

Support for the *e* Language Macros in DVT

by Cristian Amitroaie

A unique and powerful feature of the *e* language is the “*define as*” macros. It allows programmers to create new syntactic constructs like statements, actions, and expressions using simplified regex patterns enhanced with lexical terminals. It may sound sophisticated, but once you play with a few macros, you'll pretty soon enjoy this natural capability of enhancing the *e* language. Actually, this is how the *e* language has evolved. For example, the sequence constructs, which are now part of the standard language, were initially macros.

Nevertheless, “*define as*” macros can be so addictive, that many of us tend to abuse them and forget other programming patterns. For example, quite often, a macro can be used to hide copy/paste instead of using OOP inheritance. And by the way, when I say “*define as*” macros, I also include the all mighty “*define as computed*” macros, which allow you to implement whatever processing in order to “generate code”.

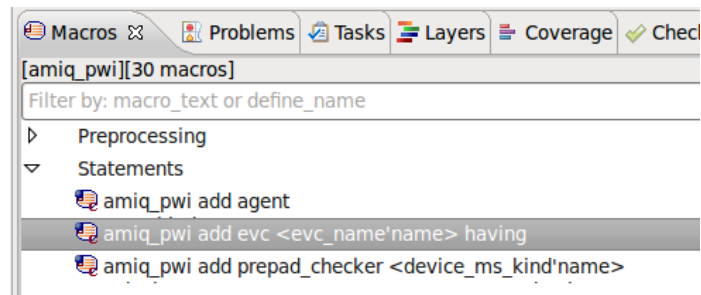
My point is that macros are very powerful, but one needs to be in control, so that this extra power doesn't actually fire back. This could be obtained with tools that help us work with both preprocessing directives and “*define as*” macros. Here is where DVT (Design and Verification Tools) gets in the picture. This integrated development environment (IDE) provides features that allow you to manipulate macros in many ways, from browsing to debugging. Here are a few examples:

- *Show me all macros*
- *Show me where this macro is used*
- *I want to see what this macro is going to be replaced with (Apply Preprocessing)*
- *I have an error inside a nested macro, please take me to the source of the problem (Macro Reparse Stack)*
- *Show me the macro value (e.g. BUS_WIDTH)*
- *Show me the active code in ifdef blocks*
- *Jump to macro definition (by hyperlink)*
- *Show me the macro description*
- *Autocomplete*

The following paragraphs summarize the DVT support for the aforementioned use cases.

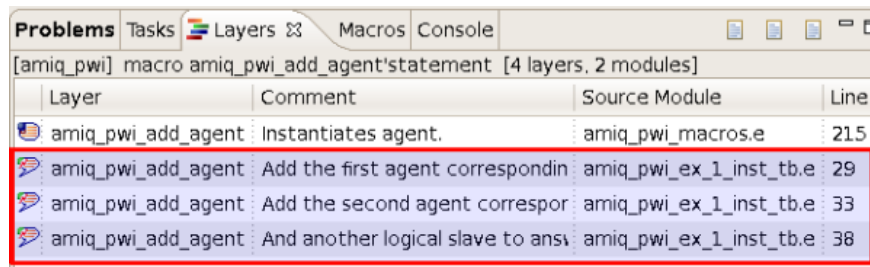
Macros View – To Locate a Macro

The *Macros View* presents all the preprocessing, “*define as*”, and “*define as computed*” macros. You can use it to browse all the macros in your code and quickly jump to a macro definition.

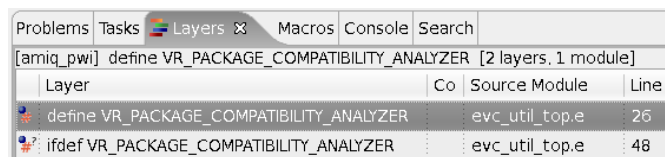


Layers View – To Locate All Places Where a Macro is Used

In the *Layers View*, you can see all the places where a macro is used in your code. This can be very useful particularly, when you want to change some macros and must understand what you break if you are not careful. Or maybe, when you just want to completely remove a macro and use another programming pattern instead of it. You need to be sure you understand how the macro is currently used before refactoring the code.



Of course, for preprocessing macros, you can see in the *Layers View* all the *defines*, *undefs* and *ifdefs*.



Expand Macros – To See What a Macro is Replaced With

Select a piece of code that includes the macro, right click and choose *Macros > Expand All Levels Inline*. The macro call will be replaced with the code that results when the compiler applies preprocessing.

This capability can help understand and debug the behavior of a macro. You can run the simulation with the macro expanded. After is done, all you have to do is click and collapse the expansion to the initial macro call. DVT automatically adds *TODO reminders* so you don't forget expanded macros in the code.

```

103
104 -- This is the sequence driver for an ACTIVE SLAVE agent.
105 // @DVT_EXPAND_MACRO_INLINE_START
106 driver: vr_xbus_slave_driver u is instance;
107 Macro expansion. [Right click to collapse] == read_only(bus_name);
108 keep driver.slave_name == read_only(agent_name);
109

```

Problems Tasks Layers Coverage Checks Macros Console

1 items

| | Description | Resource | Path | Location | Type |
|--|---------------------|----------------|------------|----------|----------------|
| | Macro expansion. [R | vr_xbus_agent. | /vr_xbus/e | line 105 | Collapse Macro |

Show Macro Stack – To Inspect Debug Errors in Nested Macros

Errors are signaled at the line where the macro is used. But sometimes the error is deeply nested in a macro that uses another macro that uses another macro and so on. With a right click on the error marker you can see the macro stack to the source of the error.

```

80 |
81 some example;
82
83 '>
84

```

Problems Tasks Layers Coverage Checks Macros Console Search

[xbus_e] trace macro error

```

*** Error: Unrecognized action 'out(...)' with {...}'.
[see UNRECOGNIZED ACTION spec for more details]
detected at line 81 in '/home/cristian/hvl_sources/DEMOS/ovm_ml_2.0.1/ovm_examples/xbus_e/e/xbus_top.e'
inside macro <s'statement> at line 35 in '/home/cristian/hvl_sources/DEMOS/ovm_ml_2.0.1/ovm_examples/xbus_e/e/xbus_bfm.e'
inside macro <ss'struct_member> at line 44 in '/home/cristian/hvl_sources/DEMOS/ovm_ml_2.0.1/ovm_examples/xbus_e/e/simple_ram_env.e'
inside macro <a'action> at line 35 in '/home/cristian/hvl_sources/DEMOS/ovm_ml_2.0.1/ovm_examples/xbus_e/e/xbus_types_h.e'

out(<name>) with {};

```

Hyperlinks – To Jump to a Macro Definition

If you place the cursor over a line where you use a macro and press F3 (or with Ctrl+Mouse Over and click the hyperlink), you'll jump to the macro definition. This may be handy in case you want to understand what's actually going to happen if you use this macro.

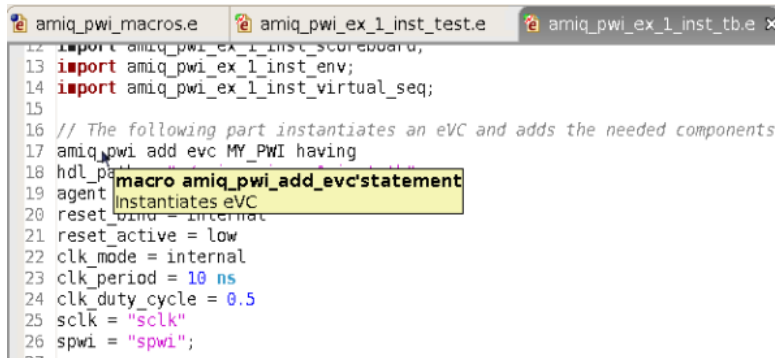
```

26 >pw1 = <pw1 ,
27
28 // Add the agents corresponding to DUT
29 amiq_pwi add agent ACTIVE MASTER MY_MASTER to MY_PWI having
30 auth = {0xF; 0x0; 0x0; 0x0};
31
32 amiq_pwi add agent ACTIVE SLAVE MY_SLAVE_ONE to MY_PWI having
33 auth = {0xC; 0xA; 0xF; 0xE}
34 address = 0x1;
35
36 // And another logical slave to answer Master's requests
37 amiq_pwi add agent ACTIVE SLAVE MY_SLAVE_TWO to MY_PWI having
38 auth = {0xD; 0xF; 0x0; 0x0}
39 address = 0x2; macro amiq_pwi_add_agent'statement
Instantiates agent.
40
41
42 // Several other configurations
43 extend amiq_pwi_ex_1_inst_env_u {
44
45 connect_ports() is also {
46 do_bind(reset, sys.my_pwi.smp.reset);
47 };
48
49 connect_pointers() is also {
50 driver.my_master = sys.my_pwi.my_master.driver;
51 driver.my_slave_two = sys.my_pwi.my_slave_two.driver;
52 driver.my_slave_one = sys.my_pwi.my_slave_one.driver;
53 driver.env = me;

```

Tooltips – To See the Macro Description and Value

If you place the mouse over a macro call, a tooltip pops-up showing the description of the macro. The description is extracted from the comments written on top of the macro definition.



```
12 import amiq_pwi_ex_1_inst_scoreboard;
13 import amiq_pwi_ex_1_inst_env;
14 import amiq_pwi_ex_1_inst_virtual_seq;
15
16 // The following part instantiates an eVC and adds the needed components
17 amiq_pwi add evc MY_PWI having
18 hdl_part
19 agent
20 reset_bind = internal
21 reset_active = low
22 clk_mode = internal
23 clk_period = 10 ns
24 clk_duty_cycle = 0.5
25 sclk = "sclk"
26 spwi = "spwi";
```

macro amiq_pwi_add_evc'statement
Instantiates eVC

For pure preprocessing directives, you also see the macro value. No need to *grep* for the define line to check the value. Don't like tooltips? Click on it and jump to the source of the *define* to see and eventually change the value.

```
!master_reg_read_address : uint(bits:AMIQ_PWI_REG_ADDR_LEN);
!master_cva : uint(bits:AMIQ_PWI_CVA_VOLTAGE_LEN);
!slave_item_kind : amiq_pwi_iter
!slave_frame_kind : amiq_pwi_frm
!slave_frame_parity_bit : bit
```

define AMIQ_PWI_CVA_VOLTAGE_LEN
Defined Values:
7

Ifdef Highlight – To See the ifdef Active Code

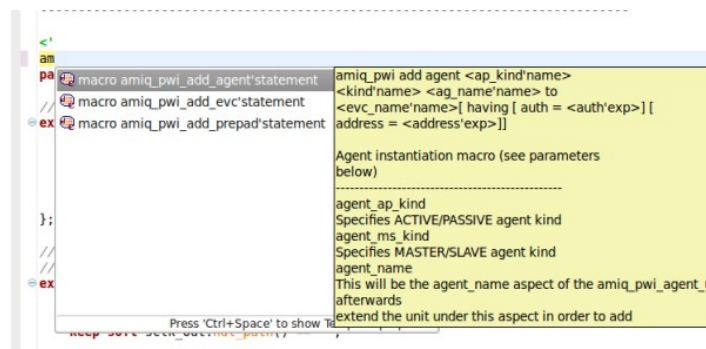
The editor marks the areas of code that are not compiled due to preprocessing *ifdefs* with gray.

```
#ifdef SPECMAN_FULL_VERSION_08_20_001 {
    import ovm_lib/e/ovm_e_top.e;
};

#ifdef SPECMAN_FULL_VERSION_08_20_001 {
    import ovm_e/e/ovm_e_top.e;
};
```

Autocomplete – To Use a Macro Like Any Other Construct

By typing the first letters of a macro, you can invoke the *autocomplete* feature, which allows you to choose from the proposed list of macros the one you want to use. The description is also there to help you choose.



```
<'
am
pa macro amiq_pwi_add_agent'statement
// macro amiq_pwi_add_evc'statement
ex macro amiq_pwi_add_prepad'statement
};
//
ex
```

amiq_pwi add agent <ap_kind>name>
<kind>name> <ag_name>name> to
<evc_name>name>[having [auth = <auth'exp>][
address = <address'exp>]]

Agent instantiation macro (see parameters below)

agent_ap_kind
Specifies ACTIVE/PASSIVE agent kind
agent_ms_kind
Specifies MASTER/SLAVE agent kind
agent_name
This will be the agent_name aspect of the amiq_pwi_agent_u
afterwards
extend the unit under this aspect in order to add

Press 'Ctrl+Space' to show

In conclusion, the *e* language macros are very powerful and useful when used properly. If you use them to enhance the language, you also have to ensure support for the new constructs. DVT helps you quickly inspect macros and understand their behavior and usage, as well as develop quality *e* code much faster.

For further information, visit the specific chapter on macros from the *DVT e Language Guide* at: <http://www.dvteclipse.com/documentation/e/Macros.html>