Simplifying UVM in SystemC

Thilo Vörtler[1], Thomas Klotz[2], Karsten Einwich[3], Felix Assmann[2]

[1] Fraunhofer IIS, Design Automation Division, Dresden, Germany
Thilo.Voertler@eas.iis.fraunhofer.de

[2] Bosch Sensortec GmbH, Dresden, Germany
{Thomas.Klotz|Felix.Assmann@bosch-sensortec.com}

[3] COSEDA Technologies GmbH , Dresden, Germany
karsten.einwich@coseda-tech.com

***Abstract*— 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 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.

I. Introduction

The Universal Verification Methodology (UVM) [1] has become the dominant verification method-ology for the verification of digital and mixed-signal circuits and systems in recent years. With making the UVM library available in SystemC [2], [3], the promise of a real universal verification methodology, which is a principle of designing testbench architectures instead of a class library, is being realized.

UVM is based on the coverage-driven verification principle with the testbench sending randomized data to the design under test. The e functional verification language [4] has been the originator of this principle, with powerful language features allowing to describe complex constrained random stimulus in a compact way. Furthermore, the aspect-oriented programming features allow to easily extend existing parts of the testbench, such as drivers and monitors, with additional features by creating a new aspect of a class.

For randomization, SystemC does not offer built-in constructs, similar to those in e. Therefore, additional randomization libraries such as CRAVE [6] or SCV [7] have to be considered. To enable easy replacement of testbench components (realized by extensions and aspects in e), the UVM implementation for SystemC uses a factory pattern similar to UVM in SystemVerilog. The latter also pro-vides field macros, which can be utilized to compactly describe constrained random stimulus. How-ever, UVM-SystemC does not support field macros, as their use is not recommended [7], [8].

Especially when creating UVM transactions (sequence items) and sequences in UVM-SystemC (as in SystemVerilog), the engineer has to write additional code for managing the test environment. This makes the code longer and harder to understand. To reduce this overhead we propose a template based add-on library on top of UVM-SystemC, which reduces the amount of code to be writ-ten by the engineer. This way, UVM-SystemC is still compliant to the UVM (and its reference implementation), but the expressiveness of C++ and the newly available features of C++11 are put into place to enable the designer with a more powerful and compact description of UVM in SystemC. Moreover, because of this, the default compiler for SystemC can be used without any adaptions.

II. Library Features

This section presents some of the features of the implemented library in terms of short code snippets. These snippets illustrate how a certain UVM feature can be implemented in UVM-SystemC and how this implementation looks like when using the proposed library.

The code snippets are inspired by the example UVC freely available in [9]. The UVC describes a simple environment providing agents to send and receive data items called “packet”. Such a packet has certain attributes such as an address, a payload, a parity, etc.

A. *Template types for variable registration and randomization*

Figure 1 shows an example header of a UVM packet written for UVM in SystemC without the library extensions. Figure 2 demonstrates our proposed extensions for describing library variables. It can be seen how the functions for printing, packing, copying and comparing do not need to be implemented explicitly by the engineer anymore.

```
enum class packet_length_t {
       SHORT, LONG, MEDIUM, RANDOM
};
enum class packet_kind_t {
       GOOD, BAD_PARITY
};

class packet: public uvm::uvm_sequence_item {
public:
       UVM_OBJECT_UTILS (packet);
       packet(const std::string& name = "packet");

       packet_kind_t packet_kind;
       sc_dt::sc_uint<2> address; // Physical field
       sc_dt::sc_uint<6> length; // Physical field
       packet_length_t packet_length;
       std::vector<sc_dt::sc_uint<8> > payload; // Physical field
       sc_dt::sc_uint<8> parity; // Physical field
       sc_dt::sc_uint<8> packet_delay;

       virtual void
       do_print(uvm::uvm_printer& printer) const override;

       virtual void
       do_pack(uvm::uvm_packer& packer) const override;

       virtual void
       do_unpack(uvm::uvm_packer& packer) override;

       virtual void
       do_copy(const uvm::uvm_object& rhs) override;

       virtual bool do_compare(const uvm::uvm_object& rhs,
                      const uvm::uvm_comparer* comparer = NULL) const override;

       virtual std::string convert2string() const override;

};
```

Figure 1: UVM SystemC packet example without extensions

```
class packet: public uvm::uvm_s_sequence_item {
public:
        uvm_enum packet_kind;
        uvm_phys_var<sc_dt::sc_uint<2> > address;
        uvm_phys_var<sc_dt::sc_uint<6> > length;
        uvm_enmum packet_length;
        uvm_phys_var<uvm_vector> payload;
        uvm_phys_var<sc_dt::sc_uint<8> > parity;
        uvm_var<sc_dt::sc_uint<8> > packet_delay;
        packet(const std::string& name = "packet");
};
```

Figure 2: UVM SystemC packet example with extensions

The introduced 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, similar to the API provided in [3]. Furthermore, based on the type of the variables template specializations for common data types can be provided. In addition, 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.

B. *Adding aspect-oriented features to UVM-SystemC*

One of the main features of using the e language is the aspect-oriented extension of classes. Aspects cover features or concerns that cut across the system or parts of it, i.e. they do not only effect only one class but multiple ones. Hence, aspect orientation can be seen as a second layer of encapsulation complementing encapsulation in classes. An aspect can be used to extend existing classes by either introducing new or overwriting existing attributes, methods, etc. UVM for SystemC uses a factory pattern which allows to exchange objects based on their types with objects from derived sub classes, similar as in SystemVerilog.

The example in Figure 1 uses “traditional” enumeration types as known from the C language. However, in order to span up additional aspects later on in the implementation, an extendable type of enumerations is needed. With this, further elements may be added to an existing enumeration at different positions in the implementation. Figure 3 shows how this feature can be utilized based on the proposed library.

```
namespace uvm_aspect_enums {
static const uvm_enum_elem SHORT(„SHORT“);
static const uvm_enum_elem LONG(„LONG“);

class uvm_enum {
    bool extend (std::initializer_list<uvm_enum_elem> );
    std::vector<uvm_enum_elem> elements;
    …
};
…
}

static const uvm_aspect_enums::uvm_enum data_elem;

UVM_ASPECT_CLASS(packet)
{

        UVM_ASPECT_CTOR(packet)
        {
                        data_elem.extend({SHORT, LONG});
        }
};
```

Figure 3: Example of extendable enumerations

By adding enumerations to a class, aspects are created which can extend this class. This can be utilized, for instance, to add constraints depending on a given aspect. In the case the enumeration is extended e.g. while using an existing verification IP for a new product with additional modes, the range for randomization is extended to this new element without changing the original verification IP.

```
UVM_ASPECT_CLASS(packet)
{
        uvm_var<int> adr;

        UVM_ASPECT_CTOR(packet)
         {
                data_elem.extend({SHORT, LONG});
                //constraints
                keep_soft("adr_max",adr<65000);
                keep("adr_pos",adr>=0);
        }
};
// add constraints for SHORT packet only
UVM_EXTEND_CLASS(packet, SHORT)
{
        UVM_EXTEND_CTOR(packet,SHORT) : uvm_aspect(data_elem==SHORT)
        {
                //add constraint
                keep(“adr_short_constraint”,adr<32000);
        }
};
```

Figure 4: Adding constraints based on aspects

Even more powerful, not just constraints but also method implementations can be changed depending on a given aspect. Figure 5 shows how a method implementation is extended in a specific aspect.

```
UVM_ASPECT_CLASS(packet)
{
        uvm_method<bool(int&,std::string)> my_method;

        bool check(int& val, std::string kind);

        UVM_ASPECT_CTOR(packet)
         {
                data_elem.extend({SHORT, LONG});
                //point to actual method
                  my_method.is(check);
        }
};
// add method extension for SHORT packet only
UVM_EXTEND_CLASS(packet, SHORT)
{
         bool make_to_short(int& val, std::string kind);

        UVM_EXTEND_CTOR(packet,SHORT) : uvm_aspect(data_elem==SHORT)
        {
                  // for SHORT packet execute the make_to_short method first
                my_method.is_first(make_to_short);
        }
};
```

Figure 5: Adding method calls based on aspects

The class packet is extended for the case that the randomization selects for the enum “data_elem” the value SHORT. In this case the method “my_method” is changed in a way that the method “make_to_short” is executed before the original method. Alternatively the original method can be replaced or an additional method can be executed after the original method. In Figure 6 a simple example is shown. First a packet is instantiated, which gets randomized afterwards. When calling the member function my_method, depending on the aspect it is determined in which order the method calls are executed.

```
int sc_main(int argn,char* argc[])
{
        //instantiates packet with all extensions
        packet* my_packet=UVM_INSTANTIATE(packet);

        my_packet->randomize(); //randomizes variables coresponding to the constraints

        //calls methods corresponding to the randomized data_elem (the aspect)
        my_packet->my_method(23,"ADR");
}
```

Figure 6: Example of using an aspect class

III. Conclusion

This paper described a C++ library on top of UVM-SystemC which eases the description of verification environments and sequences by introducing extensions. 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

This paper describes an ongoing work, the library implementation is subject to change. The authors will continue to work on this library and step-by-step introduce further features. The proposed library is planned to be donated to the Accellera SystemC Verification Working Group and to be made available as open source.

IV. References

[1] Accellera Systems Initiative, Standard Universal Verification Methodology (UVM), http://www.accellera.org/downloads/standards/uvm/

[2] IEEE Computer Society, 1666-2005 IEEE Standard SystemC Language Reference Manual

[3] T. Vörtler, T. Klotz, K. Einwich, Y. Li, Z. Wang, M.-M. Louerat, J.-P. Chaput, F. Pecheux, R. Iskander, M. Barnasconi. Enriching UVM in SystemC with AMS extensions for randomization and coverage. Proceedings of the Design and Verification Conference Europe (DVCon Europe), 2014.

[4] IEEE Computer Society, 1647-2011 IEEE Standard for the Functional Verification Language e

[5] Accellera Systems Initiative, SystemC Verification Library, http://www.accellera.org/downloads/standards/systemc

[6] F. Haedicke, H. M. Le, D. Große, R. Drechsler. CRAVE: An Advanced Constrained Random Verification Environment for SystemC. International Symposium on System on Chip (SoC), 2012.

[7] Doulos, Easier UVM Coding Guidelines, https://www.doulos.com/knowhow/sysverilog/uvm/easier_uvm_guidelines/

[8] Erickson, Adam. Are OVM & UVM macros evil? A cost-benefit analysis. Proceedings of the Design and Verification Conference (DVCON), 2011.

[9] Cadence: Udacity course on Functional Hardware Verification, https://www.udacity.com/course/functional-hardware-verification--cs348