Simplifying UVM in SystemC

CONCLUSIONS

Packet kind;

This work therefore presents an add-on library for UVM-SystemC to facilitate the easier creation of a UVM verification environment, with the goal to reduce the amount of code to be written and thereby making creation less error prone and easier.

This paper described a C++ library on top of UVM-SystemC. In particular, the examples showed the following features:

- Use of template based member variables to simplify randomization
- Extendable enumeration type (as a base for aspects later on)
- New template types
- Use of template based member variables to simplify randomization
- Variables of an object need to be registered for printing, packing, copying e.g. by the use of field macros
- New template types
- "uvm phys_var" and "uvm var" allow to register the member variables used in the current "uvm" object.
- This allows it to randomize member variables using randomization engines
- By deriving from a specialized base class it is also possible to provide standard implementations for methods such as "do_print" which the user has to implement.

The class packet is extended for the case that the randomization selects for the enum "data_elem" the value SHORT.

Method "my method" is changed in a way that the method "make to short" is executed before the original method.

Example

```cpp
int sc_main(int argc, char* argv[]) {
    // instantiates packet with all extensions
    packet* my_packet=UVM_INSTANTIATE(packet);
    // randomizes variables corresponding to the constraints
    my_packet->randomize(); // randomizes variables corresponding to the randomized data_elem (the aspect)
    // calls methods corresponding to the randomized data_elem (the aspect)
    my_packet->my_method(23, "ADR");
}
```

A packet is instantiated, which gets randomized afterwards by calling the function "randomize()"

When calling the member function "my_method", depending on the aspect it is determined in which order the method calls are executed.

© Accellera Systems Initiative

INTRODUCTION

UVM-SystemC is currently under standardization within Accellera with a first preview release expected in 2015. Although, the UVM standard is getting more and more language-agnostic with implementations available in e, SystemVerilog, and now SystemC, features for transaction-based stimulus and verification environment modeling still strongly rely on the underlying language. For example, packing, copying, and randomization operations are implemented differently in each of these languages; certain features such as aspect-oriented extensions of classes and methods are currently only available in e.

This paper described a C++ library on top of UVM-SystemC. In particular, the examples showed the following features:

- Use of template based member variables to simplify randomization
- Extendable enumeration type (as a base for aspects later on)
- Aspect-specific extension of methods
- Aspect-specific adaptions of constraints
- New template types
- "uvm phys_var" and "uvm var" allow to register the member variables used in the current "uvm" object.
- This allows it to randomize member variables using randomization engines
- By deriving from a specialized base class it is also possible to provide standard implementations for methods such as "do_print" which the user has to implement.

This work therefore presents an add-on library for UVM-SystemC to facilitate the easier creation of a UVM verification environment, with the goal to reduce the amount of code to be written and thereby making creation less error prone and tedious.

The class packet is extended for the case that the randomization selects for the enum "data_elem" the value SHORT.

Method "my method" is changed in a way that the method "make to short" is executed before the original method.

Example

```cpp
int sc_main(int argc, char* argv[]) {
    // instantiates packet with all extensions
    packet* my_packet=UVM_INSTANTIATE(packet);
    // randomizes variables corresponding to the constraints
    my_packet->randomize(); // randomizes variables corresponding to the randomized data_elem (the aspect)
    // calls methods corresponding to the randomized data_elem (the aspect)
    my_packet->my_method(23, "ADR");
}
```

A packet is instantiated, which gets randomized afterwards by calling the function "randomize()"

When calling the member function "my_method", depending on the aspect it is determined in which order the method calls are executed.