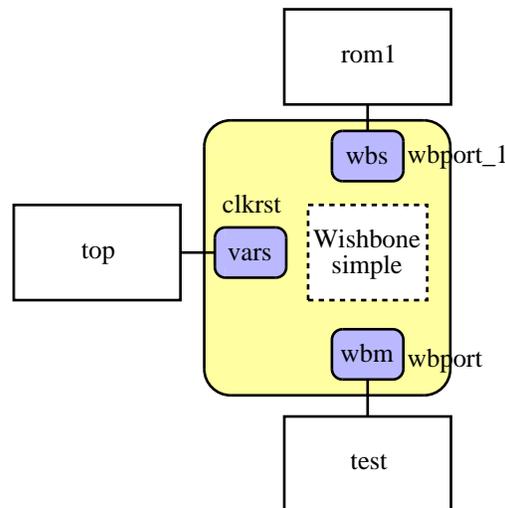
Figure 6: Outline of simple Wishbone interfacing

As before, we shade the involved ports with yellow, rather than drawing lines.

It is warmly recommended to read the code of this class. We only look at one little snippet here, taken from the `generate()` method:

```
$prevconn = $wbs->get('wishbone-connection-marker');
wrong("Attempt to connect ".$wbs->who." to ".$wbm->who.
      " when already connected to ".$self->safewho($prevconn)."\n")
  if (defined $prevconn);

$prevconn = $wbm->get('wishbone-connection-marker');
wrong("Attempt to connect ".$wbm->who." to ".$wbs->who.
      " when already connected to ".$self->safewho($prevconn)."\n")
  if (defined $prevconn);

$wbs->const('wishbone-connection-marker', $wbm);
$wbm->const('wishbone-connection-marker', $wbs);
```

where `$wbm` and `$wbs` are references to the Wishbone master and slave ports to be connected, respectively. They are assigned values in order to mark that the port has Wishbone ports have been connected, so that another attempt to connect the same port will fail with a meaningful error message.

This error check is actually not necessary to avoid double interfacing of a Wishbone port. Without this check, double interfacing would succeed, but an error would be issued at a later stage, because two variables would attempt to drive some node (for example, the Wishbone `dat_o`). Such an error message would appear to be a system bug, and its simple reason might not be easy to track.

7.5 The `vars2wbm` class

We saw in the example in section 2.4.5, that a port of type `vars` can be interfaced directly with a Wishbone bus. We shall now focus on the `vars2wbm` class, which is responsible for interfacing a single `vars` port with a Wishbone master port.

7.5.1 A more complex `attempt()`

We start with the `attempt()` method. It starts as:

```
sub attempt {
  my $this = shift;

  my @wbms =  grep {ref eq 'wbm'} @_;
  return undef unless ($#wbms == 0); # Single master, please

  my @allvars = grep {ref eq 'vars'} @_;
  return undef if ($#allvars == -1); # At least one 'vars'

  return undef unless ($#_ == $#allvars+1); # Only 'vars' and one 'wbm'
```

`vars2wbm` is more picky about the ports it is ready to interface: One master and the rest must be `vars`.

Note that the most common reasons to give up are tested first: `attempt()` failures should be declared as quickly as possible, since the method is expected to run a lot of times on inadequate sets of ports.

Next, we need to verify that there is only one `vars` port to be mapped on the bus. We expect at least one more `vars` port, which will supply the clock and reset.

This is done in the code as follows:

```
  my @extras = ();
  my $vars = undef;
```

```
foreach (@allvars) {
  my %l = $_->get('labels');
  if (grep /^(r|w|rw|wr)\d+$/, keys %l) {
    return "More than one address-mapped \'vars\' port in bundle"
      if (defined $vars);
    $vars = $_;
  } else {
    push @extras, $_;
  }
}
```

As can be seen, the port which have labels which fit the pattern of an address mapping is considered as the port to be mapped, and its handle is stored in `$vars`. The rest of the ports are held in `@extras`.

Note that if there is more than one "port to be mapped" among the ports to be interfaced, `attempt()` *returns a string*, which explains why this class didn't interface the ports. This is *not* an error message in the sense that the execution will halt. As we shall see later on, the philosophy of interface classes is that if one class fails to interface some ports, maybe some other class will succeed. Only if all classes fail to interface the ports, will the system report an error. In this case, will the system print out any hints it got while trying. More about this in section 7.9.

```
my $wbm = $wbms[0];

my $bits = $vars->get('wb_adr_bits');
my $select = $vars->get('wb_adr_select');

return ("Failed to find \'wb_adr_bits\' and \'wb_adr_select\' properties in ".
        $vars->who." while attempting to interface it with ".$wbm->who."\n")
  unless ((defined $bits) && (defined $select));
```

To begin with, we put the reference to out Wishbone master in a variable – `$wbm`.

Then we make sure that the mapped port has two essential properties set: `wb_adr_bits` and `wb_adr_select`. Note that if this condition isn't met, the routine merely returns with a hint. The class declares itself unwilling to interface, but maybe some other class will take the mission, even though this certain condition looks hopeless.

7.5.2 A call to `intobjects()`

At this stage things start to look promising. Hold tight, because strange things are just about to happen:

```perl
my $self = $this->new(nick => 'vars_to_wishbone');

my $mywbs = wbs->new(nick => 'vars2wbm_wbs_port',
                     parent => $self);

$mywbs->const('wb_adr_bits', $bits);
$mywbs->const('wb_adr_select', $select);
```

Two objects are created: One interface object of our own class, and a port object of class `wbs` – a Wishbone slave. Yes, we start to behave a bit like a main script over here.

The port is given a `nick` so its name will say something about its origin. Our own interface object is the owner of the port (`parent => $self`). In addition, the `wb_adr_bits` and `wb_adr_select` are passed on to the just generated Wishbone slave: That property is needed to interface the slave to a master.

The method ends as

```perl
my @objs = $self->intobjects($wbm, $mywbs, @extras);
return "Failed to match ".$self->safewho(@wbms).
  " with a vars-adapting Wishbone slave\n"
    unless (@objs);

return ($self, $mywbs, @objs);
}
```

The call to `intobjects()` is a request to interface between the Wishbone master, the slave just generated and the other `vars` ports that are kept in `@extras`. In essence, this is like calling the `interface()` subroutine, but the latter mustn't be called during the handling of an `attempt()` request. Instead, we have the `intobjects()` method. The differences are discussed in section 7.9.3.

And yes again, this really looks like if we wrote a small main script. Interfacing??

Calling `intobjects()` has a recursive smell about it, because this call will cause several calls to `attempt()` methods to various classes. Even though it's plausible that the `wbsimple` class will respond positively to this specific request to interface, we can't count on that when writing the `vars2wbm` class.

At any rate, `intobjects()` returns a list of all objects that were generated during the interfacing, if it was successful, or an empty list if it failed (this holds true even when the involved `attempt()` methods returned hints in strings).

If `intobjects` claimed success, our `attempt()` returns the two objects that it created itself, plus those returned from `intobjects()`. This sums up to all of the objects that were generated during the `attempt()` call. It is important that the items in the returned list are given in the order that they were created, and that the $self object is at the top of the list . This is assured by returning $self first, $mywbs after that, and the objects

returned from `intobjects()` last – this is the order in which they were created in this specific method. `intobjects()` always returns the objects in the requested order.

If `intobjects` fails, our `attempt()` fails as well, and we leave a hint behind us. This failure happens after we've generated two objects, that are not going to be used any more. These objects were automatically generated as transient, so they will be destroyed by Perl as soon as the `attempt()` calls terminates, and they left no traces otherwise in the PERLILOG environment. More about transient objects in section 7.10.

7.5.3   Whose port is this anyway?

We now move on to the `generate()` method of this class. It starts as:

```
sub generate {
  my $self = shift;

  my @ports = $self->get('ports');
  my ($wbs) = grep {ref eq 'wbs'} @ports;
  my @allvars = grep {ref eq 'vars'}
                  $self->get('perlilog-ports-to-connect');
```

In this class, we have two kinds of ports involved: The ports that were given to the class to interface, and the port which was generated by the class itself.

As with any (sustained) code generating object, the list of ports that belong to the object, is put in the `ports` property. We put this list in `@ports` and search this list for the single port of type `wbs`. In fact, in this special case the relevant port is the only one that belongs to the object, so `my ($wbs) = $self->get('ports');` would have worked the same. The `grep` search is there for demonstration of the technique.

Note that the value of `$wbs` in the `generate()` method is the same as the one in `attempt()`. It is important not to be tempted to store this value in a special property in the `attempt()` method implementation, since the object, which `$wbs` refers to, was created *after* the interface object. Therefore, the value in `$wbs` must not be stored as a property during the `attempt()` call.

And as in any interface object, the objects that were given as arguments to the `attempt()` call are kept in the `perlilog-ports-to-connect` property.

It's important to note the difference between the `ports` and `perlilog-ports-to-connect` properties: While `ports` are ports that belong to the object itself, `perlilog-ports-to-connect` belongs to other objects, and our object is going to interface between them.

7.5.4 The `whereto()` method

We now focus on the line that comes afterwards:

```
my $obj = $self->whereto;
```

The `whereto()` method returns a reference to an object, to which the Verilog code should be written.

We've seen in the example of section 2.4.5, that some Verilog code was generated in order to interface the `vars` port to a Wishbone bus, and that the code was put with another object's Verilog file, rather than in a file of its own. This was possible thanks to the call to `whereto()`, which is part of the code generating class. Its return value, `$obj`, is used in the rest of the method instead of `$self` where Verilog code is regarded, which results in the Verilog code put in another object.

When using `whereto()` care should be taken to assure that functional results will be achieved, no matter which object `whereto()` chooses for us. For a bit more about how `whereto()` works, and how we can affect which object it chooses, see section 7.5.10.

7.5.5 Creating new variables

At this stage we generate several variables at the hosting object:

```
my %wbsNames = ();
my %wbsIDs = ();

my ($clk, $rst, $adr_i, $dat_i, $dat_o, $we_i, $stb_i, $cyc_i, $ack_o);

($clk, $wbsIDs{'clk_i'})  = $obj->namevar('wb_clk_i', 'wire', 'in');
($rst, $wbsIDs{'rst_i'})  = $obj->namevar('wb_rst_i', 'wire', 'in');
($adr_i, $wbsIDs{'adr_i'}) = $obj->namevar('wb_adr_i', 'wire', 'in');
($dat_i, $wbsIDs{'dat_i'}) = $obj->namevar('wb_dat_i', 'wire', 'in');
($dat_o, $wbsIDs{'dat_o'}) = $obj->namevar('wb_dat_o', 'wire', 'out');
($we_i, $wbsIDs{'we_i'})  = $obj->namevar('wb_we_i', 'wire', 'in');
($stb_i, $wbsIDs{'stb_i'}) = $obj->namevar('wb_stb_i', 'wire', 'in');
($cyc_i, $wbsIDs{'cyc_i'}) = $obj->namevar('wb_cyc_i', 'wire', 'in');
($ack_o, $wbsIDs{'ack_o'}) = $obj->namevar('wb_ack_o', 'wire', 'out');
```

We use `namevar()` to generate new variables. The method returns the actual variable name (which may be slightly different from the suggested one) and the variable ID. These are kept, and are used later on.

Note that we call the `namevar()` method on the `$obj` object, rather than on `$self`, since the variables need to be defined in the hosting object's namespace.

7.5.6 Setting up the `labels` property

Then we set up our Wishbone slave's `labels` property.

```
$wbs->const('labels', %wbsIDs);
```

There is surprisingly much to say about this innocent-looking single line of code:

To begin with, this short-hand format of setting up the property is useful. If a hash, which binds between labels and their variables is at hand, it is this simple to set up the property (it goes the other way too: the `labels` property can be read into a hash).

Another issue, is that it's important to use IDs to describe the variables in the `labels` property, rather than their names. The reason is that if names are used, PERLILOG assumes that the variable names exist in the object to which the port belongs. In our case, the owner of the port is the `$self` object, but the variables are defined in the object referenced to by `$obj`. The may happen to be the same object, but it may also not be the case.

In other words, the `labels` property may consist of variables that are defined in another object than the one who owns the port, so using variable IDs is the only way on this port.

It may seem peculiar that the `labels` property of an port can refer to variables on another object. As a matter of fact, this feature is essential: The port had to be generated with a known owner during the `attempt()` call, because the port was used in the `intobjects()` call. On the other hand, we couldn't call `whereto()` during the `attempt()` stage, because the implementation of `whereto()` needs to know which port is connected to which – and that topology may not be set up at that stage. So the only way out, is to allow any variable ID to appear in the `labels` property. If an interface class is written properly, this fact doesn't make any difference in the difficulty of implementation.

Finally, it's worth to note that we've set up the `labels` property only during the `generate()` stage. This may be in the last minute, but not too late: Interface objects shouldn't assume that the `labels` property, of the ports they get to interface, is set until they are called with the `generate` method. Now, since interface objects are called in the order that they were created, we know that any interface object that knows about the port must be called after the current object, simply because the current object created the port.

7.5.7 The rest of the class

The code that follows deals with setting up internal data structures and generating Verilog code. Even though it's recommended to read the class in its entity, there are too many irrelevant details to go on with it here. We shall therefore look only on a few highlights. The interesting points about this class will be covered in the example of the `wbsingmaster` class.

7.5.8   The `IDvar()` method

Somewhere in the `generate()` method we have:

```
my $smartname = $self->IDvar($wireID);
($regname, $regID) = $obj->namevar($smartname.'_reg', 'reg', 'out');
```

in which we retreive a variable's name from its ID, using `IDvar()`. The whole idea here is to create a new variable, and give it a name that will help humans understand that it has something to do with another specific variable. We call `IDvar()` with the `$self` object, and not `$obj` merely to show that it makes no difference (`IDvar()`'s ignores which object it runs from).

See section 6.7.2 for more about this.

7.5.9   The `copyvar()` method

Later on, we have something like: (irrelevant parts omitted)

```
foreach (sort {$a <=> $b} keys %reads) {
  my $localname = $obj->copyvar($reads{$_});
  $readclause .= "        $_: $dat_o = $localname;\n";
  $obj->samedim($dat_o, $localname);
}
```

When this part of the method is reached, `%reads` is a hash containing the mapping of read operations on the bus: The keys are addresses from which variables are to be read on the bus, and the values are the IDs of the respective variables. The name of the Verilog variable which functions as Wishbone's `dat_o` is kept in `$dat_o`.

The purpose of the loop is to generate the heart of the Verilog `case` statement, which switches between the values of `dat_o`, depending on the address.

Note that we make a call to `samedim()`. This is explained in section 7.7.9.

Now, here's the tricky part: we don't know to which object the Verilog code goes, but we need a variable name to put on the right side of the Verilog assignment. The brute force way to do this is to generate a new variable in the `$obj` object, and `attach()` the new variable to the one we have the ID of in `%reads`. But if the variable we wanted happens to be in the `$obj` object, we could have used its name instead, rather than generating a new one. And even though PERLILOG is responsible for a Verilog `assign()` statement to connect the two variables, it's still ugly code.

The solution is the `copyvar()`. This method accepts the ID of a variable whose value we want to use in Verilog (only to read), and returns the name of the variable to be used for this, on the object on which the method was called. So by calling `$obj -> copyvar(...)` we know that we have a variable name that we can use in the Verilog code of `$obj`.

This method will create a new variable, and `attach()` it with the desired one if necessary. But before doing that, it will check up if the variable isn't defined in the desired object, or if there isn't a *variable which hold its value* as a result of some `attach()`.

Because `copyvar` can take advantage of previous `attach()`'es, it's best to use it after completing all the `attach()` calls in the method. Otherwise more variables than necessary may be generated. Either way, the resulting Verilog code will be perfectly functional.

7.5.10 `codetargets()` **and** `whereto()`

As we saw in section 7.5.4, `whereto()` gives a good choice of an object, to which interface Verilog code should be written. This choice is, among others, based upon calls to `codetargets()`.

It works like this: When `whereto()` is called, it calls the `codetargets` method of the "self" object. This method is expected to return a list of objects that are better than the "self" object to put the Verilog code in. Let's look at the `codetargets()` method for `vars2wbm`:

```
sub codetargets {
  my $self = shift;

  # Get the ports to connect...
  my @ports = $self->get('perlilog-ports-to-connect');
  my ($wbm) = grep {ref eq 'wbm'} @ports;
  my @allvars = grep {ref eq 'vars'} @ports;

  # Find our one mapped vars port.
  my $vars;
  my %l;
  foreach (@allvars) {
    %l = $_->get('labels');
    if (grep /^(r|w|rw|wr)\d+$/, keys %l) {
      $vars = $_;
      last;
    }
  }
  # Now we recommend these ports parents...

  return ($vars->get('parent'), $wbm->get('parent'));
}
```

This method looks for the `vars`-type port that is mapped on the bus, and the Wishbone master port to interface it with. These port's owners are given as recommendations to hold the Verilog code, in this order (the `vars` port is preferred).

Note that the method does not return itself as a candidate, since the "self" object is always assumed as a last resort. The "self" object should never appear in the recommended list.

Also note that the default `codetargets()` method returns an empty list, so any class which hasn't overridden this method will create objects that will volunteer to accept its own and other object's Verilog code (if an object is static, `whereto()` will not take it anyhow).

Another thing about the `codetargets()` method listed above, is that it doesn't check up whether the objects it recommends are static objects. In other words, it may recommend objects that are incapable of accepting the Verilog code. We conclude that `codetargets()` only gives some vague directions of where to go. The real decision is done by `whereto()`, which will obviously not pick a static object.

The answer given by `whereto()` is cached in the `perlilog-whereto-answer` property of the same object. For this reason, there is no need to make `codetargets()` efficient, since it's called only once.

An interesting case is when interface objects are chained, and each of them recommends ith neighbours. This may be the case, when it takes more than one `vars2wbm`-like objects to bridge between two port types. We would then like to put all the interface Verilog code in one place, rather than scattering it in various objects. What `whereto()` does, is to travel around making `whereto()` queries (recursively). The idea is that if some object X recommended Y, then we want to put the code whereever Y would put *its own code*. So the `whereto()` call to X will cause a `whereto()` call to Y. The fact that `whereto()` return values are cached, and that infinite recursion is avoided actually frees us from really needing to be aware of this mechanism, neither create such a one of our own. It comes down to writing a simple `codetargets` method, which recommends the next object to go, knowing that this recommendation may not be followed for various reasons.

7.6   Two examples revisited

Before diving into all the details of the next class, we shall stop and look at the general picture again. We now reexamine the result of interfacing two Wishbone slaves with one master, as was done in the example of section 2.4.3.

The outline of the interfacing is drawn in figure 7.

We can see from the drawing, that the central interface object, which was created as a direct result of interfacing the four ports, has three Wishbone ports of its own. These ports are mated with the ports of `rom1`, `rom2` and `test` using other interface objects, of class `wbsimple`.

Note that the notation of the `wbsimple` objects is sloppy in the drawing: We have not made another shading area to mark which ports they mate. The main reason is that each of the three matings involve the `clkrst` port as well, so it would mess the entire

drawing.

The yellow-shaded area in this drawing shows the involvement of the four ports with respect to the central interface object.

It is important to note, that the fact that the central object has ports of its own, has nothing to do with the fact that it also creates a Verilog file of its own. It could very well put its Verilog code in some other object, if that was desired.

Just to make things even more complicated, we move on to the example shown in section 2.4.5. In that example, `rom2` was replaced with a similar object, only that the new one had a `vars` port instead of a Wishbone port. This forced PERLILOG to squeeze in a `vars2wbm` interface object, as is shown in figure 8.
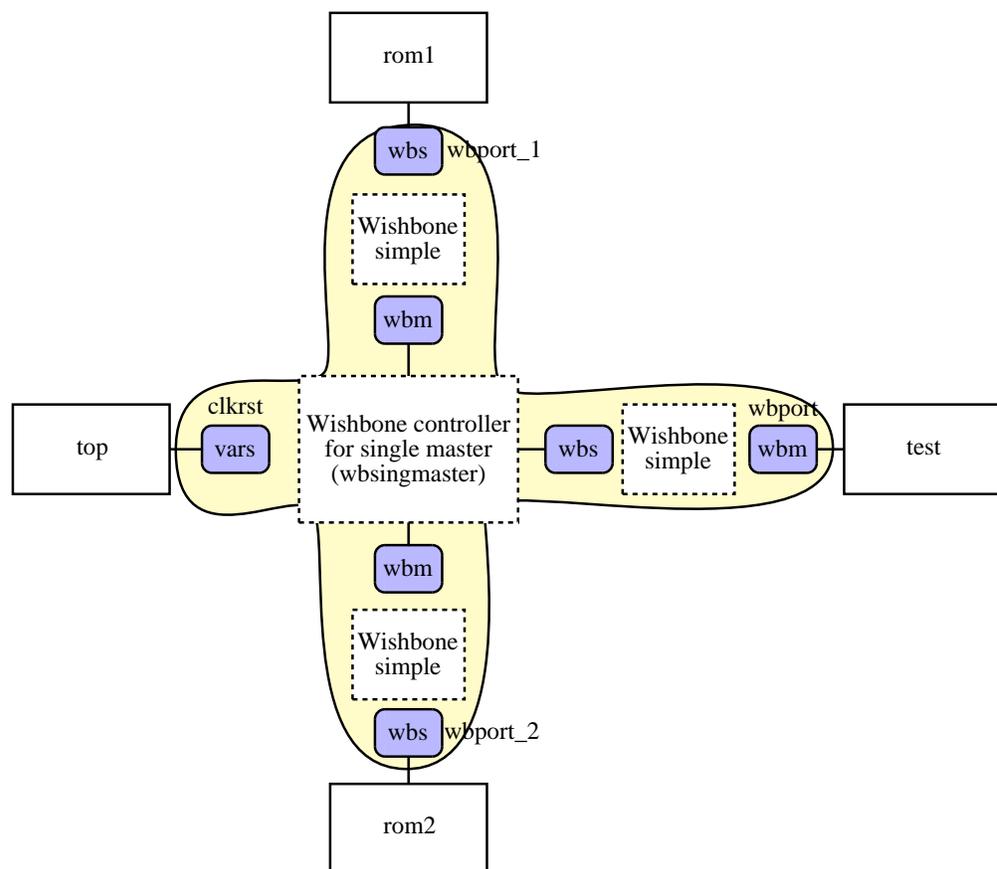
Figure 7: Outline of interfacing three Wishbone ports

What we can see in the second drawing, is that we have shaded the area which is covered by `vars2wbm`: From the `vars` port to the central interface object's Wishbone
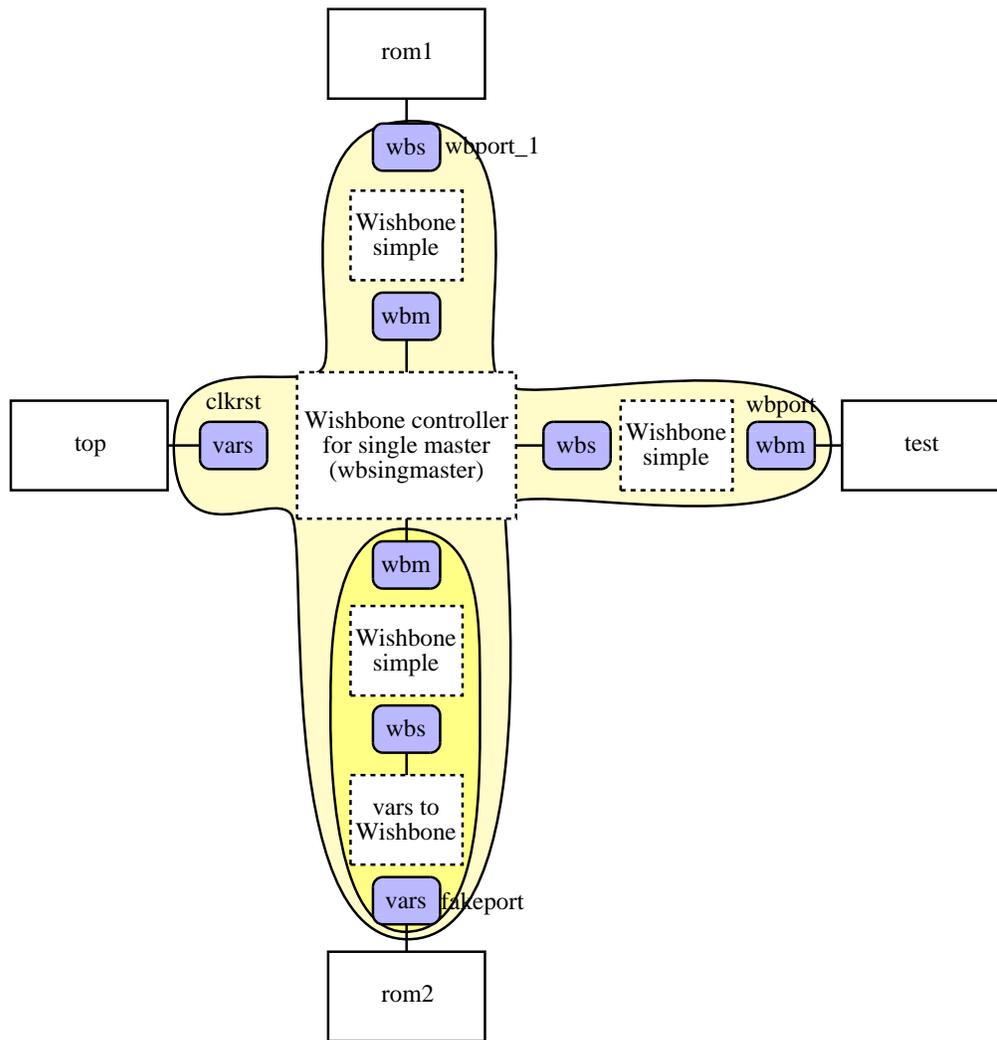
Figure 8: Outline of interfacing Wishbone ports with `vars` port

master.

The `vars2wbm` object is a bit strange with only one port (Wishbone), and no port to face the `vars` port. Strange or not, this was the natural way to implement it, and if we return to section 7.5, we can see that the class indeed creates only one port.

Hoping that the drawings gave some picture of what this is all about, we shall now go on to the `wbsingmaster` class.

7.7 The `wbsingmaster` class

This class creates a very simple Wishbone bus controller. It interfaces one Wishbone master with several Wishbone slaves, or any other port which can interface with a Wishbone master. This bus controller is not very useful for practical purposes, but it demonstrates the principles behind designing a real-life controller.

7.7.1 The `attempt()` method

At its first stage, this method only requires that the number of Wishbone masters in the port list will be exactly one. The next step is to list up the ports to be mapped on the bus in `@others`. Port of type `vars` are a special case, because they may be used to supply clock and reset to the bus, but they can also be mapped. Those of the first kind are kept in `@extras`, and the others are put in `@others`. The two kinds are distinguished by viewing the `labels` property of the `vars` ports:

```perl
sub attempt {
  my $this = shift;

  my @wbms =  grep {ref eq 'wbm'} @_;

  return undef unless ($#wbms == 0); # Single master, please

  my @vars =  grep {ref eq 'vars'} @_;
  my @extras = ();

  my @others = grep {((ref ne 'vars') && (ref ne 'wbm')) } @_;

  foreach (@vars) {
    my %l = $_->get('labels');
    if (grep /^(r|w|rw|wr)\d+$/, keys %l) {
      push @others, $_;
    } else {
      push @extras, $_;
    }
  }
```

```
return "Nothing to connect the master with"
  unless (@others);
```

At this stage we create an object of ourselves:

```
my $self = $this->new(nick => 'single_master_wb_controller');
```

Next, we verify that all of the ports that we intend to map on the bus have the two
necessary properties, `wb_adr_bits` and `wb_adr_select`, set up reasonably. Again,
note that if this is not the case, we don't create an error, but rather refuse politely to
do the job – and explain why.

```
my ($bits, $select);

foreach my $mate (@others) {
  my $bits = $mate->get('wb_adr_bits');
  return "Missing \'wb_adr_bits\' property on ".$self->safewho($mate)
    unless (defined $bits);
  return "'wb_adr_bits\' should be a decimal number and not ".
    $self->prettyval($bits)." on ".$self->safewho($mate)
    unless ($bits =~ /^\d+$/);

  my $select = $mate->get('wb_adr_select');
  return "Missing \'wb_adr_select\' property on ".$self->safewho($mate)
    unless (defined $select);
  return "'wb_adr_select\' should be a decimal number and not ".
    $self->prettyval($select)." on ".$self->safewho($mate)
    unless ($select =~ /^\d+$/);
}
```

The `prettyval()` method is used to create a human-readable string from its argu-
ment. It's described in section 9.1.20.

Next we create one Wishbone slave to match with the Wishbone master, and call
`intobjects()` to interface between the two. There is no reason why this should fail,
but we check anyhow.

```
my $myport = wbs->new(nick => 'wb_bus_controller_slave_port',
                      parent => $self);

my @objs = $self->intobjects(@wbms, $myport, @extras);
return "Failed to match ".$self->safewho(@wbms).
  " with the Wishbone given by the single-master bus controller generator\n"
    unless (@objs);
```

The following step is to go through each of the ports we want to map on the bus, create a Wishbone master to match with each, and call `intobjects()` for each pair. We are going to accumulate quite a few objects, so we start by making a list of what we've already got,

```
my @bunch = ($self, $myport, @objs);
```

and now we go for the loop:

```
foreach (@others) {
  $myport = wbm->new(nick => 'wb_bus_controller_master_port',
                     parent => $self);

  @objs = $self->intobjects($_, $myport, @extras);
  return "Failed to adapt ".$self->safewho($_).
    " to a Wishbone master, that was generated".
    "by the single-master bus controller generator\n"
    unless (@objs);
  push @bunch, $myport, @objs;
}
```

Note that all of the ports have the same `nick`. There is no problem with that, since the ports will get distinct names, if they'll reach the stage of being sustained objects.

Keep in mind that in the call to `intobjects()` we attempt to interface a Wishbone master with just any port type. The success of this call depends on the existence of adequate interface classes. For example, if we attempt to interface a `vars` port here, we expect to get three objects in return: The `vars2wbm` object, the port it created and the `wbsimple` object that was generated at the call to `intobjects()` by `vars2wbm`.

After each successful interfacing, the objects involved are listed up on `@bunch`. Note that the order of the elements of `@bunch` remains the order of the objects' creation (this is a must).

As usual, if the interfacing of one of the port fails, `wbsingmaster` gives up.

Finally, if the loop finished normally, we can happily announce a successful interfacing, and return a list of objects that were generated:

```
  return @bunch;
}
```

This is a nice point to remind ourselves that all this was the `attempt()` method: The smart work of mating ports, with all recursive calls involved, was done as part of giving an answer on if this class will or won't do the interfacing.

7.7.2 The `generate()` method

It may come as a surprise, that the implementation of the `generate()` method of this class is not significantly longer than the `attempt()` part. As we shall see, it is also less system-oriented and more into generating Verilog code which is, in fact, quite OK for the `generate()` method. As before, we break the code in pieces.

We start with the code's opening:

```
sub generate {
  my $self = shift;

  my @ports = $self->get('ports');
  my ($wbs) = grep {ref eq 'wbs'} @ports;
  my @wbms = grep {ref eq 'wbm'} @ports;
```

Here we pick up the ports that were generated during the `attempt()` method call. And then,

```
  my $obj = $self->whereto;
```

Note that unlike `vars2wbm`, there is no `codetargets()` method in this class. As a result, `whereto()` will return a value equal to `$self`. The use of `$obj` in the sequel allows to put the Verilog code in another object by extending the class with a `codetargets()` method, but as things are right now, the Verilog code will be written to the object's own `verilog` property.

7.7.3 Setting up the slave port

We now set up the `labels` property of the slave port. It's quite similar to what we had in the `vars2wbm` class (see sections 7.5.5 and 7.5.6).

```
  my %wbsNames = ();
  my %wbsIDs = ();

  ($wbsNames{'clk_i'}, $wbsIDs{'clk_i'}) =
      $obj->namevar('m_wb_clk_i', 'wire', 'in');
  ($wbsNames{'rst_i'}, $wbsIDs{'rst_i'}) =
      $obj->namevar('m_wb_rst_i', 'wire', 'in');
  ($wbsNames{'adr_i'}, $wbsIDs{'adr_i'}) =
      $obj->namevar('m_wb_adr_o', 'wire', 'in');
  ($wbsNames{'dat_i'}, $wbsIDs{'dat_i'}) =
      $obj->namevar('m_wb_dat_o', 'wire', 'in');
  ($wbsNames{'dat_o'}, $wbsIDs{'dat_o'}) =
      $obj->namevar('m_wb_dat_i', 'wire', 'out');
```

```
($wbsNames{'we_i'},  $wbsIDs{'we_i'})  =
    $obj->namevar('m_wb_we_o', 'wire', 'in');
($wbsNames{'stb_i'}, $wbsIDs{'stb_i'}) =
    $obj->namevar('m_wb_stb_o', 'wire', 'in');
($wbsNames{'cyc_i'}, $wbsIDs{'cyc_i'}) =
    $obj->namevar('m_wb_cyc_o', 'wire', 'in');
($wbsNames{'ack_o'}, $wbsIDs{'ack_o'}) =
    $obj->namevar('m_wb_ack_i', 'wire', 'out');

$wbs->const('labels', %wbsIDs); # Set up the labels for the port.
```

Note that the variable's name is somewhat confusing, with the `_i` and `_o` suffices reversed to the actual direction. While confusing to read this in the Perl code, it makes sense in the resulting Verilog code. For example, the label `adr_i` is the Wishbone *slave's* address input, but it's in fact the Wishbone *master's* output. The variable name `m_wb_adr_o` emphasizes that the variable is the Wishbone master's output, which makes it easier to understand when reading the Verilog code.

7.7.4 Retrieving bit range

In most cases, where we want the bit range of variables to be flexible, we use the fact, that when two variables are attached, their `dim` property is mutually copied. Sometimes this simple mechanism is insufficient, as in the current case: We need to make operations on some bits of the variable, so we need its bit range.

```
my ($realmaster) = grep {ref eq 'wbm'}
  $self->get('perlilog-ports-to-connect');
my %master_IDs = $self->labelID($realmaster);
my ($hi, $lo) = $self->bitrange($master_IDs{'adr_o'});

wrong("Expected hi-to-low bit range on adr_o of ".
    $realmaster->who()."\n") if ($lo > $hi);

wrong("Expected lowest bit in bit range on adr_o of ".
    $realmaster->who()." to be zero\n") unless ($lo == 0);
```

We start with looking up the (one) Wishbone master port, which is listed somewhere in `perlilog-ports-to-connect`. Following that, we create a hash of labels and IDs for that variable, and put it in `%master_IDs`. We then use the `bitrange()` method to get the values of the `dim` range. This method is a accepts a variable ID, looks up the variable's `dim` property, and returns the two numbers. For example, if `dim` was `[7:0]`, `$hi` would be set to `7`, and `$lo` to `0`.

And we only accept descending bit order, with the bottom at zero,, which is why the method complains otherwise.

Note that we needed to look up the Wishbone master's variable, although we've interfaced that master with our own slave. This interfacing assures that the master's `adr_o` will eventually have the same dimensions as the mated slave's `adr_i`, which happens to be referenced by $wbs. It may seem much simpler to get $hi and $lo from this slave. This wouldn't work, though: The equality of dimensions happens due to the `attach()` of these variables, which takes place only during the `generate()` call of the interface object, which mates the master and the slave. Now, since that interface object was generated *after* the current interface object, the `attach()` call hasn't taken place yet.

7.7.5 Some initializations

We now reach the actual implementation of the bus controller. We begin with setting up the %shorts hash, which connects between names of signals in the common slave (connected to our only master) and each of our masters. There are five such signals, that go directly with no logic inbetween:

```
my %shorts = (clk_i => 'clk_i',
              rst_i => 'rst_i',
              stb_o => 'stb_i',
              we_o => 'we_i',
              dat_o => 'dat_i');
```

Using this hash, we set up the first keys of the %mlabels hash, which will eventually be stored in all of our master ports as labels. For now, we set up the entries that are common to all of these ports. Note that by setting the entries in labels to the respective variable IDs, we actually make the connection. Or to be more precise, the connection will be done when each of our masters will be mated with the slaves.

```
my %mlabels = ();
foreach (keys %shorts) {
  $mlabels{$_} = $wbsIDs{$shorts{$_}};
}
```

Next, initialize a few Perl variables,

```
my ($bits, $select, $mate, $selcode);
my @acks = ();
my @dats = ();

my $main_ack = $wbsNames{'ack_o'};
my $main_dat = $wbsNames{'dat_o'};
my $main_cyc = $wbsNames{'cyc_i'};
my $main_adr = $wbsNames{'adr_i'};
```

and get on to the main loop.

7.7.6 The main loop

We go through each of our master ports:

```
foreach my $port (@wbms) {
```

7.7.7 The `mates` property

For each of these ports, we scan through the other ports it was mated with, to find the one with the `wb_adr_bits` and `wb_adr_select` properties set. Actually, we're not so interested in anything else about this mating port, except if we need to write an error message telling which one is badly set up. Otherwise, it's really only these properties we're after right now: The rest of the job is taken care of by some interface object.

```
MATE: foreach $mate ($port->get('mates')) {
    next MATE if (ref($mate) eq 'vars');
    $bits = $mate->get('wb_adr_bits');
    $select = $mate->get('wb_adr_select');
    last MATE if (defined $bits);
  }
```

We use the `mates` property, which is set by the PERLILOG environment after a successful interfacing (see section 13.1.2). This is a list of all ports (except the port to which the property belongs) that were given during the interfacing request. The order of this list is arbitrary.

Note that our `$mate` is not necessarily a Wishbone slave. Since our own class' call to `intobjects()` involved one of the ports that we were given during the call to our `attempt()` method, it could be of any type, as long as the interfacing with a master succeeded. The Wishbone slave, that maybe was squeezed in, will not appear in our mates list.

After finishing off this loop, we check up what we got, and complain if necessary. Note that here we create real errors, not polite `return` strings. This is `generate()`, not `attempt()`.

```
wrong("Failed to find \'wb_adr_bits\' property on any of ".
      join(', ', map {$_->who} $port->get('mates'))."\n")
  unless (defined $bits);

wrong("Found \'wb_adr_bits\' property, but not \'wb_adr_select\' on ".
      $mate->who()."\n")
  unless (defined $select);
```

```
        wrong("Faulty \'wb_adr_bits\' property ".$self->prettyval($bits).
            " on ".$mate->who()."\n")
        unless ($bits =~ /^\d+$/);
```

7.7.8   Creating the slave selector

By now, `$bits` contains the number of address bits that our slave needs. The lowest bit to use compare with `wb_adr_select` is therefore `$bits`. We can now generate a snippet of Verilog code, which defines the condition, on which the current port should be selected:

```
wrong("Range exceeds dimensions, because \'wb_adr_bits\' on ".
    $mate->who()." was too large\n")
  if (($hi + 1) < $bits);

if ($hi < $bits) {
  $selcode = "1";
} else {
  $selcode = $wbsNames{'adr_i'}.'['.$hi.':'.$bits.'] == '.$select;
}
```

If the `wb_adr_bits` demanded all address bits, our port is always selected, and `1` is the correct logic expression. Note that we've already checked that `$hi` is not smaller than `$bits` by more than one, so if it is smaller, it's exactly by one, and thus it's exactly the case of a port wanting all address bits.

7.7.9   Variables and `samedim()`

At this stage we create variables to bring to our Wishbone master port. We start with the address lines.

```
my $adrhi = ($bits==0) ? $hi : ($bits - 1);
my ($adrvar, $adrID) =
  $obj->namevar('wb_slave_adr', 'wire', 'out', "[$adrhi:0]");
$mlabels{'adr_o'} = $adrID;
```

The variable labeled `adr_o` will have `$adrhi`, plus one, bits. If `$bits` is zero, the address is not interesting – the slave is enabled with control lines only. In this case, we pass the complete address to the port, just to make it look nice (it makes no difference anyhow).

Note that we store the variable's ID in the hash, and not its name.

We now create two other variables, and save map their IDs in `%mlabels`.

```
my ($cycvar, $cycID) = $obj->namevar('wb_slave_cyc', 'wire', 'out');
$mlabels{'cyc_o'} = $cycID;

$obj->samedim($cycvar, $main_cyc);

my ($ackvar, $ackID) = $obj->namevar('wb_slave_ack', 'wire', 'in');
$mlabels{'ack_i'} = $ackID;
push @acks, $ackvar;

$obj->samedim($ackvar, $main_ack);
```

Note that we pushed our `$ackvar` into the `@ack` list. It will be used later.

The `samedim()` is like `attach()`, only it doesn't connect between the variables. What is does, is to assure that the variables have the same dimension property, as if they were `attach()`'ed.

Calling `samedim()` is highly recommended every time we make assignments in injected Verilog, whether it's with Verilog's `assign` or with procedural assignments. This holds true when we implicitly mean that the variables involved should have the same dimensions, as in this case (see below).

Whenever the relations between the dimensions of a couple of variables are not equality, it's recommended to take the time to formulate the exact magic callbacks needed to set these relations.

7.7.10   Slave-specific Verilog code

Here we generate some Verilog code, which is specific for this certain port. Note that even though `assign` allows the variables to be of different dimensions, we've used `samedim()` to eliminate many possibilities to do stupid things (not all, though).

```
my $selvar = $obj->namevar('wb_slave_active', 'wire', 'out', '');
$obj->append("   assign $selvar = $selcode;\n".
             "   assign $cycvar = $main_cyc & $selvar;\n".
             "   assign $adrvar = $main_adr".'['.$adrhi.":0];\n");
```

Also note that everything is done in the context of `$obj`: For example, `$selvar` is a name of the variable created by `namevar()` which was called from `$obj`, and this variable is used in Verilog code that is planted into the same object's Verilog code. Its important to keep track of these things, because otherwise everything will work fine until the choice of hosting object changes suddenly.

We go on with creating a `dat_i` variable for our master, which is soon to be connected to the slave's `dat_o`. Now, all of these are going to be "wire-ORed" to the main master's `dat_i`, so we create an expression which equals `dat_i` when our slave is selected, zero otherwise. And push it into `@dats`, to be used soon. Note the `samedim()`.

```
my ($datvar, $datID) = $obj->namevar('wb_slave_dat', 'wire', 'in');
$mlabels{'dat_i'} = $datID;
push @dats, "($selvar ? $datvar : 0)";

$obj->samedim($datvar, $main_dat);
```

7.7.11 Setting up `labels`

We finish the handling of each port by setting the port's `labels` property with the data
in the %mlabels hash. Note that some of the labels were set for all ports before the
loop, and others are changed during the loop to point to the port's own variables. This
is not a problem, since we pass the hash' content (and not a reference).

```
    $port->const('labels', %mlabels); # Set up the labels for the port.
}
```

7.7.12 Grand finale

Outside the loop, there are a few final things to do before finishing the method. We
create a common acknowledge signal, which is all acknowledges ORed. The main
master's `dat_i` variable is given as the OR of some previously created expressions.

```
my $allacks = join(' | ', @acks);
my $alldats = join(' | ', @dats);

$obj->append("  assign $main_ack = $allacks;\n".
             "  assign $main_dat = $alldats;\n");
}
```

7.7.13 What we didn't have in this class

There are two things that some of us may find missing in this class:

- Note that we don't verify whether the slaves are mapped correctly on the bus.
  If one or more slaves are mapped on the same addresses, corrupted code will
  be generated, and no warning will be given about that. There is no excuse for
  being this sloppy, other than that this is merely a demonstration class. In real-life
  bus controllers a sanity check, at least at the end of operation is a must. A nice
  memory map, as a text file, is a nice add-on.

- There is no `codetargets()` method defined in this class. This is perfectly OK,
  and will cause the class to generate an independent file. Just in case someone
  wondered why it isn't there.

7.8   The elements of an interface class

OK, enough examples. We can now summarize the general guidelines of writing an interface class.

7.8.1   The common methods

Usually, an interface class supplies two or three methods:

- `attempt()` – The system queries the class by calling this method. The class responds whether it can interface between a list of ports.

- `generate()` – This method functions is the same way as any code generating class: It's the execution of the object's purpose. Unlike regular code generating objects, the interface object may put its Verilog code in other object's `verilog` property, and thus not generate a file of its own.

- `codetargets()` – This method is highly optional, and should be implemented when the Verilog code is intended to be written to other objects. The purpose of this method is to *recommend* to which object the Verilog should be written to. The `codetargets()` is explained in section 7.5.10.

Apart from these methods, any other methods that make sense to a code generating object may be extended or overridden.

If the `new()` method is extended in an interface objects, the guidelines given in section 7.10 should be followed.

7.8.2   The elements of the `attempt()` method

The common guideline for the `attempt()` method is to do the minimum necessary to be able to give a reliable yes or no regarding if the current class is ready to interface between the given set of ports.

The recommended order of things will usually be some of the following steps:

1. Check up the port types with `ref`, and see if they are of the right kind and number. If we fail here, return `undef` – it was probably meant for another class to take care of.

2. Make additional checkups to see if the ports are adequate – usually this means verifying that the ports' properties are set up as necessary. If not, return a string describing what seems to be wrong.

3. Create a "self" object, by calling `new()` on the current class.

4. Create new ports that belong to the "self" object.

5. Interfacing those ports with other ports, most probably to those given to `attempt()` as arguments in a manner that makes sense.

Depending on the nature of the interface class, only some of the steps above will be relevant. The order in which the steps are presented is recommended, though.

There a few important rules for writing `attempt()` method that are derived from the fact that this method is called during "transient mode". These are given in section 7.10.

7.8.3 The elements of the `generate()` method

The `generate()` method is the essence of the class, and it's executed when there is no question regarding the choice or willingness of the current object. The guidelines are therefore looser. This method usually comprises of some or all of the following items, not necessarily in this order:

- A call to `whereto()` to determine which object will host the Verilog code and its extra variables.

- Resolving the IDs from the ports' `labels` properties, preferably using `labelID()`.

- "Connecting" variables (as with copper wires) using `attach()` as adequate.

- Generating new variables with `addvar()`, or more comfortably `namevar()`. These variables are used in the Verilog code to be generated and/or appear in the `labels` property of some ports.

- Creating read-only copies of a variable using `copyvar()`. This is useful when we want the variable's value in a Verilog expression, and no other manipulation is necessary.

- Updating or setting up `labels` properties of the ports that we created during the `attempt()` call.

- Appending Verilog code to the hosting object. Possibly using `samedim()` in conjunction with assignments.

Note that generating new objects (including ports) during the `generate()` method call is highly unrecommended, and should be done during the `attempt()` call. Even better, if the objects are not needed to complete the `attempt()` method, they should be generated as part of the extension of the `sustain()` method (see section 7.10.4 for more about this).

7.9   How PERLILOG uses interface classes

Until now we've studied the interface classes, but said very little about how they fit into the system. We shall now see how the PERLILOG environment uses the interface classes.

7.9.1   Registration of interface classes

The PERLILOG recognizes a class as an interface class as a result of a call to `interfaceclass()`. The argument to this system subroutine must be the name of a registered class. The intended way to do this is something like this: (code taken from the system's init file, see section 4.2 for more about init files)

```
&Perlilog::inherit('vars2vars', "${home}vars2vars.pl", 'interface');
&Perlilog::interfaceclass('vars2vars');
&Perlilog::inherit('wbsimple', "${home}wbsimple.pl", 'interface');
&Perlilog::interfaceclass('wbsimple');
&Perlilog::inherit('vars2wbm', "${home}vars2wbm.pl", 'interface');
&Perlilog::interfaceclass('vars2wbm');
&Perlilog::inherit('wbsingmaster', "${home}wbsingmaster.pl", 'interface');
&Perlilog::interfaceclass('wbsingmaster');
```

The variable `$home` is the path to where these classes files reside. As for the explicit package declaration (the `&Perlilog::` prefix), it can be omitted when calling these routines from the main script, but not from within a method (the init script is given as a method named `init()`).

7.9.2   Quering order of interface classes

The order in which the classes are registered with `interfaceclass()` is the order the classes are `attempt()`-queried. There is no rigid definition on what the quering algorithm should do when some interface class responds positively: It may go on asking other classes for their bids, or stop and choose the first positive answer. The algorithm may be changed to fit different needs and logic architechtures. For simple outlines, picking the first positive answer will usually be enough.

The order in which the classes are registered is crucial. Vast differences in the efficiency of the resulting Verilog code may be the result of changes in either the class register ordering or the algorithm. Even so, choosing the order of registering wisely is probably enough to achieve best results, even for the most complex set of classes. And when done unwisely, the results will be ugly enough to draw necessary attention.

The aim is to make the process of choosing the interface class quick and fruitful. This is achieved by following the following guidelines, which in essence say that a simple class should be tried before going for complex ones:

- Register simple, one-on-one interface classes first (such as `vars2vars` and `wbsimple`). Classes that connect pairs or ports reject inadequate attempts very quickly, and if they respond positively, they will usually supply the obvious solution.

- All classes, which don't use `intobjects()`, should be registered *before* those who do.

- Register "bridges" or "adapters" only after all classes that make a simple one-on-one connection. For example, `vars2wbm` is registered after `vars2vars` and `wbsimple`. Otherwise, we can get a back-and-forth adaption of ports, where a simple connection was possible.

- "Controllers" and any interface class which is intended to interconnect several ports should be registered last.

7.9.3 `interface()` **vs.** `intobjects()`

As we've seen, the `interface()` function and the `intobjects()` method have been used to do basically the same thing: Request to interface a list of ports. Calls to `interface()` appeared in main scripts, while `intobjects()` was used in implementations of `attempt()`. In a nutshell, this is the difference between them: How and when they are used.

As a result, we have the following differences:

- Return value: `interface()` returns the top-level interface object, `intobjects()` returns a list of objects involved in the interfacing.

- If the interfacing fails: `interface()` issues a fatal error, while `intobjects()` returns an empty list. It's considered normal that `intobjects()` fails.

- Recursivity: `interface()` must not be called while handling an interface request (which boils down to not calling `interface()` from an `attempt()` implementation). `intobjects()`, on the other hand, is intended to be used from within `attempt()` methods.

The `interface()` routine can be seen as a wrapper of `intobjects()`: `interface()` invokes transient mode (see section 7.10), calls `intobjects()` and views the return value. If `intobjects()` returned a list of objects, it calls `sustain()` on all these objects, and returns the first object in the list (which is the top-level interface object). Otherwise, a long error message is generated and issued.

7.9.4 `intobjects()` **and recursive calls**

As we've seen, `intobjects()` can be used within an `attempt()` method, which is a request to interface ports within a handler of such a request. This causes a hazard of

getting caught in infinite recursion, especially if the classes are queried in the wrong order or a critical class is missing. For example, if `wbsimple` is not registered as an interface class, no other class will agree to interface a Wishbone master and slave pair. As a result, the `wbsingmaster` bus controller class will eventually be queried. But this class will create a master and a slave, and attempt to interface each of these with the given two ports. So we're back with a request to the PERLILOG environment, but now we have two master-slave pairs to interface, and still all classes answer negatively, except `wbsingmaster`, which will try again...

Fortunately, in this specific case there is no risk for infinite recursion, because just a master and slave pair don't have the necessary properties to be accepted by the `wbsingmaster` class, so the recursion will stop quite soon, and an error message will be issued.

Besides, there is a PERLILOG mechanism to avoid infinite recursions: The recursion depth is kept track of, and it will not exceed the limit given in the `MAX_INTERFACE_REC` property of the global object, which defaults to 5. If this recursion level is reached, `intobjects()` fails promptly, and a warning message is displayed. Note that this level may be reached in very complex interface structures, but if the warning message is displayed, something is probably wrong with the interface classes.

7.10   Transient objects

7.10.1   Why transient objects

We've seen that `attempt()` methods may create several objects, which may turn out to be useless as a result of a failure to interface ports (even though the overall effort to interface the ports succeeded). As a result, the PERLILOG needs a means of getting rid of these objects, without leaving any traces.

The problem is that when normal objects are created, their references are kept in a few data structures to make them easier to access, and to make them callable during the execution process. This is not desirable for a lot of objects that are part of a failed attempt to interface some ports.

This has been solved by the concept of transient objects. In order to clarify what they are, we need to know a bit about what objects in general are in Perl.

A PERLILOG object is in fact normal Perl object, which in turn is a reference to a "blessed" hash. The issue of blessing is not important here, but rather the fact that the actual creation of an object is more or less creating an empty hash, and starting to use the reference to it. The "blessing" is what makes it related to a class, so with normal Perl objects, this reference to the hash is a full-powered object with methods and properties and all the goodies.

The PERLILOG environment extends this Perl mechanism in several ways. Among these extensions, we have a the registration of new object in some PERLILOG data structures.

The trick about transient objects, are that they become normal PERLILOG objects in two stages: The first stage is their creation (using `new()`) during which a common-Perl object is generated, and some basic properties are set. Then, if we want to keep the object in the system, the object's `sustain()` method is called, which turns it into a perfectly normal PERLILOG object.

Should we want to discard the object between the two stages, all we need to do is to forget about it. More formally put, we should make the object unreachable by not keeping the value of its reference anywhere. This happens naturally when the reference is only kept in a `my`-localized Perl variable within a method, and the method returns. The variable goes out of scope, and the object is destroyed by the Perl garbage collector.

7.10.2 How it works

When `interface()` is called, the PERLILOG environment invokes a special "transient mode" until a decision has been made regarding which objects are going to do the interfacing. In other words, transient mode is invoked just before `interface()` calls `intobjects()` end exited after `intobjects()` returns.

During this mode, any object that is created with `new()` will be considered "transient" until the object's `sustain()` method is called. Transient object, as their name implies, may have a very short lifecycle, and they are therefore not registered to be recognized by the system until `sustain()` is called, if at all. Effectively this means that transient objects are not called to during the multi-stage execution process. Neither do they have a `name` registered, so if the object's reference isn't explicitly kept in some variable, the object becomes unaccessible (and should therefore be destroyed by the Perl garbage collector).

The differences of calling `new()` during transient mode are as follows:

- A transient object is generated, rather than a regular one.

- The `nick` property must be set during the creation, and `name` must *not* be set. If the object is sustained, the property `name` will be set automatically by the system. `nick` will be used as an argument to `suggestname` so its value doesn't need to be unique in any way (as opposed to `name`). Several objects may be created with the same `nick`. Note that if the object is sustained, its `name` may be slightly different from `nick`.

- If the object's `parent` property is assigned, the parental relations will not be set up until the object is sustained. In other words, the object will not be added to the parent's list of children (or ports) until `sustain()`.

- The object is not registered to be called to during the execution stages, until it's sustained.

- The `perlilog-transient` property is set to `'transient'`, to be changed to `'sustained'` if the object is sustained.

This transient mode is implemented in the root class' methods `new()` and `sustain()`. There is no need to implement those methods explicitly. In particular, if the `new()` method is extended (overridden and called with SUPER::), the restrictions mentioned below for `attempt()` apply to the extension.

The `interface()` function sustains some objects, and discards the rest, before exiting the transient mode ("discarding" here means that the objects will be unreachable, unless their references have been stored, which is against the guidelines below). We may thus conclude, that transient objects effectively exist only while the system is in transient mode.

After an object has been sustained, it behaves exactly like a normal object. Thus, when implementing methods to be called during the multi-stage execution, there is no need to take the issue of transient objects into consideration.

7.10.3 Guidelines for correct code

The following guidelines apply when writing code which runs prior to `sustain()` (usually `attempt()` or extension of `new`):

- The code should run quickly, since it may be called several times due to the recursive nature of the interface class finding mechanism.

- Assume that the object will be discarded immediately upon return from the method written. Accordingly, minimal traces should be left in data structures beyond the object's own property table.

- If the object creates other objects during transient mode (in particular in the `attempt()` method), their references must not be stored in the object's property structure nor in any other place in the system. Generally, an object's property structure must not contain references to objects that were created after the object itself. The same applies for the self-reference (commonly stored in `$self`), which mustn't be stored in any way that is accessible after the method call terminates.

  This ensures that Perl's garbage-collection mechanism will work properly on discarded objects.

7.10.4 The `sustain()` method

Exactly as the built-in `new()` method is written to support transient object, so is `sustain()` written to perform the necessary actions to make a transient object permanent. Usually there is no need to extend the `new()` method, and by the same logic, the `sustain()` method can be left as is.

But if some actions need to be performed during the very early stages of an object, extending `new()` and/or `sustain()` is possible. The extension of `new()` is the same

as any object. Extending `sustain()` is like any method on an existing object (unlike `new()` which is called before the object exists).

The choice of which of these method to extend depends on how early we need our task to take place. Whenever possible, extending `sustain()` is preferred, since it frees us from restrictions related to the transient state of the object. `new()` should be extended only when `sustain()` isn't early enough.

At any rate, extending or overriding one of the execution stage's method is always better, when possible. The fact that the surrounding objects may not have set up all their properties should be taken into consideration, and therefore a late execution may be preferred.

7.11 Advanced issues

7.11.1 Dispatch classes

As we've seen, the normal way things work with interface classes, is that each class responds for itself, and generates an object of itself. Each of the classes looks at the ports, and makes its own decision during the `attempt()` call. The idea behind this scheme is to keep the system modular.

But there may be cases, where it is much easier to look at the ports once, and decide which class should take the job, rather than letting each class accept or decline on its own behalf. This is expected to be more and more the case, as the number of ports types grows, and the number of possibilities grow accordingly: The decision of how to adapt between different port types may be easier formulate as a single decision, taken at a single call to `attempt()`.

This can be done in PERLILOG by having a special "dispatch class". A dispatch class is a class which never generates objects of its own, but only has an `attempt()` method. Upon success, will the `attempt()` method generate an interface object of some *other* class or classes. It may thus wisely choose the classes to take action according to the situation with the port types. This is useful when having a bundle of interface objects that we want to manage in a central manner.

In the examples we've had, the interface object generated as in `my $self = $this -> new(nick => 'TheNick')`, but there is no reason why the interface object we generate in the `attempt()` method shouldn't be of any other class. The only thing to be careful about, is that the object we create is indeed adequate for the job.

For example, we could create a single class to take care of all interface classes which involves Wishbone ports. This class would choose which class takes the job out of the four presented.

Now, If an interface class is created (with all desired functionalities) by a dispatch class, we don't want it to be queried with `attempt()`. In that case, it should not be registered with `interfaceclass()`, and there is no need to write a `attempt()` method

for it, since it will never be called. Naturally, the dispatch class must be registered instead.

Note that a dispatch class is merely a way to concentrate several `attempt()` calls into one. If there are cases, in which the dispatch class doesn't offer an interfacing solution, it's perfectly OK to register other classes (usually after it), which take care of these cases. Dispatch classes do not restrict expandability.

7.11.2 Static objects in interface objects

Sometimes, we want the interface object to generate several modules. For example, a complex bus controller may already be written in several modules of plain Verilog. The easiest way to adapt this bus controller to PERLILOG may be to generate only one module by a script, and incorporate the rest as static objects. This has the following advantages:

- The effort is minimal: Most of the Verilog code remains completely untouched.

- The number of Verilog files is reduced if more than one interface object (of the same kind) is generated, since the interface objects use the same modules.

There is a small catch with creating static objects in interface classes: Since we can't touch static object's Verilog code, we can't change the module's name in any way. In order to avoid name collisions in the bundle, we have to make sure that every Verilog module appears only once.

In main scripts, this is quite easy to ensure this, simply by generating a single static object for the Verilog code. With interface objects, it's a bit more tricky, because the system creates as many as needed of them. So obviously we need some mechanism to recognize that a similar static object has already been generated.

The straightforward way of doing this, is by using the global object, as follows:

```
sub sustain {
  my $self = shift;
  $self->SUPER::sustain(@_); # Don't forget this!

  my $global = $self->globalobj();

  unless ($global->get(['static-done', 'jumper'])) {
    my $jumper = static->new(name => $self->suggestname('jumper'),
                             source => 'jumper.v');
    $global->const([''static-done', 'jumper'], $jumper);
  }
}
```

The heart of this little routine, is where we check up if the `jumper` routine exists under the path `static-done` in the global object's property tree.

Aside from that, there are several points to note here:

- We couldn't create the static object during the `new()` method, because the object was transient at the time. Hence, it would be a mistake to write to the global object's property table.

- The static object had to be generated before the call to `execute()` by the main script, so we couldn't do this in one of the execution methods.

- The conclusion is that the only way to generate static objects (or any other object that we want globally once for an interface class) is to implement it in the `sustain()` method.

- We can't allow automatic instantiation, because the module may be instantiated more than once.

- There is nothing system-special about the name `static-done`, or the path to that property. These names just lessen the chance of name clashes between properties.

- Calling the inherited method with `SUPER::` is very important. Failing to do so will cause an error message which is unrelated to the real problem.

And then, somewhere in the implementation of `generate()`, we may have, once or more, something like:

```
my $insname = $obj->suggestins();
$obj->addins($insname, 'detached');
$obj->append("  jumper $insname(.one($one), .two($two));\n");
```

There is more about static object in section 6.9.

7.11.3 Objects that don't generate files

A verilog file will be created if one or more of the following conditions are met on the code generating object:

- It has at least one variable declared (in the `vars` property tree branch).

- It is the parent of at least one object

- It is static

Interestingly enough, it doesn't matter if the object's `verilog` property has something in it or not. The logic is like this: If it isn't static, then we know what happens with it. So if it has no variables, it will have no inputs or outputs. So if it doesn't instaintiate anything, what use does it have?

Usually, the only code generating objects that don't have a Verilog file are interface objects that either generated no Verilog at all, or put their Verilog code in some other object's `verilog` property.

The fact that files are not always generated can make PERLILOG appear as if it was buggy, especially in small test scripts: If a Verilog object is generated, and its `verilog` property is set, it will still not generate a file if nothing useful was done with it.

See section 11.2.7 regarding how to check up if a file is going to be generated.

8   PERLILOG main script API

8.1   Exported subroutines

8.1.1   The exported subroutine `init()`

Synopsis:
> `init;`

Syntax:
> `init;`

Description:
> `init` must be executed before objects are created, and after the class tree is defined
> by using `inherit`, `override` and `underride`.
> The routine sets up the global object and other environmental items.

Return value:
> Not to be used

Example:
> See the example in section 8.1.2

8.1.2   The exported subroutine `inherit()`

Synopsis:
> `inherit('myclass', 'myclass.pl', 'root');`

Syntax:
> `inherit(`name of new class`, `Perl file`, `class to inherit`);`

Description:
> The inherit routine registers a new class into the class tree. This registration includes
> giving the name of the new class, the Perl file which includes the class' methods, and
> the class from which methods will be inherited.
> The Perl file is not read during the execution of `inherit`, nor is there need for the class
> to inherit to be declared when the call to `inherit` is performed. `inherit` only verifies
> that the class that is declared doesn't exist already, and that the third argument is

given.

A detailed description of class declaration is given in section 6.4.1.

Return value:

Always returns 1

Example:

```
use Perlilog;

inherit('class_a', 'myclass_a.pl', 'root');
inherit('class_b', 'myclass_b.pl', 'class_a');

init;

print "--- Now playing with object A --\n";
$objA = class_a -> new(name => 'AObject');
$objA -> Asayhello();
$objA -> benice();

print "--- Now playing with object B --\n";
$objB = class_b -> new(name => 'BObject');
$objB -> Asayhello();
$objB -> Bsayhello();
$objB -> benice();
```

We assume that the file `myclass_a.pl` is:

```
sub Asayhello {
  my $self = shift;
  print "This is class A as ".$self->who()." saying hello\n";
}

sub benice {
  my $self = shift;
  $self -> SUPER::benice(@_);
  print "This is class A saying hello after being nice\n";
}
```

and `myclass_b.pl` is

```
sub Bsayhello {
  my $self = shift;
  print "This is class B as ".$self->who()." saying hello\n";
}

sub benice {
```

```
  my $self = shift;
  $self -> SUPER::benice(@_);
  print "This is class B saying hello after being nice\n";
}
```

The script will thus print out:

```
--- Now playing with object A --
This is class A as object 'AObject' saying hello
This is class A saying hello after being nice
--- Now playing with object B --
This is class A as object 'BObject' saying hello
This is class B as object 'BObject' saying hello
This is class A saying hello after being nice
This is class B saying hello after being nice
```

We note that since class B inherited methods from class A, we could call both `Asayhello()` and `Bsayhello()` on object B. Furthermore, when calling the method `benice()` on object B, class A's `benice()` was called as well by virtue of `$self -> SUPER::benice(@_)`.

8.1.3 The exported subroutine `inheritdir()`

Synopsis:
> `inheritdir('classdir', 'root');`

Syntax:
> `inheritdir(`path to class directory`, `root class to inherit from`);`

Description:
> `inheritdir` is a means of loading a library of classes, with predefined relations of inheritance between them. `inheritdir` scans the given directory for files with `.pl` extension, and executes `inherit` on each of these files. The file name, without the `.pl` extension, is taken as the new class' name for each file.
>
> `inheritdir`'s second argument is the class to inherit methods from.
>
> If directories are found among the scanned class files, each directory is scanned as well. The name of the directory is then the class from which all classes will inherit methods.
>
> For example, we could get the same class structure as in the example of section 8.1.2 by having a directory named `classdir`. This directory would contain the `myclass_a.pl` file and another directory, named `myclass_a`. Now, the `myclass_a` directory would contain one file, `myclass_b.pl`.

We then run `inheritdir('classdir', 'root')`, `myclass_a.pl` is read and creates the `myclass_a` class. The `myclass_a` directory is found, and `myclass_b.pl` within it. Now, since the directory is named `myclass_a`, the `myclass_b` class inherits methods from `myclass_a`.

Note that if we have a directory named `foo`, then a class file `foo.pl` and the directory `foo` must be in the same (parent) directory. This structure resembles the way sub-modules are organized in plain Perl, and it forces the directory tree to reflect the class tree.

Also note that all file names and directory names are translated to lowercase, resulting in only lowercase class names.

Return value:
Always returns 1

8.1.4 The exported subroutine `override()`

Synopsis:
```
override('theclass', 'myclass.pl');
override('theclass', 'myclass.pl', 'root');
```

Syntax:
```
override(name of class, Perl file[, class to inherit]);
```

Description:
`override` causes the given Perl file's method declarations to override those of the already declared class. Note that unlike common practice in object-oriented programming, this allows to override particular methods without changing the class' name, despite the fact that strictly speaking, this operation generates a new class.
If the class, which is named in the first argument doesn't exist, `override` behaves exactly like `inherit`, and thus a third argument is needed. This third argument is optional, and functions as a "backup" in case the desired class does not exist, and hence needs to be generated with `inherit`.
A detailed description of class declaration is given in section 6.4.1.

Return value:
Always returns 1

Example:
```
use Perlilog;

inherit('class_a', 'myclass_a.pl', 'root');
override('root', 'myclass_b.pl');

init;
```

```
$objA = class_a -> new(name => 'AObject');
$objA -> Asayhello();
$objA -> Bsayhello();
$objA -> benice();
```

We assume that the files `myclass_a.pl` and `myclass_b.pl` are the same as in the example for `inherit` in section 8.1.2. This will print out:

```
This is class A as object 'AObject' saying hello
This is class B as object 'AObject' saying hello
This is class B saying hello after being nice
This is class A saying hello after being nice
```

This example shows an override of the root class, which will affect all classes in the system. It's usually not necessary to go that deep down.
Note that the root class was overridden after the declaration of `class_a`, and still `class_a` inherited the methods from `myclass_b.pl` via the inheritence from the root class.

8.1.5   The exported subroutine `underride()`

Synopsis:
> `underride('theclass', 'myclass.pl');`

Syntax:
> `underride(`<u>name of class</u>`,  `<u>Perl file</u>`);`

Description:
> `underride` works like `override`, only in the opposite way: It will give the Perl file's method the lowest predecense in the inheritence chain. In other words, the class tree will be set up like it would if the current Perl file was the one that generated the class, and all other declarations of the same class came afterwards as `override()`s.
> If another `underride()` is called twice on the same class, the second one will be closer to the root class.
> This routine is merely intended for debugging purposes, and is not recommended for standard use.
> A detailed description of class declaration is given in section 6.4.1.

Return value:
> Always returns 1

8.1.6   The exported subroutine `definedclass()`

Synopsis:

```
$status = definedclass('class');
```

Syntax:

<u>class status</u> = `definedclass`(<u>class name</u>);

Description:

`definedclass` accepts a class name as argument, and returns a value that reflects the class' status.
A detailed description of class declaration is given in section 6.4.

Return value:

`definedclass` returns `0` if the class has not been defined. `1` is returned if the class is defined, but its Perl code has not been loaded. `2` is returned if the class is defined, and the Perl code has been loaded.

Example:

```
use Perlilog;

print "In the beginning, the status was ".definedclass('class_a')."\n";

inherit('class_a', 'myclass_a.pl', 'root');
print "Afterwards, the status was ".definedclass('class_a')."\n";

init;
print "After init the status was ".definedclass('class_a')."\n";

$obj = class_a -> new(name => 'TheObject');
print "After usage the status was ".definedclass('class_a')."\n";
```

This will print:

```
In the beginning, the status was 0
Afterwards, the status was 1
After init the status was 1
After usage the status was 2
```

Note that the status remained `1` even after `init`: The class' Perl code was read only when an object was generated from the class.

8.1.7   The exported subroutine `globalobj()`

Synopsis:
> `$GlobObj = globalobj;`

Syntax:
> object reference = `globalobj;`

Description:
> `globalobj` returns an object reference to the global object. It must not be called before
> `init()`, since the global object doesn't exist before that.
> More information about the global object is given in section 5.4.5.

Return value:
> A reference to the global object.

8.1.8   The exported subroutine `execute()`

Synopsis:
> `execute();`

Syntax:
> `execute();`

Description:
> This routine calls the `execute` method of the global object. This will kick off the code
> generation process. See section 6.5.

Return value:
> Not to be used

8.1.9   The exported subroutine `interfaceclass()`

Synopsis:
> `interfaceclass('myclass');`

Syntax:
> `interfaceclass(`name of interface class`);`

Description:
> `interfaceclass()` registers some class as an interface class. The class must have
> been declared with `inherit()`, `override()` or `underride` prior to registration with

> `interfaceclass()`. See section 7.9 for more about this.

8.2 The global variables

8.2.1 The variable `$VERSION`

Description:
> The variable `$VERSION` is the version number. It is declared as is commonly done in Perl modules. Note that it can be used as a number.

Example:

```
use Perlilog;

if ($Perlilog::VERSION < 1.00) {
  print "We are running on a pre-release version!\n";
}
```

8.2.2 The variable `$globalobject`

Description:
> A reference to the global object is stored in `$Perlilog::globalobject`. The value of this variable is returned when calling the `globalobj()` routine.

8.2.3 The variable `%classes`

Description:
> This hash's keys are names of classes. To be more accurate, these are the names of the Perl packages that will be generated during the loading of class Perl files. The values are either references to lists, or the scalar value `1`. The latter signifies that the class has been loaded (`definedclass` would return `2`).
> When the value is a reference to a list, it is a list of three items: The first item is the name of the Perl file associated with the class. The second is the class which the current class should be derived from (in other words, the value of this class' `@ISA`). The last item is the class name that was used when creating this class. It may be different from the package name due to class override.

8.2.4 The variable `%objects`

Description:
> This hash links between object names and their references. The keys are names of objects, and the values are their references.

The integrity of this hash may be verified by calling the `treestudy` method (see section 9.1.22.

8.2.5 The variable `@VARS`

Description:

`@VARS` is a list of references to variable pointers, which are lists consisting of two items: The first item is a reference to the object holding the variable, and the second is the variable's name.

The variable's ID is the index to the variable's entry in `@VARS`.

The first to items of `@VARS` are `undef` as a measure of avoiding the indeces `0` and `1` to acciendly point to variables.

8.2.6 The variable `@EQVARS`

Description:

`@EQVARS` is a list of references to equivalence lists. An equivalence list is a list of variable ID's, whose respective variables are equivalent in such a way that they are connected in the Verilog sense.

8.2.7 The variable `@interface_classes`

Description:

This is a list of interface classes that were registered with `interfaceclass()`, in the order that they were registered. See section 7.9 for more about this.

8.2.8 The variable `$interface_rec`

Description:

If `$interface_rec` is defined, the system is in transient mode (see section 7.10), and its value is a number, which is the number of recursive calls to `intobjects()` calls that the system is ready to get deeper into recursion.

When the system is not in the middle of handling some call to `interface()`, this `$interface_rec` is undefined. In order to keep the system sane, this variable, should be only read from (except for the core of PERLILOG).

9 The root class API

9.1 Methods

9.1.1 The method `new()`

Synopsis:
```
$theobj = theclass -> new(name=>'thename');
```

Syntax:

<u>class</u> -> `new`(<u>property hash</u>);

Description:

The `new` method creates a new object of a given class. Its initial properties are passed as a hash.

The `new` method must be called with at least one property, the `name` property set. The value of `name` property must be different from that of any other previously created object, even when making case-insensitive comparison.

If the `parent` property is set in the property hash, `addchild` will be called in order to set the correct relations.

Return value:

A reference ("handle") to the new object.

Example:
```
use Perlilog;
init;
$obj1=root->new(name=>'theObject');
$obj2=root->new(name=>'anotherObject',
                parent=>$obj1);
```

Note that that `parent` is assigned `$obj1`, and not the parent's name.

9.1.2 The method `set()`

Synopsis:
```
$obj->set($property, $scalar);
$obj->set($property, @list);
```

```
$obj->set(\@path, $scalar);
$obj->set(\@path, @list);
```

Syntax:

object -> set(property, new value);

Description:

The set method sets the value of a property. If property needs not to exist prior to calling set, but it must not have been created by const (see section 9.1.4).

The value is in general a list. Scalars are handled as lists with a single item. Even so, the set and get pair can be used with scalars in a straightforward way, as is shown in the example of section 9.1.3.

To delete a property, set it to undef.

If the property name is a reference to a list, this is considered as the property's path. Note that paths are easily expressed with square brackets, [and] (see example). Paths and their rules are described in section 5.3.6.

Return value:

Always returns 1

Example:

See the example in section 9.1.3

9.1.3 The method get()

Synopsis:

```
$scalar=$obj->get($property);
@list=$obj->get($property);
$scalar=$obj->get(\@path);
@list=$obj->get(\@path);
```

Syntax:

object -> get(property);

Description:

The get method looks for the required property, and returns its value if it exists.

If the property name is a reference to a list, this is considered as the property's path. Note that paths are easily expressed with square brackets, [and] (see example). Paths and their rules are described in section 5.3.6.

The get property is suitable for reading properties defined by set and const.

The set and get pair are coordinated in such a way, that the programmer can assign a list, a scalar or a hash to a property, and get the value later on in the easiest possible way. This is best explained in the example that follows, but the formal rules are hereby described for the sake of formality:

In scalar context: The first element in the list is returned. If a scalar was used to set

the property, this arrangement makes sure that `get` returns what `set` (or `const`) got. If the property wasn't defined, `undef` is returned.

In list context: A list is returned. If the property wasn't defined, it's an empty list.

Note that if the property contains a list, and `get` is evaluated in a scalar context, it does NOT return the number of elements, like some Perl programmers would expect.

Return value:

The value of the requested property.

Example:

```
use Perlilog;
init;
$object=root->new(name=>'theObject',
                  foo => 'bar');
print "My name is ".$object->get('name')."\n";
print "Foo is ".$object->get('foo')."\n";

$object->set('myscalar', 'scalarvalue');
$value = $object->get('myscalar');
print "My scalar is $value\n";
print "My scalar as a list: ".join(",", $object->get('myscalar'))."\n";

$object->set('mylist', 'listitem1', 'listitem2', 'listitem3');
@listvalue=$object->get('mylist');
print "My list is ".join(",", @listvalue)."\n";
print "My list (scalar context!): ".$object->get('mylist')."\n";

@thelist=(1, 2, 3);
$object->set('one_two_three', @thelist);
print "Let's count: ".join(",", $object->get('one_two_three'))."\n";

$object->set(['my', 'node'], "This is my node");
$object->set(['my', 'other'], "This is another node");
print "My node: ".$object->get(['my', 'node'])."\n";
print "My other: ".$object->get(['my', 'other'])."\n";
```

This script prints out the following:

```
My name is theObject
Foo is bar
My scalar is scalarvalue
My scalar as a list: scalarvalue
My list is listitem1,listitem2,listitem3
My list (scalar context!): listitem1
Let's count: 1,2,3
My node: This is my node
My other: This is another node
```

9.1.4 The method `const()`

Synopsis:
```
$obj->const($property, $scalar);
$obj->const($property, @list);
$obj->const(\@path, $scalar);
$obj->const(\@path, @list);
```

Syntax:

<u>object</u> -> `const(`<u>property</u>, <u>scalar value</u>`)`;

Description:
The `const` method is exactly like `set` (see section 9.1.2), only it sets the value of a property as a constant. If the property already exists, the new value value must be equal (stringwise, in the Perl `eq` sense) to the value that the property already has. The sense of equality may be changed from stringwise `eq` to any arbitrary sense by using the `seteq` method detailed in section 9.1.13.

When dealing with lists, equality means equality in the number of element in the list and that each element is stringwise equal (or as chosen with `seteq`).

Either way, the previous value, if assigned, must have been set by `const` (and not `set`).

If the above conditions are not met, a fatal error occurs.

Setting a constant value may trigger off a callback mechanism. See section 9.1.12.

For more information about the constant property, see section 5.3.4.

If the property name is a reference to a list, this is considered as the property's path. Note that paths are easily expressed with square brackets, `[` and `]` (see example). Paths and their rules are described in section 5.3.6.

Return value:
Not to be used.

Example:
```
use Perlilog;
init;
$object=root->new(name=>'theObject');

$object->const('myconstant', 'Stay Forever');
$value = $object->get('myconstant');
print "I say $value\n";

$object->const('myconstant', 'Stay Forever'); # This is OK
$object->set('myconstant', 'Stay Forever'); # This is an error
$object->const('myconstant', 'Change!'); # This is an error

$object->set('myscalar', 'I am not a constant');
$object->const('myscalar', 'I am not a constant'); # Error again!
```

This will result in

```
I say Stay Forever
```

and then an error will be reported, because of the attempt to use `set` on a constant value (it doesn't matter that the value would be the same).

The example shows other possible mistakes: Trying to change the value of `myconstant`, or using `const` on a property that is already assigned with `set`.

9.1.5   The method `globalobj()`

Synopsis:
```
$theglobal = $anyobject -> globalobj();
```

Syntax:
<u>object</u> -> globalobj();

Description:
The `globalobj` method returns a handle to the global object of the system. The return value is identical for any object that is an instantiation of a class derived from root, without having this method overrides. In simple words, it doesn't matter which object you run this method on, as long as it's supported.

The purpose of this method is to allow an easy access to the global object from within subroutines that define methods. From the main script, simply use the `globalobj()` routine.

See section 5.4.5 for details about the global object.

Return value:
A reference ("handle") to the global object

Example:
```
use Perlilog;
init;
$object=root->new(name=>'theObject');

$theglobal = $object->globalobj();
print "The object's name is ".$theglobal->get('name')."\n";

$shortcut = globalobj();
```

Note that `$shortcut` will have the same value as `$theglobal`. This shorter format is only possible because of `use Perlilog`, and hence it can't be used in class declarations.

9.1.6   The method `who()`

>   **Synopsis:**
>   ```
>   print "This is ".$object->who."\n";
>   ```
>
>   **Syntax:**
>   <u>object</u> -> `who()`;
>
>   **Description:**
>   The `who` method returns a short and concise identification of the object, so it is readily
>   recognized by humans. It is commonly used in error messages and alike.
>   Classes that are derived from `root` often override this method to give a better descrip-
>   tion. The object's name is always mentioned somehow by convention.
>   Note that by using `who`, we ask the object for some information, so we assume that
>   `$object` (in the synopsis) is a proper object reference. This is assumption should
>   be avoided, especially when handling error messages, due the unexpected nature of
>   errors. See `safewho` in section 9.1.7 for a solution.
>
>   **Return value:**
>   A short identifier of the object, helpful for humans.
>
>   **Example:**
>   ```
>   use Perlilog;
>   init;
>   $object = root->new(name=>'theObject');
>
>   print "This is ".$object->who."\n";
>   ```
>
>   Running this:
>
>   ```
>   This is object 'theObject'
>   ```

9.1.7   The method `safewho()`

>   **Synopsis:**
>   ```
>   print "This is ".$safeobject->safewho($object)."\n";
>   ```
>
>   **Syntax:**
>   <u>object</u> -> `who`(<u>object in question</u>);
>
>   **Description:**
>   `safewho` calls `who` on the object that is passed as an argument, after verifying (using
>   `isobject`) that the object reference is proper.
>   This is especially useful in code that define methods, since we know for sure that the

object's self reference is proper. We may thus call ourself with the `safewho` method, when attempting to identify another object.

Return value:
Same as `who` if the argument is a proper object. Otherwise, the string '(non-object item)' is returned.

Example:
```
use Perlilog;
init;
$object = root->new(name=>'theObject');
$safeobject = root->new(name=>'ImSafe');
$junk = "Hello";

print "This is ".$safeobject->safewho($object)."\n";
print "What is this??? ".$safeobject->safewho($junk)."\n";
```

Running this:

```
This is object 'theObject'
What is this??? (non-object item)
```

Note that trying `$junk->who` would cause a Perl error, that would be quite unhelpful.

9.1.8   The method `isobject()`

Synopsis:
```
if (isobject($obj)) { ...  }
```

Syntax:
<u>object</u> -> isobject(<u> scalar</u>);

Description:
`isobject` identifies if its scalar argument is a PERLILOG object. This method is useful before attempting to call an object's method with an arrow notation ("->"). The method's result does not depend on whose object it is a method of. Only scalar in the argument matters (unless the method has been overridden, which it shouldn't).
Note that if an object has been created outside the PERLILOG mechanism, `isobject` will return false even though it's OK to use the argument as an object.

Return value:
True (`1`) or undefined value (`undef`).

Example:
```
use Perlilog;
```

```
init;
$object = root->new(name=>'theObject');

print "Is the handle an object? ".globalobj()->isobject($object)."\n";
print "Is the name an object? ".globalobj()->isobject('theObject')."\n";
print "Can I check myself? ".$object->isobject($object)."\n";
```

Note that we use `globalobj()` just to get some object to call `isobject` on.
Running this we get:

```
Is the handle an object? 1
Is the name an object? 0
Can I check myself? 1
```

Also note that we got `0` when passing the object's name as an argument. The method
will always return `0` to any string it gets.

9.1.9 The method `objbyname()`

Synopsis:
> `$obj -> objbyname('theObject');`

Syntax:
> <u>object</u> -> objbyname(<u>name of object</u>);

Description:
> `objbyname()` returns the handle (reference) to the object whose name is given as an
> argument. If no such object exists, `undef` is returned.

Return value:
> A reference to an object or `undef`.

Example:
```
use Perlilog;
init;
$object = root->new(name=>'theObject');

$same = globalobj->objbyname('theObject');

print("The same object!\n")
  if ($object == $same);
```

Description:
> In this example, we create a new object, and puts its handle in `$object`. Then we get
> the same value by using `objbyname()` on the object's name, and put it in `$same` (which

is useless for practical reasons, since we've already have the reference of the objects handy). Note that we use the global object, like we would if we had no other reference to an object handy.

When running this script, the `print` is executed.

9.1.10   The method `objdump()`

Synopsis:
```
$obj -> objdump;
$obj -> objdump('theObject');
$obj -> objdump($objref);
```

Syntax:
<u>object</u> -> `objbyname()`;
<u>object</u> -> `objbyname(`<u>list of names or references of objects</u>`)`;

Description:
`objdump()` is intended for debugging. It prints out information (to the standard output) about either a specific object, or all the objects that are defined in the system.

If a list of objects is given (this includes one object) as argument, these objects' information is printed out, which includes the properties.

The objects may be identified by their name or reference interchangably.

If `objdump()` is called with no arguments, all objects in the system are showed in the order that they were created.

The output format is intended for human reading, and is thus presented in what seems to be an easy way to read, with less emphasis on formal rules.

Return value:
Not to be used.

Example:
```
use Perlilog;
init;
$object = root->new(name=>'theObject');

globalobj->objdump($object);     # Will show 'theObject'
globalobj->objdump('theObject'); # 'theObject' again
$object->objdump(globalobj);     # Will show global object!
globalobj->objdump();            # Will show them all
```

9.1.11 The method `suggestname()`

Synopsis:
```
$safename = $obj->suggestname($IWantThis);
$obj = AnyClass->new(name => $safename);
```

Syntax:
object -> `suggestname`(desired name)

Description:
`suggestname()` will check the given string against the names of already existing objects. If the name is OK for a new object, it will simply return the string. If a new object can't be named with the given string, it is altered lightly (see example). Either way, the returned string is a legal name for a new object, which will be close enough to the original.
Note, that `suggestname` may suggest the same name more than once, if it isn't used to create a new object – it checks uniqueness against existing objects, not its own suggestions.

Return value:
A string with a name for a new object.

Example:
```
use Perlilog;
init;
$global = globalobj();
$name1 = $global -> suggestname('theObject');
$object = root->new(name => $name1);
print "The first object was called $name1\n";

$name2 = $global -> suggestname('THEOBJECT');
$object = root->new(name => $name2);
print "The next object was called $name2\n";

$name3 = $global -> suggestname('theobject');
print "Now we were suggested the name $name3\n";
$name4 = $global -> suggestname('theobject');
print "We didn't use it, so we got $name4 again\n";

$name5 = $global -> suggestname('theobjects');
print "We added one 's', and got $name5 -- it's unique\n";
```

We run this:

```
The first object was called theObject
The next object was called THEOBJECT_1
```

```
Now we were suggested the name theobject_2
We didn't use it, so we got theobject_2 again
We added one 's', and got theobjects -- it's unique
```

9.1.12 The method `addmagic()`

Synopsis:
```
$object->addmagic($property, \&callback);
$object->addmagic($property, sub { ...  });
$object->addmagic(\@path, \&callback);
$object->addmagic(\@path, sub { ...  });
```

Syntax:

<u>object</u> -> addmagic(<u>property</u>, <u>subroutine reference</u>);

Description:

The `addmagic` method queues a callback subroutine, which will be called upon when the respective property is assigned a value by using `const`. If The property's value is already set, the subroutine will be called immediately.

If the property name is a reference to a list, this is considered as the property's path. Paths and their rules are described in section 5.3.6.

`addmagic` is loop-safe: The callback mechanism assures that infinite callback loops will not occur. This is done by removing each callback entry from the queue before performing the callback itself, so each callback entry is run at most once. In particular, the callback subroutine may include a call to the `const` of another object, which may trigger off another callback. In fact, this is the intended use of `addmagic` (see examples).

Using `addmagic` and callback subroutines requires an understanding of the scope under which the subroutines are run (that is, when variables are evaluated, and what variable space is applied). A lack of such understanding may lead to strange bugs. We shall address a few of the issues in the examples below.

Return value:

Not to be used.

Example:
```
use Perlilog;
init;
$object1 = root->new(name=>'First');
$object2 = root->new(name=>'Second');

$copy1to2 = sub {
  print "Callback to copy1to2\n";
  my $val = $object1 -> get('TheProperty');
  $object2 -> const('TheProperty', $val);
```

```
};

$copy2to1 = sub {
  print "Callback to copy2to1\n";
  my $val = $object2 -> get('TheProperty');
  $object1 -> const('TheProperty', $val);
};

$object1->addmagic('TheProperty', $copy1to2);
$object2->addmagic('TheProperty', $copy2to1);

$object1->const('TheProperty', 'TheValue');
print "The value is ".$object2->get('TheProperty')."\n";
```

And when running this, we get:

```
Callback to copy1to2
Callback to copy2to1
The value is TheValue
```

The heart of this example is that we ran `const` on $object1, but read the property of $object2. This demonstrates how one object can "learn" the value of another one as soon as it is set.

We see that by calling `const` on $object1 triggered off the callback to `copy1to2`. In `copy1to2` there's `const` is called on $object2, and thus `copy2to1` is triggered off. In `copy2to1` we call `const` on $object1 again, on a constant property that already has a value. This is OK, since we attempt to assign the same value that the property already has.

No more callbacks take place, since we've exhausted the queues. Note that each of the two objects watch the other. We could have ran `const` on $object2, and read it from $object1 as well. In fact, this callback setup assures that both object's properties will be equal, both by copying the value, and by not allowing unequal values to be set. In the example above, the subroutines could have been defined as `sub copy1to2 { ... }` (and not `$copy1to2 = sub { ... }`), achieving the same results for this specific example. (In this case `\&copy1to2` would be given to `addmagic`, rather than `$copy1to2`). Even so, it's still higly recommended to follow the example's subroutine definition format, in order to assure the correct scoping. This is especially important if the subroutines are defined as part of some method routine.

We now see another example, which demonstrates a variable scoping issue. Suppose that we want 10 objects, whose property X is equal on all:

```
use Perlilog;
init;
@l=();
for ($i=0; $i<10; $i++) {
  $l[$i]=root->new(name=>'MyName'.$i);
```

```
    my $j=$i;
    if ($l[$j+1]) {
      $l[$j]->addmagic('X',
                            sub { $l[$j+1]->const('X', $l[$j]->get('X') ); });
    }
    if ($l[$j-1]) {
      $l[$j]->addmagic('X',
                            sub { $l[$j-1]->const('X', $l[$j]->get('X') ); });
    }
}

print "Before, the value is '".$l[3]->get('X')."'\n";
$l[8]->const('X', 'what we want');
print "After, the value is '".$l[3]->get('X')."'\n";
```

Running this, we get:

```
Before, the value is ''
After, the value is 'what we want'
```

In this example, we pass an anonymous subroutine (sub {...}) to addmagic. It is important to note, that even though the loop index is $i, we create a copy of it, my $j=$i; and use it within the callback.

The reason for this, is that the variables in the subroutine are evaluated only upon execution, but the values are those that the variables have at the moment of execution. Thus, we couldn't use $i in the callback subroutine, because it would have the value 10 (the final value) for all subroutines. By making a "local copy" with my, within a Perl block, we get the right $j for each callback.

This example may be confusing, but it shown the importance of knowing the scoping issue well.

9.1.13 The method seteq()

Synopsis:
```
$object->seteq($property, \&compare);
$object->seteq($property, sub { ...  });
$object->seteq(\@path, \&compare);
$object->seteq(\@path, sub { ...  });
```

Syntax:
object -> seteq(property, subroutine reference);

Description:

`seteq` changes the meaning of equalilty for a certain property. This meaning is effective when `const` is used on a property that already has a value, which is when `const` verifies that the new value and the old one are "the same". The exact meaning of being "the same" is given by the argument to `seteq`, which is a reference to a comparing subroutine.

The comparing subroutine shall accept two arguments, and return `1` if the arguments are "the same" in the relevant sense, `0` otherwise.

`const` does not update the propery nor run callbacks when executed on a property that has a value, even if it is considered equal. If the property needs to be updated, the use of constant properties should be reconsidered.

By default, stringwise `eq` is used to compare the value given to `const`, as if `seteq($property, sub {return shift eq shift;})` had been run on every property.

If the property name is a reference to a list, this is considered as the property's path. Paths and their rules are described in section 5.3.6.

Return value:

      Not to be used.

Example:

```
use Perlilog;
init;
$object = root->new(name=>'theObject');

$object -> seteq('theProperty', \&ignorecase);

$object -> const('theProperty', 'THEVALUE');

#The next line would cause an error if it wasn't for seteq above
$object -> const('theProperty', 'thevalue');

print "The value is ".$object->get('theProperty')."\n";

sub ignorecase {
  my ($a, $b) = @_;
  return (lc($a) eq lc($b));
}
```

Running, this:

```
The value is THEVALUE
```

In this example, we make the comparison case-insensitive by applying the subroutine `ignorecase` on `seteq`. Unlike `addmagic` (see section 9.1.12), it's proper to use named subroutines (defined as `sub ignorecase { ... }`), since the result of the subroutine

depends only on its arguments, and hence scoping issues are irrelevant.
We can see that the value of the property did not change, but no error was reported.

9.1.14   The method `registerobject()`

Synopsis:
$object -> registerobject($phase);

Syntax:
<u>object</u> -> registerobject(<u>phase</u>);

Description:
`registerobject` registers the object in the PERLILOG environment. This registration assures that the object will be called upon during the execution phase (see section 6.5 for more details).

The phase argument can be one of `begin`, `end` or `noreg`. `begin` and `end` mean that the object should be called as early or late as possible, for each method. `noreg` means causes the method to return without any action.

Objects that are inherited from the `codegen` class (including `verilog`) are registered upon creation, so there is almost never any reason to use this method explicitly.

Return value:
Not to be used.

9.1.15   The method `pshift()`

Synopsis:
$scalar=$obj->pshift($property);
$scalar=$obj->pshift(\@path);

Syntax:
<u>object</u> -> pshift(<u>property</u>);

Description:
`pshift` treats the given property as a list, and performs a Perl `shift` operation on that list: It removes the first item of the list, and returns its value. If the property is undefined, or it's an empty list, `undef` is returned.

The property must not have been defined with `const`.

If the property name is a reference to a list, this is considered as the property's path. Paths and their rules are described in section 5.3.6.

Return value:
Same as Perl's `shift` on a list.

Example:
> See the example in section 9.1.18

9.1.16 The method `ppop()`

Synopsis:
> `$scalar=$obj->ppop($property);`
> `$scalar=$obj->ppop(\@path);`

Syntax:
> <u>object</u> -> `ppop(`<u>property</u>`);`

Description:
> `ppop` treats the given property as a list, and performs a Perl `pop` operation on that list:
> It removes the last item of the list, and returns its value. If the property is undefined,
> or it's an empty list, `undef` is returned.
> The property must not have been defined with `const`.
> If the property name is a reference to a list, this is considered as the property's path.
> Paths and their rules are described in section 5.3.6.

Return value:
> Same as Perl's `pop` on a list.

Example:
> See the example in section 9.1.18

9.1.17 The method `punshift()`

Synopsis:
> `$obj->punshift($property, $scalar);`
> `$obj->punshift($property, @list);`
> `$obj->punshift(\@path, $scalar);`
> `$obj->punshift(\@path, @list);`

Syntax:
> <u>object</u> -> `punshift(`<u>property</u>`, `<u>list items</u>`);`

Description:
> `punshift` treats the given property as a list, and performs a Perl `unshift` operation on
> that list: It adds the given items at the beginning of the list. If the property is undefined,
> it is created with `set`.
> The property must not have been defined with `const`.
> If the property name is a reference to a list, this is considered as the property's path.
> Paths and their rules are described in section 5.3.6.

Return value:
> Not to be used.

Example:
> See the example in section 9.1.18

9.1.18 The method `ppush()`

Synopsis:
```
$obj->ppush($property, $scalar);
$obj->ppush($property, @list);
$obj->ppush(\@path, $scalar);
$obj->ppush(\@path, @list);
```

Syntax:
> <u>object</u> -> `ppush`(<u>property</u>, <u>list items</u>);

Description:
> `ppush` treats the given property as a list, and performs a Perl `push` operation on that list: It adds the given items at the end of the list. If the property is undefined, it is created with `set`.
>
> The property must not have been defined with `const`.
>
> If the property name is a reference to a list, this is considered as the property's path. Paths and their rules are described in section 5.3.6.

Return value:
> Not to be used.

Example:
```
use Perlilog;
init;
$object = root->new(name=>'theObject');

$object -> set('myList', 5, 6, 7, 8);

print "This is five: ".$object -> pshift('myList')."\n";
print "This is eight: ".$object -> ppop('myList')."\n";

$object -> punshift('myList', 1, 2);
$object -> ppush('myList', 'Finito!');

print "My list is ".join(', ', $object -> get('myList'))."\n";
```
> Which prints when runned:
```
This is five: 5
```

```
This is eight: 8
My list is 1, 2, 6, 7, Finito!
```

9.1.19 The method `addchild()`

Synopsis:

```
$child -> setparent($parent);
```

Syntax:

child object `-> setparent ->` (parent object);

Description:

`setparent` registers the object passed as an argument as a parent of the object, on which the methon is executed on.

In effect this sets the `parent` property of the child to a reference to the parent object, and adds a reference to the child object to the list of children, which the `children` property consists of.

Return value:

Not to be used.

9.1.20 The method `prettyval()`

Synopsis:

```
print "I have ".$self->prettyval(@things)."\n";
```

Syntax:

object `-> prettyval(`list`)`;

Description:

`prettyval()` accepts items in a list, and returns a string with the items presented in a way that humans understand. This should be used in error messages and similar cases, when we want to present the value in a message.

`prettyval()` handles lists by printing a few of the first elements in parantheses. What appears to be non-numerical strings is enclosed with quote marks. Object references are translated into their `who()` representation, enclosed in curled brackets.

This method may be used in conjunction with `linebreak()` in order to handle long lines.

Return value:

A well-formatted string

Example:

See the example in section 9.1.21

9.1.21 The method `linebreak()`

Synopsis:
```
print $self->linebreak($A_long_string);
print $self->linebreak($A_long_string, ' ');
```

Syntax:
object -> linebreak(string[, indent string]);

Description:
The `linebreak()` method searches the string for lines that are over 80 characters in length (newline to newline), and attempts to cut them wisely by adding newlines.

If a second argument is given, that string will be put after each newline that is inserted. If the string is a few whitespaces, this will turn out to be the indentation of each line break that the method creates.

With "newline" we mean a Perl \n.

Return value:
A (hopefully) better formatted string

Example:
```
use Perlilog;
init;
$object = root->new(name=>'theObject');

@thelist = ('Foo', 5, $object, globalobj(), 'Bar', 'FooBar');

print globalobj->linebreak("I don't know what to do with ".
   globalobj->prettyval(@thelist)."\n");
```

This will print out:

```
I don't know what to do with ('Foo', 5, {object 'theObject'},
{The Global Object}, ...)
```

Note that the line break before the forth element is a result of the `linebreak()` method. We can also see that only the first four elements were actually displayed. The rest are chopped out.

9.1.22 The method `treestudy()`

Synopsis:
> $object -> treestudy();

Syntax:
>   object -> treestudy();

Description:
> `treestudy` performs a scan of all objects that belong to the same object tree as the the object, on which the method was called (not including port objects). Section 6.2.4 describes the object tree. The `treepath` property is updated for each of these objects (`treepath` is described in section 9.2.6).
> `treestudy` can thus be called from any object in the relevant tree.
> As a side effect, `treestudy` also verifies that the tree is well built: That there are no loops, and that objects point at each other as child-parent pairs properly. In addition, it is also verified that all objects in the tree can be resolved by their name by looking them up in the the `%Perlilog::objects` hash.
> A fatal error will be produced if any of these verifications fails, so the tree's integrity is assured immediately after a return from this method.

Return value:
> A reference to the tree's root object (the object that has no parent).

9.2 Properties

9.2.1 The property `name`

Description:
> The `name` property is a unique ASCII identifier of the object. It must be set when creating a new object.
> The first character of the property string must be an underscore or a letter (upper case or lower case). The rest of the string must match `\w*` (as a Perl regular expression). `MyName`, `Hello_9`, and `_underscored` are legal names. `2good`, `-myname`, `%Name` and `/slash` are illegal.
> The `name` property must be unique in a case-insenstive sense.
> See 9.1.1 and 9.1.11 for more details.

9.2.2 The property `parent`

Description:
> `parent` is a reference to the parent object. The property is of constant type, and is hence a scalar. For the same reason, an object can have only one parent, and it can't

be changed once set.

Example:
```
use Perlilog;
init;
$object = root->new(name=>'BigPapa');
$son = root->new(name=>'Boy',
                    parent=>$object);

print "The parent is ".$son->safewho($son->get('parent'))."\n";
```

Which prints out;

```
The parent is object 'BigPapa'
```

Note that we use `safewho`, because an object may in general not have a parent, so `get('parent')` could return `undef`.

9.2.3   The property `children`

Description:
This property is a list of the object's children. The property changes as new children are registered.

Example:
```
use Perlilog;
init;
$object = root->new(name=>'BigPapa');
$son = root->new(name=>'Boy',
                    parent=>$object);
$daughter = root->new(name=>'Girl',
                        parent=>$object);

foreach ($object->get('children')) {
  print "I'm the parent of ".$_->who."\n";
}
```

Which prints out:

```
I'm the parent of object 'Boy'
I'm the parent of object 'Girl'
```

9.2.4   The property `nick`

Description:
The `nick` property must be initialized when creating a new object with `new()`, if this

call occurs during transient mode (in other words, as part of an `attempt()` method of an interface class). It is otherwise irrelevant.

The property will be used a suggestion when determining the name of the object in case it becomes sustained. An explanation of this is given in section 7.10.2.

An example of this can be seen in section 7.3.2.

9.2.5   The property `perlilog-transient`

Description:

This property is relevant to object that were created during transient mode. Thus, it should not be defined on any object except those which were part of an attempt to interface between some ports.

It set by the system to one of two values, if at all: `transient` or `sustained`. If the value is `sustained`, it means that the object was created as a transient object, but was sustained, and is therefore a normal object now. The value `transient` may be encountered by methods of interface classes, and means that the object is currently transient, and may vanish very soon.

More about transient objects in section 7.10.

9.2.6   The property `treepath`

Description:

In essence, `treepath` is a property which helps climbing along the object tree in order to reach a certain object. Each object's `treepath` tells where to go on on the one next step. Note that the destination object is requested by its name, and not by its reference.

The `treepath` property is set as a result of a call to the `treestudy` method (see section 9.1.22). It is a reference to a hash, whose keys are the names of the objects, that can be reached from the object, by climbing on the object tree. The value of each entry is a reference to another object, which is the next object to climb to, if we want to reach the object whose name is the key. The value is always a reference to the object's parent or one of its children.

By convention, the path from one object to itself is one step to the parent, and one step back. If the object has no parent (root object), the hash will not be set for the object itself (no path).

Example:

```
use Perlilog;
init;
$object = root->new(name=>'BigPapa');
$son = root->new(name=>'Boy',
                 parent=>$object);
```

```
$daughter = root->new(name=>'Girl',
                        parent=>$object);

$object->treestudy();
%path = %{$son->get('treepath')};

foreach $key (sort keys %path) {
  print $key."->".$path{$key}->who."\n";
}
```

Which prints out:

```
BigPapa->object 'BigPapa'
Boy->object 'BigPapa'
Girl->object 'BigPapa'
```

In this simple case, all entries in the hash points to `BigPapa`, including the path to the object itself. A more complex tree would give more interesting results.

10   The code generating class API

10.1   Methods

10.1.1   The method `varwho()`

Synopsis:
> `$object->varwho($varID);`

Syntax:
> object -> `varwho(`variable ID`)`;

Description:
> `varwho` returns human-helpful information on a variable, in a similar way to the way `safewho` works on objects (see section 9.1.7 about `safewho`).
> The issue of variables in PERLILOG is documented in section 6.7.

Return value:
> A string consisting of a human-understandable identification of the variable.

Example:
> See the example in section 11.1.2

10.1.2   The method `attach()`

Synopsis:
> `$object->attach($varID1, $varID2, $comment);`

Syntax:
> object -> `attach(`variable ID 1, variable ID 2, comment `)`;

Description:
> The `attach` method makes the two variables "electrically connected", or connected as in Verilog's `assign`. The variables are identified by their ID's.
> Both variables must have the same value in their `dim` property, or, alternatively, the property may not be defined. `addmagic` is used within the method routine to make each of the variables take after the other's `dim` property, even if it is assigned in the future. (see section 9.1.12).

The third argument, the comment, should be a short note for human reading, which makes clear what caused the call to `attach`. This information should be useful in tracing mistakes which cause errors that lead the wrong wires to be connected.

The behavior of `attach` does not depend on the object that it's called on, unless the method has been overridden. It is OK to `attach` a variable to itself, but this makes `attach` to return doing nothing.

The issue of variables in PERLILOG is documented in section 6.7.

Example:
See the example in section 11.1.2

10.1.3 The method `IDvar()`

Synopsis:
```
($obj, $var) = $self->IDvar($ID);
$var = $self->IDvar($ID);
```

Syntax:
list or scalar lvalue = object -> IDvar(variable ID);

Description:
`IDvar()` is given a variable's ID, and returns the varible's name in Verilog, and possibly a reference to the object it belongs to (depending on if it was called in scalar or list context).

The behaviour of `IDvar()` does not depend on which object it was executed on (unless the method has been overridden).

For more, see section 6.7.2.

Return value:
When called in list context, it returns a list of two items. The first item is a reference to the object, to which the variable belong. The second item is the variable's name.

When called in scalar context, only the variable's name is returned.

10.1.4 The method `getID()`

Synopsis:
```
$ID = $obj->getID('myVar');
```

Syntax:
list = object -> getID(list of variable names);
scalar = object -> getID(variable name);

Description:

> `getID` looks up a each variable in the list according to its name, and returns the respective list of IDs. The variables must belong to the object, on which the method is called.
>
> A fatal error is issued if some variable does not exist.

Example:

> See the example in section 11.1.3

10.1.5 The method `samedim()`

Synopsis:

> `$obj -> samedim('data1', 'data2');`

Syntax:

> object -> `samedim`(<u>list of variable names</u>);

Description:

> `samedim()` forces all the variables, whose names are given in the list, to have the same `dim` property. This is done with magic callbacks, so the variables will affect each other whenever they are assigned values.
>
> Unlike `attach()`, this method accepts any number of variables, and the variables are given by their names, and not by their IDs. All variables must belong to the same object, which is the object on which the method is called.
>
> Using this method is highly recommended when making procedural or continous assignments in Verilog between two variables, wherever it makes sense to force the two variables to be of the same dimension. See section 7.7.9.

Return value:

> Not to be used

10.1.6 The method `getport()`

Synopsis:

> `interface($one->getport('myport'), $two->getport('hisport'));`

Syntax:

> object -> `getport`(<u>definition name of port</u>);

Description:

> `getport()` should be used in conjunction with template objects or with other objects which set up the object's port list in a similar manner.

This method looks up the port by the name that was used in its declaration, in the template file. Note that this name may not be the same as the port object's `name` property.

If there is no such port, a fatal error is issued.

Formally speaking, this method returns the value of a self-call to `get(['user_port_names', $name])`, but `getport()` is much more elegant in main scripts, and may be extended in future versions, so it recommended to stick to it.

There are plently of examples of using this method in section 2.3.2 and on, and some background can be found in 3.6.

Return value:
A reference to the requested port.

10.1.7 The method `labelID()`

Synopsis:
`%hash = $self->labelID($port);`

Syntax:
<u>hash</u> = <u>object</u> -> `labelID`(<u>port object</u>);

Description:
`labelID()` is tailored for use in interface classes. Its purpose is to supply the relevant variable IDs with as little fuss as possible.

The method accepts a reference to some port, on which it looks up the `labels` property. It returns a hash (actually, a list that becomes a hash easily) whose keys are the labels in the `labels` property, and the values are the relevant variable IDs.

The IDs are resolved as follows: If the value of some entry in the `labels` property is a decimal number (matches /^\d+$/), it is considered as the ID itself. Otherwise, the ID looked up among the variables of the port's owner (its parent): The value's string should be equal to some variable's name.

It doesn't matter on which object the method is called, unless some very basic methods have been overridden.

See section 7.3.5 for an example of using this method.

Return value:
A hash, whose keys are labels, and values are variable IDs.

10.1.8   The method `whereto()`

Synopsis:
```
$obj = $self->whereto;
```

Syntax:
<u>target object</u> = <u>object</u> `-> whereto();`

Description:
`whereto()` looks for an object which is the best one to receive the called object's Verilog code. It returns a reference to that object.

The chosen object is verified to be adequate, in the sense that it's not static. The lookup may thus fail, in which case an `undef` is returned instead of an object.

The return value of `whereto()` is cached, so within a single Perl run, it will always answer with the same reference. For the same reason, the return value may depend on what may seem as irrelevant factors.

`whereto()` is intended for use by interface objects, for which it's desired not to generate a Verilog file of their own.

See sections 7.5.4 and 7.5.10 for more about this method.

Return value:
A reference to an object, to which Verilog code should be appended, or `undef` if no relevant object could be found.

11   The Verilog class API

11.1   Methods

11.1.1   The method suggestvar()

Synopsis:
$safename = $obj->suggestvar($IWantThis);

Syntax:
<u>object</u> -> suggestvar(<u>desired name</u>)

Description:
suggestvar will check the given string against the names of already existing variables on the certain object. If the name is OK for a new variable, it will simply return the string. If a new variable can't be named with the given string, it is altered lightly (suggestvar creates new names like suggestname). Either way, the returned string is a legal name for a new variable, which will be close enough to the original.

Note, that suggestvar may suggest the same name more than once, if it isn't used to create a new variable – it checks uniqueness against existing variables, not its own suggestions.

Return value:
A string with a name for a new variable.

Example:
See the example in section 11.1.2

11.1.2   The method addvar()

Synopsis:
$object->addvar($name, 'wire', 'in', '[7:0]');

Synopsis:
$object->addvar($name);

Syntax:
<u>object</u> -> addvar(<u>variable name</u>, <u>type</u> [,<u>drive</u> [,<u>dimension</u>]]);

Description:

The `addvar` adds a variable to the object, and registers the attributes given. The variable's name is compared in a case-insensitive manner against the other variables of the object, and an error occurs if the name is not unique. (In order to avoid errors, `suggestvar` may be used).

The name of the variable is the same that will appear in the Verilog file.

The other attributes are not checked up by `addvar`, but improper setting will generate errors at later stages of execution.

The drive and dimension attributes are optional, and may be omitted, or equivalently, given as `undef`.

The variable type should be one of `wire`, `reg`, `input`, `output`, `inout`, or `outreg`, but this is not verified by `addvar`. The variable type may be changed by the system as a result of attachment with other variables.

The dimension attribute is the signal vector size (such as `[7:0]`), in Verilog format. For single-bit variables, it should be set to `''` (empty string, not `undef`).

If `addvar` is called with only the `$name` argument, it's a name reservation only.

Variables are described in section 6.7.

Return value:

Not to be used.

Example:

```
use Perlilog;
init;
$object = verilog->new(name => 'TheObject');

$name1 = $object->suggestvar('First');
$var1 = $object->addvar($name1, 'wire', 'in', '[7:0]');

$name2 = $object->suggestvar('Second');
$var2 = $object->addvar($name2, 'reg', 'out');

$object->attach($var1, $var2);

print "Variable #2 is ".$object->varwho($var2)."\n";
print "Variable #2's dim is ".$object->get(['vars', $name2, 'dim'])."\n";
```

Which will print out:

```
Variable #2 is 'Second' in module 'TheObject'
Variable #2's dim is [7:0]
```

There are a few things to note about this example: First, `suggestvar` is used here merely as a template: As the script is written, `suggestvar` will never offer another name.

We also note that the variable `Second` is created without the `dim` property set, but it is copied from the variable `First` due to the `attachment` between them.

It's also worth to mention, that in this example, we attached two variables within the same object. While this is perfectly legal, and usually yields a Verilog `assign` inside the resulting Verilog module, we usually attach different objects to achieve a connection.

11.1.3 The method `namevar()`

Synopsis:
```
$name = $obj->namevar('myVar', 'wire', 'in');
($name, $ID) = $obj->namevar('myVar', 'wire', 'in');
```

Syntax:
scalar = object -> `namevar`(variable name, type [,drive [,dimension]]);
list = object -> `namevar`(variable name, type [,drive [,dimension]]);

Description:
`namevar()` is a combination between `suggestvar()` and `addvar()`. It works like `addvar()`, except that the variable name that is given to `namevar()` is treated as a suggestion, and not as the final variable name.

When called in scalar context, `namevar()` returns the name that the variable actually got. In list context, both the name and the ID are returned.

Some background about this method is given in section 6.7.7, and a proper example can be found in section 7.5.5.

Return value:
In scalar context: The name of the variable created. In list context: A list, whose first item is the name of the variable actually created, and the second item is the ID of the variable.

Example:
```
use Perlilog;
init;
$obj = verilog->new(name=>'theObject');

($name1, $ID1) = $obj->namevar('myVar', 'wire', 'in');
($name2, $ID2) = $obj->namevar('myVar', 'reg', 'out');

$IDagain = $obj->getID('myVar');

print "name1=$name1, name2=$name2\n";
print "ID1=$ID1, ID2=$ID2, IDagain=$IDagain\n";
```

Note that two variables were generated with `myVar` as a suggestion. This script will

print out:

```
name1=myVar, name2=myVar_1
ID1=2, ID2=3, IDagain=2
```

We can see that the second variable got the name `myVar_1`, and that `IDagain` got the ID of the first variable, 2.

11.1.4 The method `copyvar()`

Synopsis:
```
$localname = $obj->copyvar($ID);
($localname, $theID) = $obj->copyvar($ID);
```

Syntax:
<u>scalar</u> = <u>object</u> -> `copyvar`(<u>variable ID</u>);
<u>list</u> = <u>object</u> -> `copyvar`(<u>variable ID</u>);

Description:
This method makes sure that some variable's value is accessible in the object it was called on. In practice this means returning some variable name in the specific object, which can be used in the Verilog.

If the requested variable happens to be in the relevant object, or there's already another variable in the object, which is attached to it, `copyvar()` will point out the variable which already exists. Otherwise, a new variable will be created `attach`'ed to the requested one.

Either way, only the Verilog code that the system itself generated will and is allowed to assign a value (in a Verilog statement) to the variable.

This method is explained futher in section 7.5.9.

Return value:
If called in a scalar context: The name of the variable, as it appears in the called object. In list context: A list of two items, first of which is the name of the variable. The second item is the ID of the variable that is known to be in the called object (it may be the same as the one given to the method).

11.1.5 The method `suggestins()`

Synopsis:
```
$safename = $obj->suggestins($IWantThis);
$safename = $obj->suggestins;
```

Syntax:
>object -> suggestins(<u>desired name</u>);
>object -> suggestins();

Description:
>When given a string as an argument, `suggestins()` will check the it against the names
>of already existing variables and instances (they share the same name space in Ver-
>ilog). If the name is OK for a new instance, it will simply return the string. If a new
>instance can't be named with the given string, it is altered lightly, as in `addvar()`.
>Note, that `suggestins` may suggest the same name more than once, if it isn't used to
>create a new instance – it checks uniqueness against existing instances, not its own
>suggestions.
>With no arguments, `suggestins()` will make up name according to the object's `name`
>property. Note that this will create a counterintuitive name, because the instance
>name will be based upon the name of the object containing the instantiation, and not
>the name of the object instantiated, as one would expect.

Return value:
>A string with a name for a new instance.

11.1.6 The method `addins()`

Synopsis:
>$object->addins($name);
>$object->addins($name, 'detached');

Syntax:
>object -> addins(<u>instance name</u> [,<u>detach</u>]);

Description:
>The `addins` method registers a new instance name to the system. When an instance
>of a real module is desired, the one-argument format should be used.
>When instantiating external units, such as Verilog `tran`, `tranif1` or `tranif0`, the sec-
>ond argument should be anything evaluated TRUE (the string `'detached'` is a good
>choice).
>The instance's name is compared in a case-insensitive manner against the other
>names in the object's common namespace of variables and instances. If that name is
>already taken, an error is issued.
>In fact, `addins()` is currently implemented as a call to `addvar()`, but it may make use
>of the additional second argument in the future. For this reason `addvar()` should not
>be used instead.

Return value:
>Not to be used.

11.1.7 The method `equivalent()`

Synopsis:
> `$object->equivalent($twin);`

Syntax:
> object -> addins(equivalent object);

Description:
> `equivalent()` should be used to avoid duplicate Verilog files with exactly the same content. This is commonly the case when identical objects are generated to fulfill the very same functionality and connectivity.
> If several objects are mutually equivalent, one should be used as the `$twin` argument in the synopsis above, all other objects taking the function of `$object`. Chained equivalence declarations will cause a fatal error.
> See section 2.4.4 for an example of using this method.

Return value:
> Always returns `1`.

11.1.8 The method `ontop()`

Synopsis:
> `$success = $object->ontop($verilog_code);`
> `$success = $object->ontop(@verilog_code_segments);`

Syntax:
> object -> ontop(list or scalar);

Description:
> `ontop` is used to add Verilog code to the `verilog` property by putting it before existing code. In general, this method accepts a list of segments of Verilog code. Each segment is appened with a newline (\n) if doesn't end with one. All segments are stringwise concatenated, and put at the beginning of the `verilog` property (the first item in the list will appear first in the code).
> `ontop` should be used only to put variable declarations in the `verilog` property, and not logic-generating Verilog.
> The simplest way to use this method, is to put the desired Verilog code in a single string, and call the method on it.
> A call to `ontop` will fail if the object is "static": Some objects don't allow the Verilog code to change (see section 6.9).

Return value:
> `1` on success, `0` on failure.

11.1.9 The method `append()`

Synopsis:
```
$success = $object->append($verilog_code);
$success = $object->append(@verilog_code_segments);
```

Syntax:
object -> append(<u>list or scalar</u>);

Description:
`append` is used to add Verilog code to the `verilog` property by putting it after exist-
ing code. In general, this method accepts a list of segments of Verilog code. Each
segment is appened with a newline (\n) if doesn't end with one. All segments are
stringwise concatenated, and put at the end of the `verilog` property (the last item in
the list will appear last in the code).

`append` should be used to add Verilog that generates logic, and not variable declara-
tions.

The simplest way to use this method, is to put the desired Verilog code in a single
string, and call the method on it.

A call to `append` will fail if the object is "static": Some objects don't allow the Verilog
code to change (see section 6.9).

Return value:
`1` on success, `0` on failure.

11.2 Properties

11.2.1 The property `vars`

Synopsis:
```
$object->set(['vars', $var, 'type'],$type);
$object->const(['vars', $var, 'dim'],$dim);
$object->const(['vars', $var, 'drive'],$drive);
$object->const(['vars', $var, 'ID'], $ID);
$type = $object->get(['vars', $var, 'type']);
$dim = $object->get(['vars', $var, 'dim']);
$drive = $object->get(['vars', $var, 'drive']);
$ID = $object->get(['vars', $var, 'ID']);
```

Description:
`vars` is not a property by itself, but rather a branch in the property tree.
`$obj->get('vars')` will generate a fatal error. The correct format to access the actual
properties is given in the synopsis.

A new variable must be created with `addvar`. Attempting to register a new variable

by setting the properties above in any other way will generate an inconsistency in the PERLILOG system. In particular, the subproperty `ID` must be set only by calling the `addvar` method.

Note that in the synopsis, `$var` is the variable name string, as it appears in the Verilog file (and not some referece format).

Example:

See the example in section 11.1.2

11.2.2 The property `varslist`

Description:

`varslist` is a list of variable and instance names of the specific object. This property should be altered only by the system.

11.2.3 The property `verilog`

Synopsis:

```
$code = $obj->get('verilog');
wrong(...)  if (defined $obj->get('static'));
$obj->set('verilog', $code);
```

Description:

The `verilog` property is a scalar strings, consisting of the Verilog code that is produced by the object. This property's value is written into the file given in `vfile` in the system `files` stage.

The manipulation of this property should be done with `ontop` and `append` whenever possible. When more sophisticated manipulations are needed, using `set` to change the property's value is fine. In this case, the `static` property must be verified to be undefined to assure that altering this property is indeed allowed. A fatal error may occur otherwise.

Description:

The `static` property signals that the `verilog` property is set by `const` and must not be changed. Any defined value of this property means that `verilog` should not be `set`.

This is useful when importing Verilog files that are already complete, and tampering with them is not allowed.

11.2.4 The property `vfile`

Description:

The `vfile` property is the name of the Verilog file which will be created out of the specific object. `vfile` is given as a complete directory path, either from the current

working directory, or from root.

If the property is not set when reaching the system `complete` stage, it will be `const`ed to the name of the object, followed by a `.v` suffix. The path will be taken as the `filesdir` property of the global object.

11.2.5   The property `static`

Description:

When this property is true (has a value that evaluates true in Perl), the object is static. See section 6.9 about static objects.

11.2.6   The property `viasource`

Description:

This property contains a list of objects, from which variable's names should be copied, whenever a `via`-named variable would be generated.

By default, whenever a wire is created by the system in order to let a signal go via the Verilog module, the variable's name is chosen by the system, and it is marked as a via wire with the `via` suffix.

If the `viasource` property contains a list of objects, these objects are searched for a variable that is "electrically equivalent" to the to-be via wire. If such variable is found, the via variable will have the same name as the the variable found. The objects are scanned in their order of appearance in the property.

If none of the objects contains a suitable candidate to copy its name from, the default mechanism is used.

This is useful to assure that top-level modules have variables with meaningful names. See section 2.3.5 for more about this.

Example:

```
$tb = template->new(name => 'tb',
                    tfile => 'tb.pt');
$top = template->new(name => 'top',
                     tfile => 'top.pt',
                     parent => $tb);
$top->set('viasource', $tb); # A single object in this case
```

11.2.7   The property `perlilog-no-file`

Description:

This property will be assigned the value `1` (with `const()`) by the system during the `instantiate()` execution stage if it turns out that the current object should not generate a file. It's highly discouraged to write to this property otherwise.

See section 7.11.3 for more about objects that don't generate files.

---

12 The global object's API

12.1 Methods

12.1.1 The method `execute()`

Synopsis:
        globalobj->execute;

Syntax:
        <u>The global object</u> `-> execute;`

Description:
        The `execute` method kicks off the execution stage of the PERLILOG environment. A
        call to this method is usually performed at the end of the main script.
        See section 6.5 for a detailed description of this topic.

12.2 Properties

12.2.1 The property `filesdir`

Description:
        `filesdir` is a scalar property whose value is the default path for the Verilog files. It is
        used during the `complete` stage while setting the complete file names for Verilog files
        whose `vfile` property has not been set until that stage.
        This property is set (and may be altered later) to `'./PLverilog'` at PERLILOG initial-
        ization.

12.2.2 The property `vfiles`

Description:
        `vfiles` is a list of the Verilog file names registered, or in other words, a list of the
        values of `vfile` of Verilog objects and its derivatives.
        `vfiles` is created during the `complete` execution stage during the calls to the different
        objects.

12.2.3   The property `system`

Description:
The `system` is a reference to the system object, which is a second global object, whose main purpose is to hold some system-sensitive properties.

12.2.4   The property `MAX_INTERFACE_REC`

Description:
The maximal recursion depth of `intobject()` calls. See section 7.9.4

13 Port and interface object's properties

13.1 Port object properties

13.1.1 The property `labels`

Description:

The `labels` property is a recommended way of pointing out which variables play roles in some certain port. It is explained in section 7.1.3.

13.1.2 The property `mates`

Description:

This property is a list of all ports, that the current port was mated with during a successful interfacing. The list does not include the port that holds the property.

If the same port was used several times in a recursive interfacing attempt chain, which was successful, `mates` will contain the ports that were mated during the innermost call to `intobjects()`. In other words, the ports that will appear in the list will be those that were involved in the most direct port mating, and will reflect the most trivial connection.

See section 7.7.7.

13.2 Interface object properties

13.2.1 The property `perlilog-ports-to-connect`

Description:

When an interface object is sustained, the list of ports that its `attempt()` method was called with, is stored by the system in `perlilog-ports-to-connect`. This makes it easy to access this list of ports in successive routines.

But more important, it frees the `attempt()` method from storing the list of ports it was called with (which it isn't allowed to do, anyhow).

For more about this, see section 7.5.3.

A   GNU FDL license

GNU Free Documentation License
Version 1.2, November 2002

Copyright (C) 2000,2001,2002 Free Software Foundation, Inc. 51 Franklin St, Fifth Floor, Boston, MA 02110-1301 USA

Everyone is permitted to copy and distribute verbatim copies of this license document, but changing it is not allowed.

A.0   PREAMBLE

The purpose of this License is to make a manual, textbook, or other functional and useful document "free" in the sense of freedom: to assure everyone the effective freedom to copy and redistribute it, with or without modifying it, either commercially or noncommercially. Secondarily, this License preserves for the author and publisher a way to get credit for their work, while not being considered responsible for modifications made by others.

This License is a kind of "copyleft", which means that derivative works of the document must themselves be free in the same sense. It complements the GNU General Public License, which is a copyleft license designed for free software.

We have designed this License in order to use it for manuals for free software, because free software needs free documentation: a free program should come with manuals providing the same freedoms that the software does. But this License is not limited to software manuals; it can be used for any textual work, regardless of subject matter or whether it is published as a printed book. We recommend this License principally for works whose purpose is instruction or reference.

A.1   APPLICABILITY AND DEFINITIONS

This License applies to any manual or other work, in any medium, that contains a notice placed by the copyright holder saying it can be distributed under the terms of this License. Such a notice grants a world-wide, royalty-free license, unlimited in duration, to use that work under the conditions stated herein. The "Document", below, refers to any such manual or work. Any member of the public is a licensee, and is

addressed as "you". You accept the license if you copy, modify or distribute the work in a way requiring permission under copyright law.

A "Modified Version" of the Document means any work containing the Document or a portion of it, either copied verbatim, or with modifications and/or translated into another language.

A "Secondary Section" is a named appendix or a front-matter section of the Document that deals exclusively with the relationship of the publishers or authors of the Document to the Document's overall subject (or to related matters) and contains nothing that could fall directly within that overall subject. (Thus, if the Document is in part a textbook of mathematics, a Secondary Section may not explain any mathematics.) The relationship could be a matter of historical connection with the subject or with related matters, or of legal, commercial, philosophical, ethical or political position regarding them.

The "Invariant Sections" are certain Secondary Sections whose titles are designated, as being those of Invariant Sections, in the notice that says that the Document is released under this License. If a section does not fit the above definition of Secondary then it is not allowed to be designated as Invariant. The Document may contain zero Invariant Sections. If the Document does not identify any Invariant Sections then there are none.

The "Cover Texts" are certain short passages of text that are listed, as Front-Cover Texts or Back-Cover Texts, in the notice that says that the Document is released under this License. A Front-Cover Text may be at most 5 words, and a Back-Cover Text may be at most 25 words.

A "Transparent" copy of the Document means a machine-readable copy, represented in a format whose specification is available to the general public, that is suitable for revising the document straightforwardly with generic text editors or (for images composed of pixels) generic paint programs or (for drawings) some widely available drawing editor, and that is suitable for input to text formatters or for automatic translation to a variety of formats suitable for input to text formatters. A copy made in an otherwise Transparent file format whose markup, or absence of markup, has been arranged to thwart or discourage subsequent modification by readers is not Transparent. An image format is not Transparent if used for any substantial amount of text. A copy that is not "Transparent" is called "Opaque".

Examples of suitable formats for Transparent copies include plain ASCII without markup, Texinfo input format, LaTeX input format, SGML or XML using a publicly available DTD, and standard-conforming simple HTML, PostScript or PDF designed for human modification. Examples of transparent image formats include PNG, XCF and JPG. Opaque formats include proprietary formats that can be read and edited only by proprietary word processors, SGML or XML for which the DTD and/or processing tools are not generally available, and the machine-generated HTML, PostScript or PDF produced by some word processors for output purposes only.

The "Title Page" means, for a printed book, the title page itself, plus such following

pages as are needed to hold, legibly, the material this License requires to appear in the title page. For works in formats which do not have any title page as such, "Title Page" means the text near the most prominent appearance of the work's title, preceding the beginning of the body of the text.

A section "Entitled XYZ" means a named subunit of the Document whose title either is precisely XYZ or contains XYZ in parentheses following text that translates XYZ in another language. (Here XYZ stands for a specific section name mentioned below, such as "Acknowledgements", "Dedications", "Endorsements", or "History".) To "Preserve the Title" of such a section when you modify the Document means that it remains a section "Entitled XYZ" according to this definition.

The Document may include Warranty Disclaimers next to the notice which states that this License applies to the Document. These Warranty Disclaimers are considered to be included by reference in this License, but only as regards disclaiming warranties: any other implication that these Warranty Disclaimers may have is void and has no effect on the meaning of this License.

A.2   VERBATIM COPYING

You may copy and distribute the Document in any medium, either commercially or noncommercially, provided that this License, the copyright notices, and the license notice saying this License applies to the Document are reproduced in all copies, and that you add no other conditions whatsoever to those of this License. You may not use technical measures to obstruct or control the reading or further copying of the copies you make or distribute. However, you may accept compensation in exchange for copies. If you distribute a large enough number of copies you must also follow the conditions in section A.3.

You may also lend copies, under the same conditions stated above, and you may publicly display copies.

A.3   COPYING IN QUANTITY

If you publish printed copies (or copies in media that commonly have printed covers) of the Document, numbering more than 100, and the Document's license notice requires Cover Texts, you must enclose the copies in covers that carry, clearly and legibly, all these Cover Texts: Front-Cover Texts on the front cover, and Back-Cover Texts on the back cover. Both covers must also clearly and legibly identify you as the publisher of these copies. The front cover must present the full title with all words of the title equally prominent and visible. You may add other material on the covers in addition. Copying with changes limited to the covers, as long as they preserve the title of the Document and satisfy these conditions, can be treated as verbatim copying in other respects.

If the required texts for either cover are too voluminous to fit legibly, you should put the

first ones listed (as many as fit reasonably) on the actual cover, and continue the rest onto adjacent pages.

If you publish or distribute Opaque copies of the Document numbering more than 100, you must either include a machine-readable Transparent copy along with each Opaque copy, or state in or with each Opaque copy a computer-network location from which the general network-using public has access to download using public-standard network protocols a complete Transparent copy of the Document, free of added material. If you use the latter option, you must take reasonably prudent steps, when you begin distribution of Opaque copies in quantity, to ensure that this Transparent copy will remain thus accessible at the stated location until at least one year after the last time you distribute an Opaque copy (directly or through your agents or retailers) of that edition to the public.

It is requested, but not required, that you contact the authors of the Document well before redistributing any large number of copies, to give them a chance to provide you with an updated version of the Document.

A.4 MODIFICATIONS

You may copy and distribute a Modified Version of the Document under the conditions of sections A.2 and A.3 above, provided that you release the Modified Version under precisely this License, with the Modified Version filling the role of the Document, thus licensing distribution and modification of the Modified Version to whoever possesses a copy of it. In addition, you must do these things in the Modified Version:

A. Use in the Title Page (and on the covers, if any) a title distinct from that of the Document, and from those of previous versions (which should, if there were any, be listed in the History section of the Document). You may use the same title as a previous version if the original publisher of that version gives permission.

B. List on the Title Page, as authors, one or more persons or entities responsible for authorship of the modifications in the Modified Version, together with at least five of the principal authors of the Document (all of its principal authors, if it has fewer than five), unless they release you from this requirement.

C. State on the Title page the name of the publisher of the Modified Version, as the publisher.

D. Preserve all the copyright notices of the Document.

E. Add an appropriate copyright notice for your modifications adjacent to the other copyright notices.

F. Include, immediately after the copyright notices, a license notice giving the public permission to use the Modified Version under the terms of this License, in the form shown in the Addendum below.

G. Preserve in that license notice the full lists of Invariant Sections and required Cover Texts given in the Document's license notice.

H. Include an unaltered copy of this License.

I. Preserve the section Entitled "History", Preserve its Title, and add to it an item stating at least the title, year, new authors, and publisher of the Modified Version as given on the Title Page. If there is no section Entitled "History" in the Document, create one stating the title, year, authors, and publisher of the Document as given on its Title Page, then add an item describing the Modified Version as stated in the previous sentence.

J. Preserve the network location, if any, given in the Document for public access to a Transparent copy of the Document, and likewise the network locations given in the Document for previous versions it was based on. These may be placed in the "History" section. You may omit a network location for a work that was published at least four years before the Document itself, or if the original publisher of the version it refers to gives permission.

K. For any section Entitled "Acknowledgements" or "Dedications", Preserve the Title of the section, and preserve in the section all the substance and tone of each of the contributor acknowledgements and/or dedications given therein.

L. Preserve all the Invariant Sections of the Document, unaltered in their text and in their titles. Section numbers or the equivalent are not considered part of the section titles.

M. Delete any section Entitled "Endorsements". Such a section may not be included in the Modified Version.

N. Do not retitle any existing section to be Entitled "Endorsements" or to conflict in title with any Invariant Section.

O. Preserve any Warranty Disclaimers.

If the Modified Version includes new front-matter sections or appendices that qualify as Secondary Sections and contain no material copied from the Document, you may at your option designate some or all of these sections as invariant. To do this, add their titles to the list of Invariant Sections in the Modified Version's license notice. These titles must be distinct from any other section titles.

You may add a section Entitled "Endorsements", provided it contains nothing but endorsements of your Modified Version by various parties–for example, statements of peer review or that the text has been approved by an organization as the authoritative definition of a standard.

You may add a passage of up to five words as a Front-Cover Text, and a passage of up to 25 words as a Back-Cover Text, to the end of the list of Cover Texts in the Modified Version. Only one passage of Front-Cover Text and one of Back-Cover Text may be added by (or through arrangements made by) any one entity. If the Document already

includes a cover text for the same cover, previously added by you or by arrangement made by the same entity you are acting on behalf of, you may not add another; but you may replace the old one, on explicit permission from the previous publisher that added the old one.

The author(s) and publisher(s) of the Document do not by this License give permission to use their names for publicity for or to assert or imply endorsement of any Modified Version.

A.5   COMBINING DOCUMENTS

You may combine the Document with other documents released under this License, under the terms defined in section A.4 above for modified versions, provided that you include in the combination all of the Invariant Sections of all of the original documents, unmodified, and list them all as Invariant Sections of your combined work in its license notice, and that you preserve all their Warranty Disclaimers.

The combined work need only contain one copy of this License, and multiple identical Invariant Sections may be replaced with a single copy. If there are multiple Invariant Sections with the same name but different contents, make the title of each such section unique by adding at the end of it, in parentheses, the name of the original author or publisher of that section if known, or else a unique number. Make the same adjustment to the section titles in the list of Invariant Sections in the license notice of the combined work.

In the combination, you must combine any sections Entitled "History" in the various original documents, forming one section Entitled "History"; likewise combine any sections Entitled "Acknowledgements", and any sections Entitled "Dedications". You must delete all sections Entitled "Endorsements".

A.6   COLLECTIONS OF DOCUMENTS

You may make a collection consisting of the Document and other documents released under this License, and replace the individual copies of this License in the various documents with a single copy that is included in the collection, provided that you follow the rules of this License for verbatim copying of each of the documents in all other respects.

You may extract a single document from such a collection, and distribute it individually under this License, provided you insert a copy of this License into the extracted document, and follow this License in all other respects regarding verbatim copying of that document.

A.7 AGGREGATION WITH INDEPENDENT WORKS

A compilation of the Document or its derivatives with other separate and independent documents or works, in or on a volume of a storage or distribution medium, is called an "aggregate" if the copyright resulting from the compilation is not used to limit the legal rights of the compilation's users beyond what the individual works permit. When the Document is included an aggregate, this License does not apply to the other works in the aggregate which are not themselves derivative works of the Document.

If the Cover Text requirement of section A.3 is applicable to these copies of the Document, then if the Document is less than one half of the entire aggregate, the Document's Cover Texts may be placed on covers that bracket the Document within the aggregate, or the electronic equivalent of covers if the Document is in electronic form. Otherwise they must appear on printed covers that bracket the whole aggregate.

A.8 TRANSLATION

Translation is considered a kind of modification, so you may distribute translations of the Document under the terms of section A.4. Replacing Invariant Sections with translations requires special permission from their copyright holders, but you may include translations of some or all Invariant Sections in addition to the original versions of these Invariant Sections. You may include a translation of this License, and all the license notices in the Document, and any Warrany Disclaimers, provided that you also include the original English version of this License and the original versions of those notices and disclaimers. In case of a disagreement between the translation and the original version of this License or a notice or disclaimer, the original version will prevail.

If a section in the Document is Entitled "Acknowledgements", "Dedications", or "History", the requirement (section A.4) to Preserve its Title (section 1) will typically require changing the actual title.

A.9 TERMINATION

You may not copy, modify, sublicense, or distribute the Document except as expressly provided for under this License. Any other attempt to copy, modify, sublicense or distribute the Document is void, and will automatically terminate your rights under this License. However, parties who have received copies, or rights, from you under this License will not have their licenses terminated so long as such parties remain in full compliance.

A.10   FUTURE REVISIONS OF THIS LICENSE

The Free Software Foundation may publish new, revised versions of the GNU Free Documentation License from time to time. Such new versions will be similar in spirit to the present version, but may differ in detail to address new problems or concerns. See http://www.gnu.org/copyleft/.

Each version of the License is given a distinguishing version number. If the Document specifies that a particular numbered version of this License "or any later version" applies to it, you have the option of following the terms and conditions either of that specified version or of any later version that has been published (not as a draft) by the Free Software Foundation. If the Document does not specify a version number of this License, you may choose any version ever published (not as a draft) by the Free Software Foundation.