Introduction

The VUnit identity package (`id_pkg`) enables the creation of a hierarchical organization of named objects within a testbench. It expands upon the functionality provided by VHDL's `simple_name`, `instance_name` and `path_name` attributes and eliminates some of the limitations associated with string-based naming conventions.

Limitations of Name Strings and Name Attributes

When naming a testbench object, such as a verification component (VC), we can utilize one of the VHDL name attributes. In the example below, we have a testbench with two VCs, X and Y, each with a `name` generic that we assign the `path_name` of the VC entity instantiation.

```
architecture vunit_style of tb_dut is
  -- Declarations
begin
  stimuli : process
  begin
    ...
  end process;

  my_dut : entity work.dut
    port map(
      ...
    );

  vc_x : entity work.verification_component_x
    generic map(
      name => vc_x'path_name
    )
    port map(
      ...
    );

  vc_y : entity work.verification_component_y
    generic map(
      name => vc_y'path_name
    )
    port map(
      ...
    );
end architecture;
```

Note

A limitation with using name attributes is that, at the time of writing, not all simulators support assigning the `name` generic a value that references the label of the instantiation itself.

The VCs use the `name` generic to tag log messages. A basic message, without using VUnit logging functionality, could appear as follows:

```
Writing 0xDEADBEEF to address 0x12345678 (:tb_dut:vc_x:).
```

In this case, the name produced by the `path_name` attribute, `:tb_dut:vc_x:`, reflects the logical structure of the testbench and clearly identifies the producer. However, `path_name` is limited to code structure and knows nothing about the logical structure of the testbench. These are not necessarilty the same. For example, if we want to reorganize our testbench by moving some declarations in the architecture declarative part to a more local location, we can do this by adding a block statement around the VC instantiations.

```
local_declarations : block is
  -- Local declarations of signals, constants etc
begin
  vc_x : entity work.verification_component_x
    generic map(
      name => vc_x'path_name
    )
    port map(
      ...
    );

  vc_y : entity work.verification_component_y
    generic map(
      name => vc_y'path_name
    )
    port map(
      ...
    );
end block;
```

We haven't changed the logical structure, `vc_x` is still part of `tb_dut` and should be presented as such. However, the code structure has been changed and so has the naming:

```
Writing 0xDEADBEEF to address 0x12345678 (:tb_dut2:local_declarations:vc_x:).
```

To avoid this problem, as well as the problem of limited simulator support, we could opt to abandon the name attributes and instead use manually crafted name strings. Manually crafting name strings also provides more flexibility as the name is no longer limited by the rules for identifier naming. For example, we could name the VC `:tb_dut:verification component X:` if preferred.

Whether using a name attribute or providing the name explicitly, the concept of hierarchy is determined by a naming convention and not by the `string` type itself. By using a dedicated type, we can create a more explicit concept and also enable more advanced functionality.

Identity Basics

To overcome the issues discussed in the preceding section, `id_pkg` provides an `id_t` type, which is compatible with VHDL name attributes in the sense that we can create an identity from such an attribute:

```
variable vc_x_id : id_t;

vc_x_id := get_id(vc_x'path_name);
```

Or from a string if the logical structure doesn't match the code structure:

```
vc_x_id := get_id(":tb_dut:vc_x:");
```

We can also omit the leading and trailing colons for brevity:

```
vc_x_id := get_id("tb_dut:vc_x");
```

The identity returned when calling `get_id` represents the last component in the hierarchical path. Calling `name (vc_x_id)` will return the name of that component and `full_name(vc_x_id)` returns the full path. However, identities are created for each component in the path, and the parent identity can be obtained by invoking the `get_parent` function:

```
print("Name = " & name(vc_x_id));
print("Full name = " & full_name(vc_x_id));

parent_id := get_parent(vc_x_id);
print("Parent name = " & name(parent_id));
```

```
Name = vc_x
Full name = tb_dut:vc_x
Parent name = tb_dut
```

Calling the function `get_id` only creates identities for the components that are missing in the path provided to the function. For example, invoking `get_id` with `vc_y'path_name` will not create a new identity for `tb_dut` since that identity already exists after our previous invokation:

```
vc_y_id := get_id(vc_y'path_name); -- Creates an identity for vc_y but not for tb_dut.
```

Another way to add identities is to use `get_id` with a parent ID parameter. For example, to create the identity for `vc_y`, we can use the following equivalent code:

```
vc_y_id := get_id("vc_y", parent => parent_id);
```

To gain a better understanding of the identities that have been generated by previous `get_id` calls, we can utilize the `get_tree` function to view the identity tree with a given identity as its root. For example,

```
print("This is the tb_dut tree:" & get_tree(parent_id));
```

will output:

```
This is the tb_dut tree:
tb_dut
+---vc_x
\---vc_y
```

The `get_tree` function returns a string starting with a linefeed character (LF) to align the root line of the tree with its other elements. To omit this initial LF character, set the optional parameter `initial_lf` to false.

We can also call `get_tree()` without any parameters to view the full identity tree. This provides a comprehensive overview of all the identities created in user code, by third-party IPs, and in VUnit itself. An example of the output is provided below:

```
This is the full identity tree:
(root)
+---default
+---vunit_lib
|   +---dictionary
|   \---event_pkg
+---check
+---runner
\---tb_dut
    +---vc_x
    \---vc_y
```

At the root of the tree is a symbol `(root)` which represents the predefined `root_id`. `root_id` has no name but is given a symbol in the tree representation for clarity. The lack of name means that we cannot create a new identity with no name as that is already taken.

We can easily check if an identity is taken by calling the `has_id` function with either the full name of the identity or a partial name relative to its parent identity. For example:

```
has_id("") = true
has_id("tb_dut:vc_x") = true
has_id("vc_y", parent => get_id("tb_dut")) = true
has_id("vc_x") = false
```

Structuring Identities

The identity package does not place any limitations on what we use identities for. However, there are a few recommendations:

1. All identifers should have a primary owner, the object that the identity is associated with. For instance, in our prior examples the identity was associated with a verification component.
2. An identity can be used by objects other than its owner, provided that these objects are acting on the owner's behalf. For example, a verification component can create a logger from its identity. The logger logs messages on behalf of the verification component and can share its identity.
3. Use parent-child relationsships when the parent object is composed of child objects. For example, a bus protocol checker can be a standalone VC that monitors transactions on a bus to ensure that they are compliant with the protocol. As a standalone VC, it can have an identity such as `protocol_checker`. However, if the protocol checker is built into a bus initiator VC, capable of initiating read and write transactions, then a more descriptive identity such as `bus_initiator:protocol_checker` is more appropriate.

4. If your object is an entity, it should have an identity generic. The entity itself cannot determine which functional hierarchy it belongs to, so this must be specified externally. The generic should have `null_id` as the default value. If no value is assigned, the entity is free to choose its own identity.

Searching the Identity Tree

The `get_tree` function collects identity names by traversing the full tree. We can also create our own custom tree-traversing functionality by leveraging the `get_parent`, `num_children` and `get_child` functions. The `num_children` function returns the number of children identities a given identity possesses and we can retrieve each of these children identities by calling `get_child` with an index in the range [0, number of children - 1]. For example:

```
num_children(get_id("tb_dut"))) = 2
name(get_child(get_id("tb_dut"), 1)) = vc_y
```

To further illustrate these functions we can examine how `vc_x` handles the situation when its `id` generic hasn't been assinged any value and defaults to `null_id`. Rather than simply taking a fix identity name which would be shared by all instances, or an `instance_name` which may or may not yield a good representation, it creates another logical structure based on the format <company name>:<VC name>:<instance number>. The instance number is calculated by searching the company identity space for other existing instances of `vc_x`. If n instances are found, the new instance is assigned number n + 1.

```
if id = null_id then
  acme_corp_id := get_id("Acme Corporation");
  num_vc_x := 0;
  for child_idx in 0 to num_children(acme_corp_id) - 1 loop
    if name(get_child(acme_corp_id, child_idx)) = "vc_x" then
      num_vc_x := num_vc_x + 1;
    end if;
  end loop;
  vc_x_id := get_id("vc_x", parent => acme_corp_id);
  my_id := get_id(to_string(num_vc_x + 1), parent => vc_x_id);
  logger := get_logger(my_id);
else
  logger := get_logger(id);
end if;

...

debug(logger, "Writing 0x" & to_hstring(data) & " to address 0x" & to_hstring(addr) & ".");
```

With only one `vc_x`, the instance will be designated as number 1.

```
10000000 fs - Acme Corporation:vc_x:1 -   DEBUG - Writing 0xDEADBEEF to address 0x12345678.
```

In this example we were able to search for other instances of `vc_x` by searching the identity namespace. Had there not been a naming convention for the `vc_x` instances, it would not have been possible. However, not all resources have such a convention. For instance, if a VC wants to use the closest existing logger among its ancestors, there is no naming convention to rely on. In such cases, `get_logger` is of no use as this procedure will only create a new logger if it doesn't already exist. Fortunately, there is another function, `has_logger`, which can be used to query if there is a logger for an existing identity. All in all, it is generally recommended that any resource which utilizes identities should also provide methods for determining the existence of such a resource for a given identity.