Easier UVM - Coding Guidelines and Code Generation

John Aynsley
Doulos
Church Hatch, 22 Market Place
Ringwood, United Kingdom
+44 1425 471223
john.aynsley@doulos.com

Dr. Christoph Sühnel
Doulos
Freikorpsstrasse 2c
D-82487 Oberammergau
Germany
christoph.suehnel@doulos.com

ABSTRACT

This paper describes our experiences with the Easier UVM coding guidelines and code generator with the objective of encouraging the UVM community to think about and eventually converge upon some set of coding guidelines for UVM. Easier UVM consists of a comprehensive set of coding guidelines for the use of UVM and an open-source UVM code generation tool that automatically generates the boilerplate UVM code for a project according to these guidelines. Easier UVM helps individuals and teams get started with UVM, helps avoid pitfalls, helps promote best practice, and helps ensure consistency and uniformity across projects. Easier UVM helps teams to become productive with UVM more quickly, and reduces the burden of maintaining a UVM codebase over time. Both the guidelines and the tool can be taken as they are or can be used as a starting point and modified according to the demands of a specific project.

Keywords

SystemVerilog, UVM, functional verification, constrained random verification, programming language, code generator

1. MOTIVATION

Over the past two years SystemVerilog has become the language of choice for new adopters of constrained random, coverage-driven verification, effectively displacing the earlier proprietary languages (e and Vera) in many situations. Although e still retains a customer base, most new adopters are choosing SystemVerilog for one reason alone: SystemVerilog is a standard. SystemVerilog is actively supported by all of main simulator vendors, both in the technical sense that the simulators have robust support for most of the language features, and in the commercial sense that the vendors actively promote the adoption and use of SystemVerilog. There is now an ecosystem of tool vendors, IP providers, consultants, and training providers supporting SystemVerilog.

But SystemVerilog is not without its problems. It is an extremely large and complex language, and the road to the current level of tool support has not been an easy one. The sheer scale and complexity of the task of implementing a SystemVerilog simulator has forced tool vendors to prioritize their implementation efforts, giving most attention to the particular language features being demanded by their customers or perceived as necessary to meet their own product positioning and marketing goals. Even today, there are still differences in interpretation and implementation across the simulators.

One very powerful business driver for the convergence of the main SystemVerilog implementations has been the desire to win customers from the competition by offering excellent compatibility with whatever SystemVerilog methodology libraries the competing vendor happens to champion. A few years back, this was AVM for Mentor, URM for Cadence, and VMM for Synopsys. Commercial pressures eventually lead to the convergence of these three methodologies in a single methodology, UVM, but along the way each tool vendor had to make sure that they fully supported the SystemVerilog language constructs used by the competing methodologies, and more than that, had to ensure that they interpreted the relevant areas of the standard in a mutually consistent way. This process became a *virtuous circle*, whereby the presence of an emerging standard SystemVerilog class library put pressure on the simulator vendors to implement the standard fully and accurately, and the improved tool support across all the full range of vendors lead to an increased confidence to adopt both the SystemVerilog language and the UVM library.

Doulos has directly experienced a significant jump in the volume of SystemVerilog training delivered with the introduction of each standard methodology (AVM, VMM, URM, OVM, UVM), the biggest jump appearing alongside with the introduction of UVM.

So while UVM has been a catalyst for the adoption of SystemVerilog, the adoption of UVM itself has not been without its challenges. While UVM is arguably an improvement over its ancestor methodologies, it is itself complex and challenging to learn and to use. A quick survey of UVM training classes offered by the tool vendors themselves and by independent training providers shows the average class length to be 4 days in a range of 3 to 5 days, and in every case these classes assume a working knowledge of the SystemVerilog language as a starting point. In the case of Doulos, the typical formal training program for engineers already familiar with Verilog would consist of a 4 day training class to teach the verification features of the SystemVerilog language, followed by 4 days to teach UVM. Experience has shown that this level of training is the minimum required for effective adoption: although managers with limited budgets of time and money will often try to reduce the extent of formal training, this is usually a false economy in that it greatly increases the on-the-job learning time and leads to bugs and false starts. All-in-all, the adoption of UVM requires a significant up-skilling which is frequently underestimated.

Even assuming the highest quality training, there is still a need for further help to get started with the first project. Training is necessary, but it is not sufficient. User companies quote timescales from 3 months to 12 months before their engineers are fully up-to-speed with UVM. In order to assist in the adoption process, tool vendors offer UVM-aware graphical capture tools, text editors, simulation environments, and debug environments. Specialist verification consultants offer coaching and mentoring. Some vendors offer packages combining tool purchase with training or mentoring schemes. User companies themselves develop their own in-house rules and coding guidelines.

The UVM base class library itself presents its own challenges. It consists of around 300 separate SystemVerilog classes, and the documentation included with the UVM release, consisting of a Class

Reference and a User Guide, is incomplete in some areas and leaves many questions unanswered. This is exacerbated by the fact that UVM still maintains a form of backward compatibility with its ancestor methodologies, so the UVM codebase still includes code from AVM and URM that is not necessarily maintained to the same quality as the more widely used features. UVM offers “more than one way to do it”, and there are areas in which the experts still debate which is the best approach. There are sometimes alternative approaches offered by the various ancestor methodologies between which the user must choose. There are optional shortcuts (such as text macros) which the experts still debate. There are new features with an experimental flavor that are championed by some expert users while being frowned upon by other expert users. To add fuel to the fire, the web has created a forum in which experts, both real and self-proclaimed, can propagate their advice to the wider community. This advice is sometimes excellent, sometimes contradictory, sometimes misguided, and sometimes just plain wrong. Having a sandpit in which people can play with new ideas can be a very positive thing, but at the same time it can be hard and time-consuming to sort the wheat from the chaff in the absence of a definitive methodology. Several pundits have observed that UVM is still in need of a “methodology”, in the sense of a definitive set of rules and guidelines directing its use.

2. INTRODUCING EASIER UVM

It is three years since Doulos first presented a paper entitled Easier UVM at DVCon. The goal of that first *Easier UVM* paper was to identify a minimal set of concepts sufficient for constrained random coverage-driven verification in order to ease the learning experience for engineers coming from a hardware design background. The first paper was explicitly aimed at mainstream Verilog and VHDL users, not verification experts. A lot of UVM marketing material and workshops were then (and still are) aimed at early adopters and verification experts, and as such have their place, but it is Doulos' direct experience that many new UVM users do not consider themselves experts and need some help getting started. The goal of Easier UVM "version 1" was primarily educational and pedagogical, that is, to reduce UVM to a set of simple concepts and coding idioms that are relatively easy to learn. Practically speaking, Easier UVM was a subset of UVM, but was not meant to exclude any features of UVM, just give a good starting point for learning: other features of UVM could always be introduced as the user became more confident. It was always recognized that, despite the title, learning UVM is still not easy.

Since that time there has been a very rapid adoption of UVM across the industry, and the experience gained from training and consulting with many users over that period gives us the confidence to propose a more prescriptive set of UVM coding guidelines complemented by a UVM code generation tool.

This second paper on Easier UVM introduces a set of specific coding guidelines that suggest “one way to do it”, helping to give new users clear direction regarding best practice, and an automatic code generation tool that can generate the first tier of the UVM codebase for a new project, generating the basic UVM structures starting from a simple template. This offers a number of specific benefits, as follows.

- Easier UVM can help individuals and teams to get started with UVM, reinforce what they learned during training, learn best practice, and avoid the most common pitfalls.

- Easier UVM can help individuals and teams to become productive with UVM more quickly. In practical use on industrial projects, one of us (Suehnel) has found the use of the code generator to cut around 6 weeks from the coding effort at the start of the project

- Easier UVM helps teams to use UVM in a more consistent way across and between projects within a company, and thus to reduce the burden of supporting the UVM codebase over time.

- Easier UVM helps make the planning and execution of UVM projects a more predictable process and can help keep even the first project on schedule.

The net effects are to accelerate the timescales of the first project on which UVM is adopted and to reduce the costs of maintaining a UVM codebase over time.

There is also a human element to this which should not be underestimated. By helping to avoid some of the basic pitfalls of SystemVerilog and UVM and by getting a simple working test bench up-and-running within a few days, project teams are encouraged and motivated to persevere with UVM where otherwise they might give up in frustration. On real projects the provision of a code generator has been found to make a dramatic difference in overcoming the resistance to change.

The Easier UVM guidelines document itself is too long to be embedded directly in this paper, but the guidelines and code generator can be downloaded, used, and modified free-of-charge [1].

3. CODING GUIDELINES

This paper describes our experiences with a specific set of coding guidelines and an associated code generator, but the intent of this paper is not to advocate the adoption of any one specific guideline over any other. We have found that any individual guideline will have its advocates and its detractors: company-specific coding guidelines usually end up being unique because opinions as to best practice can be subjective and differ substantially according to experience. If this paper encourages the UVM community to think about and eventually converge on *any* set (or sets) of coding guidelines, it will have fulfilled its objective.

That said, the Easier UVM coding guidelines address the following areas:

- Lexical Guidelines and Naming Conventions
- General Guidelines
- General Code Structure
- Clocks, timing and synchronization
- Transactions
- Sequences
- Objections
- Components
- Connection to the DUT
- TLM Connections
- Configurations
- The Factory
- Tests
- Messaging
- Functional Coverage
- The Register Layer
- Agent Data Structure and Packaging

Most of the guidelines could be regarded as common sense: there is nothing revolutionary. However, the guidelines document is a lot more prescriptive than either the official UVM Class Reference or the UVM User Guide: this document gives very specific recommendations about which UVM features to use and exactly how to use them. In some cases, this meant recommending best practice as commonly agreed upon across the industry. In other cases it meant making a rather arbitrary choice to favor one way of doing things rather than another. In most cases it was felt more useful to provide clear direction to do things in a certain way rather than to present alternatives along with a rationale for choosing between them, although there are still a few cases where we felt obliged to present users with a choice. In any case, Easier UVM is not meant to exclude any part of the SystemVerilog or UVM standards: the Easier UVM guidelines are offered as a suggestion of best practice, and users are free to take them, leave them, or modify them for their own purposes.

3.1. Coding Patterns

The Easier UVM coding guidelines start by defining coding patterns for the most common user-defined UVM classes, including obvious things such as the order of declarations, specific naming conventions, which macros to use, and which methods to override. To give an idea, here are the outlines of the three main coding patterns for components, transactions, and sequences:

Components

```
class my_comp extends uvm_component;
  `uvm_component_utils(my_comp)

  // Transaction-level ports and exports

  // Virtual interfaces named vif or *_vif

  // Internal data members named m_*

  function new(string name, uvm_component parent);
    super.new(name, parent);
  endfunction

  function void build_phase(...);
    ...

  // Other standard phase methods
endclass
```

Transactions

```
class my_tx extends uvm_sequence_item;
  `uvm_object_utils(my_tx)

  // Data members named m_*

  function new (string name = "");
    super.new(name);
  endfunction

  function string convert2string;
    ...
  function void do_copy(uvm_object rhs);
    ...

  // Other overridden methods
endclass
```

Sequences

```
class my_seq extends uvm_sequence #(my_tx);
  `uvm_object_utils(my_seq)

  // Data members named m_* acting as control knobs

  function new(string name = "");
    super.new(name);
  endfunction

  task body;
    ...
endclass
```

The guidelines also prescribe coding patterns to deal with specific situations, for example how to use configuration objects, how to configure sequences through the configuration database, how to start regular sequences from virtual sequences, and how to prolong run-time phases until all components have finished. The set of coding patterns is not exhaustive, and some of the guidelines will inevitably be contentious, but it was felt that offering some concrete advice was better than offering no advice.

An example of a very specific but very arbitrary choice is that concerning naming conventions. We recommend specific prefixes and suffixes to be used when naming class members (m_), ports (_port), virtual interfaces (_vif) and so forth. There is nothing right or wrong about the particular conventions chosen, except insofar as they are in general consistent with the conventions used within the UVM base class library itself, but there is clear benefit to be gained from adopting *some* naming convention. A company could conceivably replace these conventions with their own while otherwise adhering to some or all of the Easier UVM guidelines.

As another example of an arbitrary choice, on the contentious issue of whether or not to allow the use field macros, the Easier UVM guidelines advise against the use of field macros in general and give specific guidelines, with examples, concerning how to override built-in methods such as do_copy, do_compare, do_print and so forth. The code generator is able to generate the code for these methods automatically, thus counteracting one of the potential disadvantages of choosing *not* to use field macros. The arguments against the use of field macros include the compile-time and run-time overhead introduced by the code generated by these macros and the difficulty many users have experienced trying to understand and debug the field macros. On the other hand, many users do use field macros successfully, and recognizing that there may be situations where the field macros do get used, the Easier UVM guidelines suggest a way to highlight whether or not field macros are being used, namely:

- When not using field macros (recommended), register the class with the factory using one of the macros `uvm_component_utils or `uvm_object_utils as the first line within the class.
- If using field macros (not recommended), register the class and the fields using one of the macros `uvm_component_utils_begin or `uvm_object_utils_begin immediately after the declaration of any member variables.

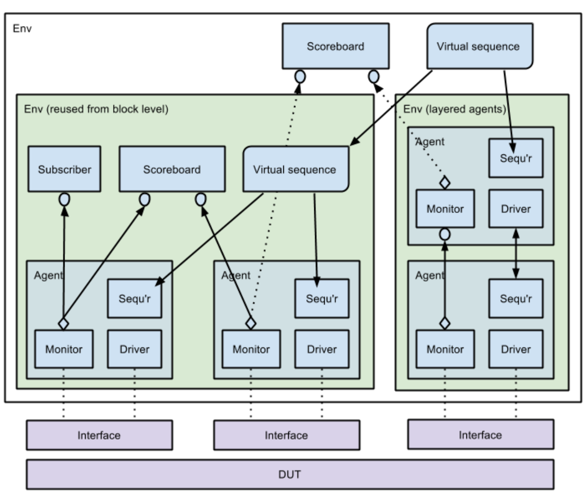
As another example of an arbitrary choice, we have chosen to recommend overriding the callbacks pre_start and post_start rather than pre_body and post_body when wanting to execute code before or after the execution of the body task of a sequence. There is some justification for this decision in that the pre/post_body methods are not called when a sequence is started using a macro from the `uvm_do family, whereas pre/post_start are called however the sequence is started.

An example of a coding guideline that encourages good coding style is to always call the randomize method of a sequence object before starting the sequence on a sequencer, whether or not the sequence class explicitly named at that point in the source code contains any rand variables. The justification for this guideline is that the type of the sequence object could be replaced at run-time using a factory override, and the extended class with which the sequence is replaced could contain rand variables, even if the base class did not. If the sequence object is not randomized before it is started, the rand variables within the extended class would not get randomized.

3.2. General Guidelines

As well as specific conventions, the coding guidelines also offer more general advice on the best way to structure the verification environment and how to handle commonly encountered problems. This goes beyond what might typically be found in company coding standards and overlaps with the good practice that might be learned during training. Hence the coding guidelines may help a project team to reinforce what they learned during formal training.

At the highest level, the Easier UVM guidelines show how to structure a UVM verification environment containing parallel agents controlled by virtual sequences and sending out transactions to a scoreboard for analysis, as illustrated in the figure below:



At a more detailed level, the general guidelines include common good practice such as:

- separation of tests from the verification environment
- developing verification components and tests with reuse in mind
- use of the factory and the configuration database
- use of transaction-level ports and exports
- use of virtual interfaces
- use of run-time phases
- use of virtual sequences and scoreboards in the presence of multiple parallel agents
- proper use of message ID and verbosity in user-defined reports
- use of the register layer
- advice on functional coverage
- advice on packaging data and structuring files for reuse

As an example, consider the use of the configuration database. The Easier UVM guidelines recommend that where a component has multiple configuration parameters, those parameters should be grouped together into a configuration object that gets stored in the configuration database and is associated with that component instance. Typically, each component instance would be associated with a unique configuration object, but it is also allowable that a component instance has no configuration object or, where appropriate, that several component instances share the same configuration object. A component may set configuration objects for its children, grandchildren and so forth, thus skipping generations down the hierarchy, but a component is only allowed to read its own configuration object, not the configuration object of its parent or grandparent. By this rule, the configuration of a component is only dependent on parameters contained within its own configuration object, and hidden dependencies on configuration information contained outside the immediate context of the component are forbidden.

The coding guidelines address the issue of how to return transactions from the driver in response to requests from the sequencer, where there are three common approaches:

- returning a separate response object from the driver to the sequencer by having the driver call the get method
- allowing the driver to modify one or more members of the request object itself rather than returning a separate response object
- not returning a response from the driver, but instead using transactions sent from the analysis port of the monitor component in the same agent

The coding guidelines explain the main approaches to layering sequencers and agents when building verification environments for layered protocols and when reusing agents in a layered fashion.

The coding guidelines contain examples which complement the application-specific code from the code generator. Guidelines and examples are important because the automatically generated code is only a starting point for writing application-specific verification environments. Having automatically generated code can nonetheless be important as part of the learning process, as well as an aid to productivity.

4. CODE GENERATION

The code generator itself is written in Perl and is freely available for download under an Apache 2.0 license. It generates SystemVerilog code that conforms to the Easier UVM guidelines, but because it is open source it can be modified if necessary to generate code according to the guidelines used on specific projects.

The code generator was originally based on the juvb11.pl contributed to the OVM-World by Mc Grath of Cadence [8]. Juvb11.pl was intended to generate an OVM framework for one single VIP only, i.e. no testbench was generated. The current script consists of about 3000 lines of Perl code and has been extended to generate a complete UVM verification environment including multiple UVM agents, the register model, and virtual sequences.

The code generator creates all the boilerplate code necessary to extend UVM classes such as drivers, monitors and agents, customized with application-specific information such as transaction fields and TLM port names. The application-specific information is drawn from a simple template fed into the Perl script. The code generator also creates placeholders where the user should insert their own application-specific code, and examples of top-level classes and modules that will be replaced by user-defined code.

The code generator creates the following set of files for any given DUT interface, where <name> is a prefix that designates the interface:

<name>.svh	List of `includes, one-per-class
<name>_agent.sv	UVM agent
<name>_common.sv	Placeholder for shared declarations
<name>_config.sv	Configuration class
<name>_coverage.sv	Subscriber with placeholder for covergroup
<name>_driver.sv	UVM driver with placeholder for pin wiggling
<name>_env.sv	UVM env that instantiates agent
<name>_if.sv	SystemVerilog interface
<name>_monitor.sv	UVM monitor with placeholder
<name>_pkg.sv	SystemVerilog package
<name>_seq_item.sv	UVM transaction with overridden methods
<name>_seq_lib.sv	Example sequences
<name>_sequencer.sv	UVM sequencer

Having a standard, uniform file structure and a uniform way to organize the various elements (agent, interface, package, sequence …) was found to help to maintain consistency across a team or company.

The user provides the code generator with a setup file that defines the contents of each DUT interface and each sequence item. From this, the code generator creates the following classes for each DUT interface:

- Sequence item (transaction)
- Sequencer
- Driver
- Monitor
- Agent
- Configuration (one per agent)
- Subscriber (for coverage collection)
- Sequence (simple sequence to run one transaction)
- Package (that includes the above classes)
- Interface (pin-level)

At the top level, the code generator also creates:

- Top-level module, which instantiates the interfaces
- Env, which instantiates
    - Agents
    - Configuration objects
    - Scoreboard (empty)
    - Register model (register layer)
- Test, which runs a virtual sequence to start one simple sequence per agent.

The top-level module and classes can be run out-of-the-box as an example that exercises the entire UVM verification environment down to the level of the drivers, which are initially just empty dummy implementations. Having a verification environment that could be simulated immediately was found to be of great value in overcoming peoples' skepticism toward adopting UVM for the first time on a project.

Having the boilerplate code generated automatically saves the effort of having to type the tedious things over and over again, provides a set of examples that are customized with user-defined interface, port and field names specific to the protocols being used, and ensure a level of consistency throughout the foundation on which the codebase is built.

This example shows boilerplate code from the code generator, including some user-defined class properties fed as input to the code generator:

```
`ifndef SPI_SEQ_ITEM_SV
`define SPI_SEQ_ITEM_SV

class spi_seq_item  extends uvm_sequence_item;

`uvm_object_utils(spi_seq_item)

// class properties
rand logic [127:0] data;
rand bit [6:0] no_bits;
rand bit RX_NEG;

extern function new(string name="spi_seq_item");
extern function void do_copy(uvm_object rhs);
extern function bit  do_compare(uvm_object rhs, uvm_comparer
comparer);
extern function string convert2string();
extern function void do_print(uvm_printer printer);
extern function void do_record(uvm_recorder recorder);

endclass : spi_seq_item
```

This example shows placeholders where user-defined code would need to be inserted:

```
task spi_driver::run_phase(uvm_phase phase);

 // add additional declarations here

 super.run_phase(phase);
 `uvm_info(get_type_name(),"run_phase",UVM_MEDIUM)

 // set signals on reset values here

 @(posedge vif.reset) // reset goes inactive
 forever begin
   seq_item_port.get_next_item(req);
   @(posedge vif.clk)
   `uvm_info(get_type_name(), {"req item\n",req.sprint},
UVM_MEDIUM)

   // insert the driver protocol here

   $cast(rsp, req.clone());
   // adopt the rsp
   seq_item_port.item_done();
 end
endtask : run_phase
```

The current version of the open-source code generator does not provide the ability to regenerate the code without disturbing any user-defined code inserted at the placeholders: any modifications would get lost if the code were regenerated. The intent is run the code generator just once and use the output from the code generator to create the initial framework for the user-defined code, but not to regenerate the code thereafter.

This is an example of the setup file that introduces the user-defined names to be inserted into the boilerplate code from the generator:

```
#
# uvc template
#
  # indented comment

###
### comment lines start with #
### comment lines and whitespace (blank lines) ignored
###
### "|" vertical bar is the field separator

#uvc_name| name of uvc (i.e. ahb_master)
uvc_Name| spi

#uvc_item| name of item (i.e mstr_pkt)
 uvc_item | spi_seq_item

#uvc_var | list of seq_item variables
uvc_var | rand logic [127:0] data;
uvc_var | rand bit [6:0] no_bits;
uvc_var | rand bit RX_NEG;

#uvc_if | name ofinterface (i.e. mstr_if)
uvc_if | spi_if

#list_of_ports:  (port list for interface)
uvc_port | logic clk;
uvc_port | logic reset;
uvc_port | logic [`SPI_SS_NB-1:0] ss_pad_o;
uvc_port | logic sclk_pad_o;
uvc_port | logic mosi_pad_o;
uvc_port | logic miso_pad_i;

#list_of_clocks:  (clock list for interface)
uvc_clock | clk

#list_of_reset:  (reset list for interface)
uvc_reset | reset
```

4.1. Practical experience with code generation

In discussing the issue of UVM code generation, the question often arises as to whether it is possible to use one-and-the-same UVM architecture across multiple projects, the counter argument being that every project is unique and demands a bespoke verification environment. In our experience, there are indeed certain common denominators across most projects that permit automatic code generation, and furthermore, having all UVM environments based on a uniform and flexible architecture enables verification effort to be focussed on the differences between projects, where it should matter the most. For example, all projects should use one agent per DUT interface, and the code for each agent should be organised in a uniform manner. The automatic code generator achieves this.

For a typical project, code generation might proceed as follows.

1. Hold a kick-off meeting to identify the top-level DUT and enumerate all of its functional interfaces. It is important to identify the functional interfaces up-front to reduce iterations.
2. Create setup files that name the pins and transaction variables for each DUT interface
3. Generate the code for the complete environment.
4. Simulate the complete environment (drivers, monitors, and scoreboards would still be dummies at this stage). This is important in helping beginners and their managers to learn where to start when working with an unfamiliar language and methodology. The automatically generated environment contains a virtual sequence that runs a transaction through each agent, and thus exercises the entire environment down to the level of the drivers.
5. Start implementing the drivers one-by-one.
6. Simulate each driver in turn by adding new sequences and tests to the environment, while many parts of the environment are still missing. This helps to demonstrate that progress is being made.
7. Implement monitors, subscribers, scoreboards, and add further sequences and tests.
8. As the tests increase in sophistication, add further data members to the sequence items and refined the methods as needed.

There is clear value in generating the verification environment for the first clean-sheet project, and automation helps to ensure completeness and consistency of verification environments across the company even when the users have become more experienced with UVM.

For derivative projects, if the functional interfaces of the DUT have not changed, it may be sufficient just to add further tests and sequences. On the other hand, if there are new or modified functional interfaces, our recommendation would be to regenerate the entire verification environment and to replace automatically generated dummy code with code created by hand from the previous project, where appropriate. Because the file structure is unchanged, it is straightforward to replace the newly generated code with the original code. Effort must be spent analyzing any new interface signals and making any necessary updates to the existing implementations. This approach ensures that all of the elements are created, connected, and configured correctly at the top level.

The goal is to generate each verification environment once and once only. In a few cases we found it necessary to make a second pass at code generation and to replace automatically generated parts with parts previously modified by hand, as described in the previous paragraph. This was caused by a failure to identify all the necessary functional interfaces up-front during the kick-off meeting, usually because of some oversight or a late change to the specification.

In practice, automatic code generation was not always straightforward. Each project may contain specific details that do not fit well with an automatically generated environment, or that require significant extensions to the environment. One example might be the handling of interrupts, where a single interrupt pin may need to affect the behavior of several functional interfaces. In that particular case, you have to decide whether to duplicate the interrupt pin across the interfaces, whether to create a separate interrupt interface manually, or whether to implement horizontal communication between the agents.

Although this paper has focussed on automatic code generation, a considerable part of any verification environment will need to be written by hand. For example, the current code generator does not create the internal structure of scoreboards, nor does it handle layered agents. Even using a code generator, people can still make mistakes, and use of a code generator will not necessarily guarantee compliance with specific project constraints.

5. CONCLUSION

We have witnessed a lot of enthusiasm from users regarding the availability of a set of specific coding guidelines concerning best practice for UVM code. It remains to be seen whether the Easier UVM coding guidelines and code generator will be widely adopted, but in any case we remain convinced of the necessity for some such set of guidelines over-and-above the UVM class library itself and the accompanying documentation.

6. REFERENCES

[1] The Easier UVM guidelines and code generator http://www.doulos.com/knowhow/sysverilog/uvm/

[2] IEEE Std 1800-2012 "IEEE Standard for System Verilog-Unified Hardware Design, Specification, and Verification Language"

[3] Universal Verification Methodology (UVM) 1.1d Class Reference, updated March 7, 2013

[4] Universal Verification Methodology (UVM) 1.1 User's Guide, May 18, 2011

[5] On-line resources from http://www.accellera.org/downloads/standards/uvm

[6] On-line resources from http://www.accellera.org/community/uvm/

[7] On-line resources from https://verificationacademy.com/

[8] Updated juvb template generator for UVM-1.1: http://forums.accellera.org/files/file/82-updated-juvb-template-generator-for-uvm-11/