# Macros to the Rescue

## André Winkelmann, Verilab GmbH
## Thorsten Dworzak, Verilab GmbH

The e-language was designed from the ground-up for the sole purpose of verification. Today an RTL design database is more than the sum of its HDL sources - it is more heterogeneous than ever, with the SoCs assembled using executable specifications together with meta-data in many different formats and from diverse suppliers. Although HVLs (Hardware-Verification Languages) have had a head start to keep up with these changing demands, verification is becoming more and more akin to a software development task. As an illustration of this fact, keyed lists (also known as hash tables) are a great tool, but why should the programmer need to care about the key management themselves? It is a reasonable expectation that an HVL should offer the same features that Perl or Ruby has to offer. With the so-called "define-as-computed" macros, the *e*-language already has powerful extension capabilities that allow the engineer to create expressive short-cuts in order to hide the details of such low-level tasks as the key management example. These make code more concise and easier to understand. This paper presents a collection of macros which will simplify the everyday programming tasks that many verification engineers inevitably face sooner or later. Macros allow the definition of new actions, expressions, struct members, coverage items and a lot more. Other examples shown will include a repetition operator for scalar vectors (similar to that feature of Verilog), advanced regular expression matching against a list of regular expressions and a new cover item macro for time values as well as scalars larger than 32 bits. The reader will be given a thorough insight into how the macros have been implemented including coding techniques for helper methods. The whole macro library will be made available as open source for everyone to download. Finally, examples showing the "before" and "after" code will be presented. Overall the paper highlights how macros increase both the

expressiveness and readability of verification code. Such code is faster to write, and easier to understand and maintain.

## INTRODUCTION

The motivation for the macro library is to improve on some of the shortcomings of the *e*-language. For example we aim to provide easy-to-use hashes, add more list- and string-processing functions, and improve the limited coverage API.

## MACROS

The *e*-language provides three different kinds of macro. The *define* and *define-as* macros are simple text replacements. While the *define* type is comparable to parameter-less Verilog pre-processor macros, *define-as* macros are parameterizable and syntactically checked.

The third macro type is the *define-as-computed* macro which is implemented as *e*-code to compose the macro expansion string. This allows us to add true language extensions.

Both the *define-as* and the *define-as-computed* macros are assigned a syntactic category (like *struct member* or *expression*) and thus have to expand to a complete *e*-code construct (see also [1]). For example the macro in figure 1 shows an implementation of the Ruby times method. It gets the action category assigned, since the expanded code, a for loop, is an action.

```
define <vlab_repeat_loop'action> "<exp>\.times <block>" as
{
  for it from 1 to <exp> <block>;
};
```

**Figure 1: Implementation of "vlab_repeat_loop" macro**

# TOOLS FOR WRITING MACROS

## Auxiliary Code

The macro expansion happens during load-time whenever the parser finds a match. This means the e-code of the *define-as-computed* macros is executed – so the question is: What is the lexical or run-time context for this code? Every type definition loaded prior to the macro definition is known of course. But at this time, the instance tree has not yet been generated or built.

```
extend sn_util {
   !my_aux_data: vlab_util_aux_data_s;
   init() is also { my_aux_data = new };
   aux_method() is { … };
};
```

**Figure 2: Adding auxiliary code**

However, the top-level object *sys* does already exist, and so does the singleton *sn_util*. This can be used to add global fields and methods accessible from the macro code. Example: the method in figure 2 can be accessed via

`util.aux_method()`

from anywhere. Care has to be taken due to the global namespace. It is recommended to only add distinctive names, e.g. prefixed (like *vlab*).

## Debug Messages

Complex macros with lots of input parameters need to perform checks on these and provide the user with differentiated error messages. Specman provides the built-in methods `get_current_line_num():int` and `get_current_module():rf_module` which can be used to display information about the location of the macro call (as opposed to where it is defined).

```
define <my_macro'statement> "(<MATCH>(<COMMAND>add|change)
foo bar_name=<name> <block>)" as computed{ … };
```

**Figure 3: Using <MATCH> in macro definition**

Furthermore, the match expression in the macro definition can be accessed by using submatch labels. These do not change the matched expression. E.g. in

figure 3 <MATCH> returns the entire matched part in the macro call which can be used in the debug/error messages. <COMMAND> returns either “add” or “change”.

### Combining Macros

Advanced macros can utilize other basic macros. To do this, the file defining the basic macros must be loaded first - before the file containing the advanced macros is loaded. It is a similar requirement to load define as computed macros before any code using them.

```
var agent_name: string;
...
if agent_name ~~ qw( /_master_/ /_slave_/ ) {
  outf("... %s is a master or slave.", agent_name);
};
```

**Figure 4: Nested macro calls**

Figure 4 shows two macros from the library in one expression: the `qw()` method and the `~~` operator. If the variable `agent_name` matches any of the regular expressions `/_master_/` or `/_slave_/`, it will be printed.

## The “vlab_util” Library

This chapter will present some more macros that are part of the vlab_util library.

### Hash Pseudo-Methods

```
var kl: list (key: name) of element_t;
var new_elem: element_t = new with
   { .name = "foo"; .value = 3141 };

kl.key("foo") = new_elem;
```

**Figure 5: Adding to a Hash**

The original implementation of keyed lists has a major drawback compared to e.g. Perl or Ruby: it allows for duplicated keys. We created two macros for adding and deleting keys. Figure 5 gives an example for adding an entry to a keyed list, ensuring unique keys. Any old entry with the same key will be silently overwritten.

The implementation of this macro can be seen in Figure 6. Note that the macro expansion creates a variable `idx` which is local to the scope of the expanded macro.

```
define <vlab_add_to_keyed_list'action>
   "<name>.key\(<key'exp>\)[ ]=[ ]<val'exp>" as computed {
   var rl: list of string;
   var kl : string = str_expand_dots(<name>);
   var key: string = str_expand_dots(<key'exp>);
   var val: string = str_expand_dots(<val'exp>);

   rl.add(appendf("var idx: int =  %s.key_index(%s);",kl,key));
   rl.add(append ("if idx != UNDEF {"));
   rl.add(appendf("   %s.delete(idx);", kl));
   rl.add(append ("};"));
   rl.add(appendf("%s.add(%s);",kl, val));

   result = appendf("{%s};", str_join(rl, " "));
};
```

**Figure 6: Implementation of "vlab_add_to_keyed_list" macro**

## List Pseudo-Methods

```
for each in (my_list.all_indices(it < 2).reverse()) {
   list.delete(it);
};
```

**Figure 7: Deleting list elements**

Deleting more than one list element has to be done from tail to head because the built-in `delete()` list-method modifies the list itself. See Figure 7, on how to traditionally delete all list elements that are smaller than 2.

A new pseudo-method simplifies this. Using `my_list.delete_all(it < 2)` makes the task a one-liner.

## Perl-like String Creation

We added two macros to automatically quote strings: `qw ( <string1> <string2> … ): list of string` and `qs ( <string> ): string`.

```
keep paddr.hdl_path()   == qs( paddr     );
keep pwrdata.hdl_path() == qs( writedata );

var agents: list of string = qw( MP3_MASTER USB_HUB1 );
```
**Figure 8: Creating strings**

See example Figure 8 where a set of ports is constrained (this is particularly useful for alignment when writing tabular code) and a list of strings is assigned. Note that without the macro you would have to write

```
var agents: list of string = {"MP3_MASTER"; "USB_HUB1"};
```

which is difficult to get first-time-right.

## Ruby-like OOP Methods

```
n.times { do seq keeping { .driver == ahb_drv } }

my_agents.each {
  it.active_passive = PASSIVE;
  bind(it.pmp.paddr, empty)
};
```
**Figure 9: Ruby-like methods**

Ruby offers some very concise constructs that we can model using macros. E.g. the *times* and *each* constructs. Figure 9 depicts two examples of repeating a block <n> times and applying a block to each entry of a list.

These are easy to implement, see figure 1 from the beginning which shows the *times* macro as an example.

## Coverage Macros

The authors implemented a macro to overcome the limitation *e* of the coverage API (no item wider than 32 bit). The macro accepts any scalar of any widths. One can define min, max boundaries and the number of buckets created within these bounds. E.g. to cover a time variable with 4 buckets, you can use

```
vlab_cov_item t using
   min = 100 ns, max = 200 ns, num_of_buckets = 4;
```

This will create four coverage buckets for the cover item `t`.

### Operators

In Verilog exists a useful repetition operator that allows constructs like

```
reg xyz = {2{3'b101}};
```

which assigns `'b101101` to `xyz`. Using the vlab_util library, you can express this similarly as

```
var xyz: uint = 2***(3'b101);
```

The implementation makes use of auxiliary code, see Figure 10.

```
extend sn_util {
  vlab_repetition(
       factor: uint, exp: list of bit): list of bit is {
    var i: uint;
    for i from 1 to (factor) { result.add(exp); };
  };
};
define <vlab_repetition_op_scalar'exp>
  "<factor'exp>***\(<rep'exp>\)" as computed {
  result = appendf("util.vlab_repetition(%s, %%{%s})[:]",
    str_expand_dots(<factor'exp>),
    str_expand_dots(<rep'exp>));
};
```

**Figure 10: Implementation of repetition operator**

## RESULTS

The source-code of the vlab_util library is licensed under Apache 2.0 and is available at bitbucket.org [2] (archive *verilab/vlab_util*). It contains some more macros which have not been described.

## SUMMARY

Increasingly complex test environments require more high-level constructs than what the *e*-language has to offer until now. The extensive macro capabilities make it possible to overcome such limitations. The authors have created a macro library with useful constructs that allow more concise coding.

# REFERENCES

[1] Dworzak, Thorsten. 2013. Using the Extension Capability and the Reflection Interface of Specman/e for Automatic Memoization. <http://www.verilab.com/files/cdnlive2013_TDworzak_Memoization_Paper.pdf>

[2] Winkelmann, André, Dworzak, Thorsten. 2014. The Bitbucket repository. <http://bitbucket.org/verilab/vlab_util>