VHDL 2018: New and Noteworthy

L. Lemiengre and H. Eeckhaut
Sigasi

Kerkstraat 108

9050 Gent-Brugge Belgium

***Abstract* - VHDL 2018 improves many aspects of the popular hardware design language.**

The most important improvement is the ability to bundle ports in an interface. This feature greatly reduces the verbosity of instantiations while improving the maintainability and clarity of the code.

The language was simplified where possible, restrictions were removed and various inconsistencies were resolved. New APIs, enhanced protected and generic types enable designers to build the next generation of verification libraries.

To achieve these improvements, the VHDL working group continues to build on the existing strengths of VHDL: strong typing, early bug detection and clear language semantics remain the key features of VHDL.

I. Introduction

We start by motivating why VHDL is still relevant today and how the language can evolve to remain relevant.

Next, we look at the two biggest features of the new revision: Interfaces and enhanced generics dramatically change the way VHDL is used. Improving type safety and readability while reducing verbosity. Next, we take a look at a number of small features and APIs that improve the usability of the language. Finally, we take a look at a new set of APIs and language features aimed at verification library designers.

II. Reviving the VHDL Standard

Before we dive into the new features, it is interesting to take a step back and examine if it still makes sense to evolve this thirty plus year old VHDL standard. Is VHDL still relevant in 2018? This is a question that the members of the VHDL working group had to answer.

In recent years, VHDL was pushed into the background as other languages have gotten more attention. It is rare to see VHDL-related talks at conferences. Some EDA companies [1] and some thought leaders [2] have even declared the language dead. The previous release, VHDL 2008, is still not completely supported by simulators, and synthesis support is rare. But despite the negative press and neglect by some tool vendors, VHDL is still a very popular language. So, why did it not disappear?

When users are asked what they like about VHDL they typically mention the strong type system and the clear semantics of the code. The code can be verbose, but it is also readable, with few surprises. VHDL is built in such a way that tools can detect a lot of issues at compile time. Many kinds of bugs are eliminated by design. This gives the designer more confidence that once he starts simulating the code, it actually works. Clear code means that VHDL users can easily exchange code. VHDL in its current form has some unique advantages. The challenge for the working group is to evolve the language whilst keeping these advantages.

To explain how VHDL achieves these advantages we need to look at how the language is built. From a language engineering point of view, VHDL is an interesting language. The core language, based on Ada, does not have any concepts related to hardware design. Instead, the core language only defines three things:

- Primitive and aggregate types
- Some predefined operations on these types
- A well-defined simulation model

Hardware oriented features are not defined directly in the language, but rather in the layer above it: in the IEEE library. From the point of view of the user however, this distinction is not observable. Complex user-defined types

and flexible literals, combined with rich operator- and subprogram overloading, make the data types described in the IEEE libraries look like data types which were built into the language.

This might look like a contrived way to make a hardware description language, but it has many advantages. First of all, the grammar only has to cover the core language, which results in a much smaller language. The semantics of a smaller language is easier to specify and simple specifications have less corner cases. The core language is also easier to implement, which lowers the chance of variation between implementations. Since the IEEE libraries are specified in the VHDL language itself, their provided implementation is their specification. This means that their behavior does not have to be described in the LRM and it eliminates variations in implementation. Another benefit is that the language is built to be extended. Users can define their own complex data types, extending the IEEE library. These extensions work like the built-in or provided IEEE data types. This allows users to extend the language without involving the VHDL language engineers.

There are also drawbacks to this approach. Many designers perceive VHDL as a very abstract language, closer to a software language than other hardware description languages. The type system feels very inflexible and types often have to be explicitly converted. This approach makes some features hard to add because the core language is oblivious of clocks and resets. Simulation performance can also be an issue, specialized built-in types would give the simulator more room to optimize. The source of these issues and their solutions are not in the scope of this paper, but the VHDL working group is aware of them.

We believe that separating the language and the IEEE library is, on the whole, beneficial for VHDL. If a proposal is made to change the core language, we first evaluate if it is possible to implement it as a library. For example: an important subject for hardware design languages is verification. The working group has made the explicit choice not to include verification features into VHDL 2018. Instead, the group decided verification should be implemented as a library.

Verification methodologies are evolving quickly. It is hard to justify adding constrained random generation into the core language when this methodology may someday be replaced with for example formal methods or something yet unknown. There are many verification methodologies and, depending on the design requirements, the used methodology should differ accordingly. In some cases simple unit testing may suffice, in other cases a variation of constrained random or formal may be required. Supporting all known verification methodologies with specialized syntax in the core language would make the language specification too big and complicated. This would mean it is harder to implement and would significantly increase the chance for bugs and inconsistencies. There already are several open source verification libraries developed for VHDL 2008. VUnit [3], OSVVM [4] and UVVM [5] have proven that a library-based approach to verification with VHDL is a good approach.

Instead of adding language level features to support verification in VHDL we chose to identify what these library authors were missing in the current language.

For all of the reasons above, we considered it worthwhile to create a new VHDL revision. With as little changes to the core language as possible, we significantly improve VHDL's capabilities for design and library development. The focus was on improving the core language. In a future revision we may focus on standardizing more IEEE libraries.

III. Realizing VHDL 2018

The work on VHDL 2018 started in 2014 by conducting a survey. To focus the standardization effort, people active on the VHDL mailing list were asked to rank a set of proposals. The results were very clear: interfaces were the most requested feature. At that point there were many different proposals to achieve this functionality. It took more than two years to reach a consensus about this feature.

Meanwhile other proposals of features were also discussed, distinguishable into two groups.

One group of features is aimed at streamlining the usability of VHDL. These are simple features and easy to implement, but they can have tremendous value to the user.

The second group are features aimed at library designers. These can be complicated but they enable the next generation of VHDL verification libraries. It is unlikely that VHDL designers will use these features, they are added specifically for library designers.

A.Interfaces

Interfaces are a central element in hardware design. There are many standardized interfaces like I²C, AXI or VGA and every design also has internally designed interfaces to connect various parts of a system. Unfortunately, these interfaces are cumbersome to model using VHDL. Typically, they are not explicitly defined. Instead their description is repeated on every entity. The only way to identify them is through some naming convention, e.g. all ports of the slave AXI interface could be prefixed with "slave_axi_0_".

This approach has many drawbacks. Since there is no centralized definition and the solution relies on naming conventions, it can be hard to identify the interface in complex entities. It is also hard to document an interface without a central definition. The interface is not only repeated for every entity that uses it, but also in every instantiation. All this duplicated code makes it very cumbersome to maintain the code. If you need to change the type or name of an element of the interface, this requires edits in many files, even in architectures that just pass the interface to an instantiation. This situation is unacceptable for a language that highly values strong typing and early bug detection, language level support for interfaces is needed.

Over the years designers have developed workarounds to model interfaces in VHDL. One solution is to model the interface as a record. But different elements of a record cannot have different port directions. There are two solutions to this problem: model this record as "inout" or split the record into multiple records, one for each port direction. The latter method is often called the Gaisler method [6].

Modelling an interface as an "inout" record has the advantage that the interface is defined in one place and it can easily be passed around. You can nest interfaces by nesting records and you can make an array of interfaces. This approach however, has one big drawback, the compiler can no longer check the port direction. For this reason, it is almost never used. The Gaisler method is often used. Unfortunately, it cannot handle nested interfaces and it still relies on some naming conventions.

When we examined how interfaces could be implemented, many approaches were examined. You can make directions part of the subtype or create a new kind of record or define a new container with different object kinds like constants and signals. We ended up choosing the simplest solution, which we coined "mode views". A mode view defines port directions for elements of a record. You can look at it as a user-defined mode for records, instead of writing "in" or "out" you refer to the "mode view".

```
package interfaces is
 type streaming_bus is record                -- the record definition
   valid : std_logic;
   data  : std_logic_vector(7 downto 0);
   ack   : std_logic;
 end record;

 view streaming_master of streaming_bus is  -- the mode view of the record
   valid, data : out;
   ack         : in;
 end view;

 alias streaming_slave is streaming_master'converse;
end;
```

In this "interfaces" package a record "streaming_bus" is defined. This record defines all the names and types of the elements of the record. The record elements "valid" and "data" are used to push data over the bus, the element "ack" is used to provide some back pressure. In the mode view "streaming_master" we create an interface for the record "streaming_bus". A port mode is applied to every element of the record. To maximize code reuse one record can have

many mode views. Because they are defined in a package, the definition can be reused. The record and the mode view do not have to be defined in the same package. Using the "converse" attribute we can invert the mode view directions in a single line. Alternatively, you can also explicitly define "streaming_slave" as follows:

```
view streaming_slave of streaming_bus is
 valid, data : out;
 ack         : in;
end view;
```

Let us take a look at how this interface can be used.

```
entity source is
 port(clk, rst : in std_logic; output : view streaming_master);
end;

entity processor is
 port(clk, rst : in std_logic;
      input    : view streaming_slave;
      output   : view streaming_master);
end;

entity sink is
 port(clk, rst : in std_logic; input: view streaming_slave);
end;

architecture a of e is
 signal clk, rst      : std_logic;
 signal input, output : streaming_bus;
begin
 producer  : entity work.source    port map(clk, rst, input);
 processing: entity work.processor port map(clk, rst, input, output);
 consumer  : entity work.sink      port map(clk, rst, output);
end;
```

In this example we have three instantiations that are connected using two busses. The instantiation "producer" reads some data and pushes it to the bus "input". This data is processed by the instantiation "processing" and pushed to the bus "output". Which is then consumed by the "consumer" instantiation.

Note that the view mode used in the entities "source", "processor" and "sink" is actually written in its short form. The long form is `view streaming_master of streaming_bus`. But in this example, referring to the type is redundant. It is already defined on the mode view and in this case we do not use a specific subtype. The need for a long form will become clear in the next example.

If we now want to make the element "data" in streaming bus generic in size we need to change some definitions.

```
type streaming_bus is record
 valid : std_logic;
 data  : std_logic_vector;
 ack   : std_logic;
end record;
```

"streaming_bus" is now unconstrained, this feature has been allowed since VHDL 2008. Our mode view definitions do not have to change. We can change the definition of the "source" entity to make the size of the bus explicit as a generic parameter.

```
entity source is
 generic(size : natural);
 port(
   clk, rst : in std_logic;
   output   : view streaming_master of streaming_bus(data(size downto 0))
 );
end;
```

Using constraint records subtypes and explicitly defining the subtype we reuse the mode view for multiple subtypes of the same record.

It is also possible to nest interfaces, e.g. you can combine the "input" and "output" interfaces of the "processor" entity into one interface.

```
type processing_element_rec is record
 input, output : streaming_bus;
end record type;

view processing_element of processing_element_rec is
 input  : view streaming_slave;
 output : view streaming_master;
end view;
```

Instead of writing "in" or "out" in the view we refer to the mode view that should be applied to the record element.

It is also possible to create arrays of interfaces, e.g. expanding the element "input" of the previous example to an array of slaves:

```
type streaming_bus_array is array (natural range <>) of streaming_bus;
type processing_element_rec is record
 inputs : streaming_record_array(0 to 9);
 output : streaming_rec
end record type;

view processing_element of processing_element_rec is
 inputs : view (streaming_slave);
 output : view streaming_master;
end view;
```

In this case the view mode is applied to an array with "streaming_bus" as array element. The definition has to be surrounded with parenthesis because the view mode is applied to an array and not a record. This syntax is similar to how resolution functions are applied to array types.

Finally, it is also possible to pass an interface to a procedure or a function. It is not possible to return an interface from a function.

```
procedure p(signal b : view streaming_slave; ...);
function  f(signal b : view streaming_slave; ...) return ...;
```

The ability to cleanly express interfaces is the most visible improvement in VHDL 2018. It drastically improves the readability and maintainability of VHDL designs. As shown in the examples, the provided features are very flexible and allow the designer to model any interface.

B. Enhanced generic types

VHDL 2018 improves generic types and subprograms. In VHDL 2008 generic types were introduced, these generic types can be bound to any type. In addition to the generic type, a number of generic operations can be provided with the type. For example:

```
entity max is
 generic (
   type element_t;
   type array_t;
   function maximum(l : array_t) return element_t
 );
 port (
   clk    : in  std_logic;
   input  : in  array_t;
   output : out element_t
 );
end;

architecture impl of max is
begin
 output <= maximum(input) when rising_edge(clk);
end
```

The operations passed with the generic type describe the capabilities of this type. Implicitly the "=" and "/=" operators are also passed in the generic list, for every generic type.

In VHDL 2018 we decided to make generic types easier to use by allowing the designer to constrain the type to a type class. The implicit declarations of that type class are then automatically available.

```
entity max is
 generic (
   type element_t is <>; -- "<>" means "scalar type"
   type index_t   is <>;
   type array_t   is array(index_t) of element_t
 );
 port (
   clk    : in  std_logic;
   input  : in  array_t;
   output : out element_t
 );
  end;
```

Now we have constrained "element_t" and "index_t" to be scalar types. Which means that all implicit declarations like "=", ">=", to_string, … and all attributes defined for scalars are available for these types. The subprogram "maximum" is implicitly defined for scalar types so it is no longer necessary to pass the subprogram explicitly. We have also constrained "array_t" to be an array type. Aside from scalar and array, VHDL 2018 defines seven more generic types, each with their own implicit operations such as maximum, to_string and "=". In conclusion, we have gained type-safety by constraining the generic types, and the solution is less verbose because many implicit operations don't have to be passed explicitly.

To make generic types even less verbose incomplete formal port types are introduced. In short, this allows us to describe a nested generic type without giving it a name.

```
entity max is
 generic (
   type array_t is array(type is <>) of type is <>
 );
 port (
   clk    : in  std_logic;
   input  : in  array_t;
   output : out array_t'element
 );
end;
```

This example can be improved further: the generic type definition can be moved to the port definition.

```
entity max is
 port (
   clk    : in  std_logic;
   input  : in  type array_t is array(type is <>) of type is <>;
   output : out input'subtype'element
 );
end;
```

VHDL 2018 allows incomplete formal types to be used on the port declaration, we have used this to declare the port "input". For consistency, port lists are now analyzed in order, just like generic lists have been since 2008. This means that you can refer to a previously defined port to declare a port, this is how the type of port "output" is declared. The result is that the declaration of the entity "max" is less verbose but also that the instantiation of "max" is less verbose. This is because no generic parameters have to be passed, the generic type arguments are inferred from the port list.

```
maximizer : entity work.max port map(clk, input, output);
```

The use case for these generic types is not limited to RTL. Dynamic data structures can now be developed as a library, these data structures are essential for verification libraries. In the next example we show how such a data structure may be used.

```
type int_list is new generic_list
  generic map(vect => integer_vector);

variable lst : int_list;
lst.addAll((1, 2, 3));  -- (1, 2, 3)
lst.add(10);            -- (1, 2, 3, 10)
lst.index_of(3);        -- returns 2
lst.remove_all(2);      -- (1, 3, 10)

type int_to_string is new generic_map
  generic map(key => integer, value => string);

variable lookup_table : int_to_string;
lookup_table.put(1, "this");  -- (1 -> "this")
lookup_table.put(8, "that");  -- (1 -> "this", 8 -> "that")
lookup_table.get(8);          -- returns "that"
lookup_table.exists(3);       -- returns false
```

To build these dynamic data structures several other features had to be added to the language. First of all, protected types can be parameterized with generic types, in this example "generic_list" and "generic_map" are protected types.

Many restrictions on where and how protected types can be used had to be lifted. Garbage collection has been added such that the designer does not have to manually delete these dynamic data structures after use. These data structures are not a part of the 2018 standard. We will let verification libraries experiment with them first.

In conclusion, the improvements that were made to generic types make them more type-safe and reduces their verbosity. They also enable new kinds of libraries.

B.Usability

Aside from improving some of the core constructs of VHDL it is also important to pay attention to the small, daily frustrations of design.

B.1 Conditional Expressions

Users have often missed a ternary operator in VHDL. For this version we have expanded the places where the "when-else" construct can be used:

1. Inside expressions: `y <= a xor b after (3 ns when FAST else 5 ns);`
   The when-else-expression must be in parentheses.
2. To initialize constants and attributes:
```
constant DELAY : time := 3 ns when FAST else 5 ns;
attribute RAM_STYLE of RegFile : signal is
  "distributed" when SMALL else "block";
```
3. In return statements:
```
return when condition; -- only return when the condition is true
return a when condition else b;
```

B.2 Conditional analysis

A preprocessor was added to perform conditional compilation. The preprocessor has access to six predefined variables: VHDL_VERSION, TOOL_TYPE, TOOL_VENDOR, TOOL_NAME, TOOL_EDITION, TOOL_VERSION. The syntax for the preprocessor closely follows the VHDL syntax, for example:

```
`if TOOL_VENDOR = "SIGASI" then
  attribute dont_touch of sig1 : signal is "true";
`endif
```

These predefined variables are also available as regular VHDL constants on the package std.env, making them usable in regular expressions as follows:

```
if env.tool_type = "SIMULATION" then ... endif;
```

This allows designers to work around tool issues that can not be solved within the VHDL language. Libraries can now target multiple VHDL versions or multiple toolchains, offering features depending on the environment.

B.3 Sequential declaration regions

Allowing designers to declare constants and variables inside sequential regions was a frequently requested feature. Initially, we only added a simple sequential block statement but after some experimentation, we found that adding sequential declaration regions to if statements and loop statements is more useful in practice.

In this example we show an if statement inside a for statement

```
p : process is
begin
  for i in some_vector'range then
```

```
    constant element : integer := some_vector(i);
  begin
     if element > CONST then
       variable result : integer;
     begin
       some_procedure(element, result);
       report result'image;
     end if;
  end for;
end process p;
```

In VHDL, variables can be synthesized in either combinatorial logic or registers, depending on how they are used. If the value of a variable is read in a clocked process, before it is written, the variable will result in a register. Sometimes, designers do this by mistake, leading to design errors. With sequential declarations we avoid this confusion: Variables declared in a sequential declarative part will always be combinatorial logic.

```
p : process is
 variable combinatorial_or_register : unsigned(8 downto 0);
begin
  if rising_edge(clk) then
    variable only_combinatorial : unsigned;
  begin
    ...
  end for;
end process p;
```

B.4 Bigger integer

The minimum size of integer has been increased from 32-bit to 64-bit.

B.5 Improved attributes

Objects now have direct access to the attributes of their type. This simplifies the use of many attributes.

Example: obtaining the string value of an object

```
report o'subtype'image(o); -- VHDL 2008
report o'image             -- VHDL 2018
```

All attributes were reviewed and many inconsistencies were resolved. For example, the "image" attribute is now available for records and arrays.

B.6 Attributes for PSL

PSL support was updated to the latest version (IEEE 1850-2010). In addition, it's now possible to interact with PSL directives. There are two ways to do this.

Designers can interact with PSL directives using two attributes. The "signal" attribute is used to read the value of the PSL directive and the "event" attribute to detect that the PSL directive has completed in this simulation cycle.

Through new subprograms in the std.env package, the verification library can check if any PSL asserts have failed, check that all PSL objects were covered, reset the state of PSL objects and more.

B.7 New and improved APIs

Several APIs to interact with the operating system were updated or added.

B.7.1 File API

Four features were added to the std.textio API:

- Files can now be opened in “read_write_mode”
- You can determine if a file is still open using the “file_state“ function
- You can determine and modify the size of a file using the subprograms “file_size” and “file_truncate”
- Random file access was added using the subprograms “file_position”, “file_seek” and “file_rewind”

B.7.2 File system API

In the package std.env several subprograms were added to interact with the file system.

- Directories can be explored using the “dir_open” subprogram and the “directory” data type
- Files and directories can be created and deleted

B.7.3 Date & time API

The date and time API was added to the std.env package. It has the following features:

- The ability to query the time since EPOCH as a real
- The ability to query time as a “time_record” record, using either the local timezone or UTC
- A minimal API to increment and decrement “time_record” objects
- The ability to pretty print time using “to_string”

```
type time_record is record
  microsecond : integer range 0 to 999_999
  second      : integer range 0 to 61;
  minute      : integer range 0 to 59;
  hour        : integer range 0 to 23;
  day         : integer range 1 to 31;
  month       : integer range 0 to 11;
  year        : integer range 1 to 4095;
  weekday     : dayofweek;
  dayofyear   : integer range 0 to 365;
end record time_record;
```

B.7.4 Environment variables

A minimal API to query environment variables was added to std.env.

B.8 APIs for library builders

B.8.1 Introspection

The introspection API allows users to inspect arbitrary data types. This feature is oriented towards verification libraries and not intended for RTL.

Introspection consists of two parts: attributes to convert any object into a generic mirror object and a library std.reflect that can be used to inspect mirror values and types. The API offers a type-safe method to inspect values at runtime.

The most common use case for introspection is to convert arbitrary values into a known type. For example: converting a VHDL value into a string representation, writing it to a file in the JSON format or flattening any record into a std_logic_vector.

The introspection API is based on the mirror-based reflection research by Gilad Bracha [7]. This is a proven approach that has been used in many other languages [8][9]. In this initial release it is not possible to create or modify values, in a future revision this functionality could be added.

B.8.2 Asserts

This API is similar to the new PSL API. You can check how many asserts have failed, modify the failure messages, clear assertion results and much more.

This API is vital for verification libraries. The subprograms were added to the package std.env.

B.8.3 Call path

Two additions were made to provide better debug information to the users of verification libraries.

The API provides functions to retrieve the file name, file path and line in the current VHDL file. Another set of subprograms and types can be used to retrieve and inspect call path, also called stack traces.

IV. Conclusion

In this paper we covered the most important features of VHDL 2018. There are big improvements to both RTL and verification. We did so by adding interfaces, improving generic types, streamlining the language and by increasing support and available tools for verification library designers.

The result will breathe new life into the VHDL community. The revision will be balloted and released in 2018. The finished proposals are publicly available on the VHDL working group wiki [10].

Acknowledgements

This language revision was the result of continuous feedback from many experienced hardware designers and library builders who participate in the working group. The working group's resources are limited, as such, most of the work was done by voluntary contributors. We had to reject many proposals, often because we did not have the time to fully explore them. But focusing on the smallest features that could have the biggest impact has been a successful strategy.

We would like to thank all members of the VHDL working group, in particular the chair: Jim Lewis and vice chairs: Patrick Lehmann and Rob Gaddi. Without their relentless commitment VHDL 2018 would never have seen the light of day.

References

[1]Michael Santarini, "*Synopsys executive predicts end of VHDL*" https://www.eetimes.com/document.asp?doc_id=1216860, 4/11/2003
[2]John Cooley, *"VHDL, the new Latin"* https://www.eetimes.com/document.asp?doc_id=1216865, 4/7/2003
[3]"VUnit" https://vunit.github.io
[4]"OSVVM" https://osvvm.org
[5]"UVVM" https://bitvis.no/dev-tools/uvvm
[6]J. Gaisler, "A structured VHDL design method" http://www.gaisler.com/doc/vhdl2proc.pdf
[7]G. Bracha, D. Ungar, "Mirrors: design principles for meta-level facilities of object-oriented programming languages" In procedings of the 2004 ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages, and Applications, pp. 331-344, ACM, New York (2004)

[8]"Reflection in Dart with Mirrors" https://www.dartlang.org/articles/dart-vm/reflection-with-mirrors
[9]"Mirror C++ reflection utilities" http://kifri.fri.uniza.sk/~chochlik/mirror-lib/html/
[10]"VHDL2017 proposals" http://www.eda-twiki.org/cgi-bin/view.cgi/P1076/VHDL2017