Macros to the Rescue

André Winkelmann
Thorsten Dworzak
Verilab GmbH

CDNLIVE℠
Cadence User Conference 2014
EMEA – Munich, Germany—May 19-21

Code Readability Before

What is the code supposed to do?

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

Code Readability After

What is the code supposed to do?

```
list.delete_all(it < 2);
```

What is a Macro in *e*?

- A powerful code generator

```
define <name'category> "match" as { replacement };
```

- Some name
    - Where the macro is allowed to appear
        - “Regular” expression match
            - Direct replacement

Example: Delete Elements of a List

- New list pseudo-method: list.delete_all(condition)

```
list.delete_all(it < 2);
```

- Expands to

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

- Implemented as

```
define <vlab_del'action> "<list'exp>\.delete_all\(<filter'exp>\)" as
{
  for each in (<list'exp>.all_indices(<filter'exp>).reverse()) {
     <list'exp>.delete(it);
  };
};
```

What is a Computed Macro in *e*?

- An even more powerful code generator

```
define <name'category> "match" as computed {result=rpl_str};
```

- Some name
  - Where the macro is allowed to appear
    - “Regular” expression match
      - “**Computed**” replacement string

Remember to use str_expand_dots()

Example: Extend Enum

- New statement: vlab_extend_upper

```
type myT: [];
vlab_extend_upper myT: upcase_me;
```

- Expands to

```
extend myT: [UPCASE_ME];
```

- Implemented as

```
define <upper'statement>
   "vlab_extend_upper <enum'name>: <elem'name>" as computed {

  result = append(
     "extend ", <enum'name>, ": [", str_upper(<elem'name>), "];„
  );
};
```

Macros vs. Computed Macros

- Macros
  - Simple code replacement
  - Like a template
- Computed Macros
  - Transform the match expression
  - Full usage of *e*-code inside the macro
  - E.g. debug output, reflection API, own parsers

Shortcomings of *e*

- Hard to use hashes
  - Better solved in Perl or Ruby
- Missing list functions
- Missing Systemverilog goodies
  - If then else in constraints
  - Repetition operator
- Limited Coverage API

BUT: can be solved with macros!

Hash Macros

Add or delete a hash entry

- Hash.key() = <val>
- Hash.key_del(<key>)

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

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

- No need to test for existence of “foo” anymore

Ruby like OOP

Ruby offers some very concise constructs that we can model using macros, e.g.

- 5.times { do something with it }

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

- List.each { do something with it }

```
my_agents.each {
  it.active_passive = PASSIVE;
  bind(it.pmp.paddr, empty)
}
```

If-then-else as an Expression

- Systemverilog allows if-then-else in constraints
- *if_expr* Can be used to replace the ternary (c?t:f) operator to make expressions more readable:

```
keep
  if_expr (m_slave_or_master == MASTER) {
    if_expr (m_ocp_profile.burstlength == 0) {
      m_precise_burst_size == 1;
    } else {
      m_precise_burst_size == ipow(2,m_burst_pwr2);
    };
  } else {
    m_precise_burst_size == 0;
  };
```

List of match expressions

```
define <ternary'exp>
  "if_expr <cond'exp> {<cond_if'exp>;...}
   else {<cond_else'exp>;...}„
as computed {
…
  result = append(result, "(",
    str_join(<cond_if'exps>, ") and (").
    ")„
  );
};
```

- {<cond_if'exp>;...} denotes a list of expressions separated by semicolon
- <cond_if'exp**s**> denotes a *list of string*
    - Gives access to each <cond_if'exp> in a *as computed* macro

Repetition Operator

In Verilog: reg xyz = {2{3'b101}}

➔ xyz = 'b101101

In vlab_util:

- factor***(exp)

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

- Expands to

```
var xyz: uint = util.vlab_repetition(2, %{3'b101})[:];
```

Auxiliary Code

```
extend sn_util {
   vlab_repetition(
      factor: uint,
      exp: list of bit
   ): list of bit is {...};
};
```

- **sn_util**: singleton that is already generated at time of macro expansion

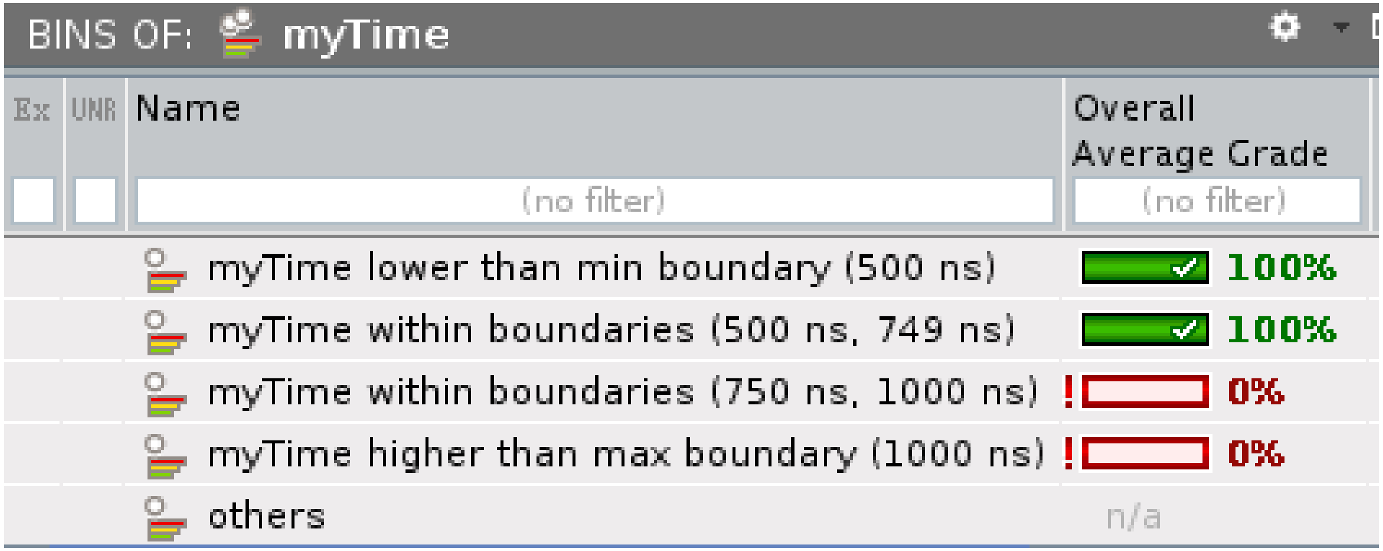
- **util**: *e* built in variable which gives access to singleton

```
util. vlab_repetition();
```

Coverage items

Coverage of time and scalars beyond 32 bits

```
cover cover_e {
  vlab_cov_item myTime using
    min= 500 ns,
    max=1000 ns;
    num_of_buckets=2;
};
```



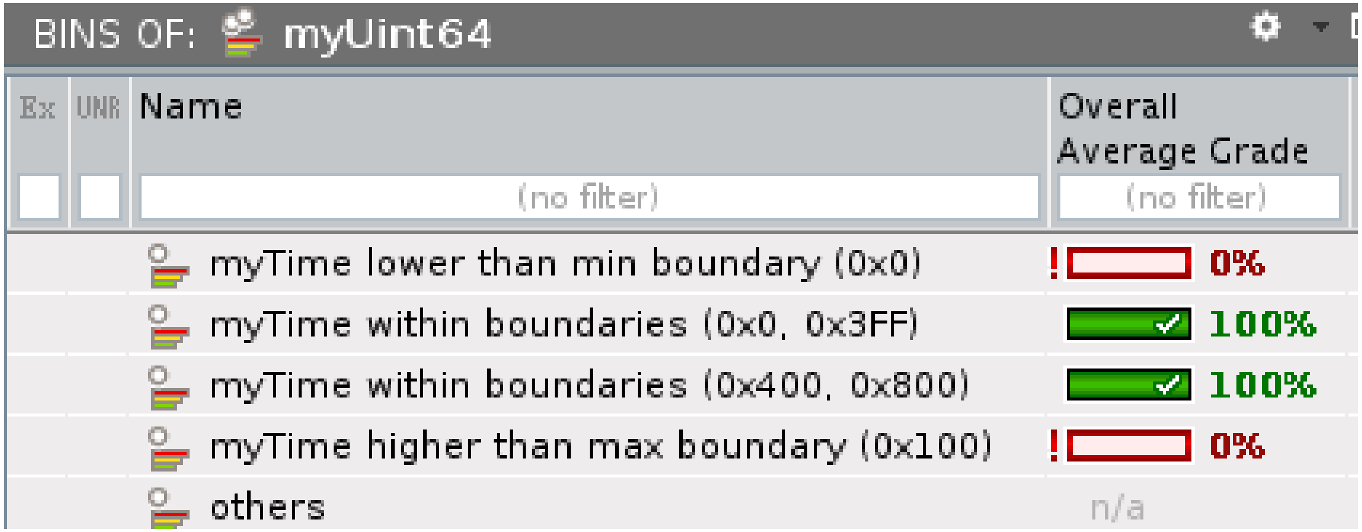
BINS OF: myTime

Ex	UNR	Name	Overall Average Grade
		(no filter)	(no filter)
		myTime lower than min boundary (500 ns)	✓ 100%
		myTime within boundaries (500 ns, 749 ns)	✓ 100%
		myTime within boundaries (750 ns, 1000 ns)	! 0%
		myTime higher than max boundary (1000 ns)	! 0%
		others	n/a

Coverage items

Coverage of time and scalars beyond 32 bits

```
cover cover_e {
  vlab_cov_item myUint64 using
    min=0x0,
    max=0x1234_5678_abcd,
    num_of_buckets=64;
};
```



BINS OF: myUint64

Ex	UNR	Name	Overall Average Grade
		(no filter)	(no filter)
		myTime lower than min boundary (0x0)	! 0%
		myTime within boundaries (0x0, 0x3FF)	✓ 100%
		myTime within boundaries (0x400, 0x800)	✓ 100%
		myTime higher than max boundary (0x100)	! 0%
		others	n/a

Coverage items

- Expands to

```
cover cover_e {
  item myTime: uint = util.vlab_get_cov_range(myTime, 500 ns, 1000 ns, 2)
  using ranges = {
    range([0], "myTime lower than min boundary (500 ns)" , UNDEF,  1);
    range([1], "myTime within boundaries (500 ns, 749 ns)"  , UNDEF,  1);
    range([2], "myTime within boundaries (750 ns, 1000 ns)"  , UNDEF,  1);
    range([MAX_UINT], "myTime higher than max boundary (1000 ns)", UNDEF,  1);
  };
};
```

- Calculates range descriptions
- Generates list of ranges based on num_of_buckets
- Uses auxiliary code

Debug Messages

- Complex macros (e.g. the coverage macro) need to test input parameters
- Present the user differentiated error messages
- Submatch labels are useful:

```
… “(<MATCH>vlab_cov_item <name> …)” as computed {
    out(<MATCH>);
};
```

- Would print: “vlab_cov_item myTime …” for each occurrence of the macro in the code

Further Tools for Writing Macros

- Enhance debug messages
- get_current_module(), get_current_line_num()
- Multi level macros
  - Load basic macros in first file
  - Then load advanced macros which utilize basic macros in second file
- “define as computed” macros have access to already loaded/extended types (e.g. enum extension)

Macros in the Library

- Hash (keyed lists) pseudo-methods
- Pattern matching
- Bit width of scalar
- Perl like string creation
- If then else expressions
- List pseudo-methods
- Ruby like OOP methods
- Coverage beyond 32 bit scalars

Summary

- The vlab_util library extends the e language to enhance the programmer's productivity
- The library is Open Source (Apache 2.0)
  - The library can be downloaded from https://bitbucket.org/verilab/vlab_util

Contributions welcome!