How Far Can You Take UVM Code Generation and Why Would You Want To?

John Aynsley
Doulos, Ringwood, UK

***Abstract*-This paper describes the motivation for UVM code generation and the experiences of the author in implementing and deploying a particular code generator. It analyzes both the benefits and weaknesses of code generation as an approach to building UVM environments and describes how to go about creating and maintaining a code generator that is able to maximize the benefits while working around the weaknesses. The paper gives specific advice to anyone planning to implement and deploy a UVM code generator.**

I. Introduction

UVM, the Universal Verification Methodology for SystemVerilog, has become a very successful standard, yet adopting UVM can still be a significant challenge. UVM is large and complex. Moreover, the specification of UVM leaves plenty of room for choice when it comes to which particular set of features to use and which coding style to adopt. As a result, projects can find themselves with a code base of inconsistent and unmaintainable UVM code.

Code generation can be used as part of the solution. A code generator can encapsulate knowledge of good ways to structure a UVM verification environment, can help avoid coding pitfalls, and can increase productivity. A code generator can also be a good way to ensure that a growing code base is written in a consistent style and adheres to best practice. But code generators also have potential weaknesses, such as generating unreadable code and having insufficient flexibility to accommodate the kinds of coding structures that users want to write. This paper describes our experience with designing and deploying a particular code generator, the Easier UVM Code Generator, and investigates to what extent it is possible to construct and use a UVM code generator deep into a project by blending automatically generated boilerplate code with user-defined code fragments.

In this paper we describe the motivation for and the design philosophy of a particular UVM code generator. Meeting the primary design goal of a code generator, namely the automatic generation of a body of code that is syntactically and semantically correct and stylistically consistent, is straightforward. But there are secondary goals that present more of a challenge. A code generator and the code that it generates could be evaluated along the following dimensions:

- Good coding style and coding practices – encapsulating best practice and avoiding pitfalls
- Robustness and portability – code that runs on any simulator and is resilient to likely future changes
- Readability and maintainability – code that can be read and maintained by humans
- Flexibility and expressibility – incorporating user-defined code fragments within a user-defined structure
- Productivity – automating as much of the code production as possible
- Iteration – supporting an iterative development process where the generator is run repeatedly

A key issue is whether the user is allowed to modify the generated code. Any modification to the generated code is problematic, since any changes will potentially be overwritten the next time the generator is run.

II. Background to Easier UVM

The DVCon 2011 paper "Easier UVM for Functional Verification by Mainstream Users" [1] was an early presentation of what would eventually grow to become the Easier UVM Coding Guidelines, which consist of a set of 180 detailed coding guidelines with explanations and examples. Most of the guidelines could be regarded as common sense, but the guidelines document [2] is a lot more prescriptive than the official UVM documentation. The Easier UVM Coding Guidelines are not meant to exclude any part of the SystemVerilog or UVM standards: rather, they are offered as a suggestion of best practice, and users are free to take them, leave them, or modify them for their own purposes. A primary design goal of the Code Generator was to be consistent with the Coding Guidelines.

The DVCon 2014 poster "Easier UVM - Coding Guidelines and Code Generation" [3] described our motivation and early experiences with the Coding Guidelines (based closely on the previous paper) and with rolling out an early version of the Code Generator in an industrial context.

The Easier UVM Code Generator itself was originally based on the juvb11.pl script v1.09 by Jim McGrath, Cadence, which was uploaded as a UVMWorld contribution on 16 September 2011 [4]. This script was then modified by Christoph Sühnel, then at Mentor, before being modified extensively by David Long and John Aynsley at Doulos. The current version of the Easier UVM Code Generator [5] is available from Doulos under the Apache 2.0 license.

III. Why Code Generation?

Code generation offers some real benefits to both new and experienced users.

For new users:

- Code Generation can help individuals and teams ramp up with UVM by reinforcing what they have learned during training and providing complete, working code examples to build on. In one case, use of the Code Generator saved around 6 weeks at the start of a project [3].

For all users:

- Productivity. The use of a code generator avoids the tedious and error-prone work of writing and re-writing the boilerplate code of any UVM project, and avoids having to manually propagate changes though envs, sequences, configurations, and so forth.
- Consistency. Automatically generated code is always self-consistent, by design. This helps avoid both inconsistencies between the coding style and choice of features used by multiple engineers and also accidental inconsistencies within the coding style of a single engineer. Consistency has the effect of making the UVM code base easier to read and easier to maintain.
- Adhering to best practices and avoiding common pitfalls. This is related to consistency but goes deeper in that a code generator can capture known good coding idioms and lock them into a code base.

There are several ways in which a code generator could be put to use, and we are not claiming that all approaches will be suitable for every situation:

1. A code generator can be used simply as a learning aid by generating and running working examples of UVM code that are based on a particular DUT, which can be toy or real. The only requirements are to provide control files that represent the interfaces of the specific DUT, provide driver and monitor code to implement the protocols, run the code generator, and simulate. This approach can be used to demonstrate to yourself or to management that getting a UVM environment up-and-running is not so difficult!
2. A code generator can be used to create the initial framework of the production code, after which the code generator is abandoned and the production code maintained by hand. Although the code generator is only used in the initial stages of the project and is then discarded, there is still the benefit that the production code starts out with a consistent structure and coding style.
3. A code generator can be used to create the framework of the production code, and then used in combination with control files and include files to extend and modify the functionality of the generated code according to the specific needs of the project. If this approach is implemented carefully, it should be possible to re-run the code generator at any stage to reflect any changes made to the control files, but this does depend on not making any modifications to the generated code that cannot be reproduced at will using control files or include files, which might not be trivial to do. Using the code generator for as long as possible into a project makes it possible to continue to take of advantage of the code generator to create the code that connects everything together.

IV. Characteristics of UVM Code for Generation Purposes

Every UVM code base can be broken down into two parts: boilerplate code that is always the same and user-defined code that is always unique. This statement is an oversimplification, of course. The boilerplate code can be broken down into true boilerplate code that really is identical from case-to-case, and near-boilerplate code where there are strong similarities across instances, but also minor differences.

What this boils down to is the need for template-driven code generation that can accommodate rather arbitrary user-defined elements. In theory, the generated code could be copied from a set of templates that consist of legal, properly formatted SystemVerilog code using the UVM class library and containing a few special elements as follows:

- Variable fields that are replaced with user-defined names during generation. The variables fields are often used as part of a name in the generated code.
- Repeated line groups, where the loop might iterate over some quite arbitrary list of agents or settings and where the loop variable is available as a variable field for use within the line group
- Conditional line groups, where the existence of the line group depends on some user-defined setting
- Arbitrarily nested repeated and conditional line groups (like any structured programming language)
- Marking vertical alignment points within a line group for pretty-printing
- Marking the locations where user-defined code fragments could be included (either with `include or inlined)
- Marking the line groups where automatically generated methods could be suppressed
- A few additional ad hoc rules that require partially parsing user-defined code fragments
- A few variable fields assigned from arbitrary expressions over other variable fields

Here is an example of such a code template in a hypothetical Perl-like meta-format, where lines containing meta-commands are prefixed by --

```
`ifndef ${agent_name}_ENV_SV                                          VARIABLE FIELD
`define ${agent_name}_ENV_SV

--insert_inc_file $agent_env_inc_before_class{$agent_name};          MARK LOCATION FOR INCLUDE

class ${agent_name}_env extends uvm_env;                              VARIABLE FIELD

  `uvm_component_utils(${agent_name}_env)                             VARIABLE FIELD

  extern function new(string name, uvm_component parent);             BOILERPLATE

--for ( $i = 0 ; $i < $number_of_instances{$agent_name}; $i++ ) {     REPEATED LINE GROUP
--  $suffix = calc_suffix($i, $number_of_instances{$agent_name});     AD HOC VARIABLE ASSIGNMENT

  ${agent_name}_config    m_${agent_name}${suffix}_config;            VARIABLE FIELD
  ${agent_name}_agent     m_${agent_name}${suffix}_agent;
  ${agent_name}_coverage  m_${agent_name}${suffix}_coverage;

--  if ( @env_list) {                                                 CONDITIONAL LINE GROUP
  // Child environments
--  }
--}
--foreach $agent_env (@env_list) {                                    REPEATED LINE GROUP
--  if ( $agent_env ne "" )  {                                       CONDITIONAL LINE GROUP
--    align("  ${agent_env}  ", "m_${agent_env};", "");               VERTICAL ALIGNMENT POINT
--  }
--}
--gen_aligned();

--unless ( $agent_env_generate_methods_inside_class{$agent_name} eq "NO" ) {
--  unless ($comments_at_include_locations eq "NO" ) {                MARK OPTION TO SUPPRESS
    // You can remove build_phase and connect_phase by setting agent_env_generate_methods...
--  }
  extern function void build_phase(uvm_phase phase);                  BOILERPLATE
  extern function void connect_phase(uvm_phase phase);

--}
--insert_inc_file $agent_env_inc_inside_class{$agent_name};           MARK LOCATION FOR INCLUDE

endclass : ${agent_name}_env


function ${agent_name}_env::new(string name, uvm_component parent);  BOILERPLATE
  super.new(name, parent);
endfunction : new
```

What this shows is that UVM code generation, at the level at which we wish to use it, requires a degree of programmatic flexibility that cannot be accomplished merely by copying a set of exemplar source files, despite the existence of boilerplate code and the similarities between many UVM code fragments. The differences between UVM files are sufficient to demand an algorithmic approach to code generation. Hence the Easier UVM Code Generator is actually implemented as a Perl script.

The Code Generator creates the following set of classes, packages, modules, and interfaces.

Per-interface/agent:	**Top-level:**
uvm_sequence_item class	Top-level configuration object
Configuration class	Top-level env
Sequencer class	Reference model
Driver class	Instantiation of Syosil scoreboard
Monitor class	Instantiation of register block
Agent class	Default top-level virtual sequence class
Split transactors for emulation	Top-level package
Subscriber class	Test class
Default sequence class for agent	Test package
Env class	Test harness module to instantiate DUT
Default virtual sequence class for env	Top-level module
Default register sequence class for env	
Register model adaptor	
Agent package	
SystemVerilog interface	

V. Implementation of the Code Generator

In case you wish to use the Easier UVM Code Generator as a starting point for creating your own code generator, the overall flow of the script is as follows:

1. Parse any command line switches
2. Parse the common control files and set internal variables to represent settings
3. Check any mutual constraints on those settings
4. For each interface/agent
    a. Parse the control file and set internal variables to represent settings
    b. Check any mutual constraints on those settings
    c. Generate code based on the code templates and internal variables
5. Generate top-level code based on the code templates and internal variables
6. Generate simulator scripts

As described above, the Code Generator interleaves boilerplate code, variable fields, and user-defined code fragments. The control files provide a wide range of settings that allow a degree of flexibility in the generated code. Whether or not this flexibility is sufficient is discussed below.

Aside from errors introduced in the user-defined code fragments, the resulting generated code is complete and ready-to-run using the generated simulator scripts.

The virtues of this particular code generator are that it a) generates code that is complete and ready-to-run, b) is open-source and freely available, c) is well-documented [6], and d) is relatively well-tested, having its own regression test suite. However, the goal of this paper is not to persuade you of the merits of this particular code generator, but to give you a better understanding of UVM code generation in general.

VI. Idioms and Choices Captured by the Code Generator

Code templates, as described above, can capture many different aspects of coding style, ranging from low-level lexical conventions through to major decision on how the code is structured. This section contains a comprehensive list of these aspects, with the goal of giving you some sense of an answer to the question posed by the title of this paper, "How Far Can You Take UVM Code Generation".

At the lexical and file levels, the Easier UVM Code Generator captures the following conventions:

- File naming and file organization within a directory structure
- Inserts a standard file header at the top of each file generated
- Inserts conditional compilation guards around each compilation unit
- Organizes classes into packages using the `include directive to include the class code
- Indentation, spacing, alignment, and blank lines, otherwise known as pretty printing
- Naming conventions for variables and types, with standard prefixes and suffixes to names
- Matching the strings names of UVM objects with the corresponding variable names
- A conventional ordering for declarations and statements within each class

By keeping the code templates in something close to plain text form, it is quite straightforward to implement variations on any of the above. It helps to keep the meta-format as consistent as possible, e.g. having a simple and consistent way of vertically aligning text or constructing numerical suffixes.

At a structural level, the Easier UVM Code Generator captures the following:

- Separates the SystemVerilog module that instantiates the DUT from the module that instantiates the UVM test
- Supports top-level agents, agents within their own envs, and multiple instances of a given agent
- Uses configuration objects, one per-agent and one for the top-level env
- Connects a subscriber to the analysis port of each agent (versus nesting the subscriber inside the agent)
- What to do in the test versus the top-level env
    - Modifies configuration objects and sets factory overrides from the test
    - Instantiates the root register model and starts top-level virtual sequences from the env

The point here in this paper is not to argue the case for the specific choices made in Easier UVM, but rather to point out that it would be straightforward to modify the code generator for any similar structural choice, for example, extending all tests from a test base class.

At a more detailed level, the Easier UVM Code Generator captures a lot of specific choices concerning which UVM features to use and which coding idioms to use:

- The choice of which UVM component base classes to extend
- The choice of which UVM macros to use and which to avoid (Easier UVM avoids the field macros)
- Which actions to perform in which UVM phase methods (build_phase, connect_phase, and so on)
- Always using the factory method T::type_id::create to instantiate all components, sequences, and transactions
- The choice of a default set of variables to be included in configuration classes, including checks_enable and coverage_enable
- The choice of a specific mechanism for passing virtual interfaces from the top-level module through the configuration database to the agents
- The choice of where to assign the virtual interfaces located in the monitor and driver components
- The choice of a mechanism to set and get the is_active variable of the agent (by overriding get_is_active)
- Passing the configuration object for the agent to the associated sequencer and subscriber in the configuration database
- The choice of whether to introduce a user-defined sequencer class or use uvm_sequencer directly
- Always randomizing transaction objects and sequence objects between creation and sending
- Using set_starting_phase/get_starting_phase with sequences in case any nested or derived sequences happen to raise objections
- The choice of overriding pre/post_start in preference to pre/post_body or just body in sequences
- The choice of instantiating explicit predictors (set_auto_predict(0)) with the register model
- The choice of message ID and verbosity for reports according to the location and intention of the report
- Implicitly, the definition of a subset of UVM classes and methods which are sufficient for most purposes.

In addition to the above, which all represent conventional coding choices to some degree, the Easier UVM Code Generator also contributes to automation by generating code that would be time-consuming and error-prone to write by hand:

- Generation of a test harness module containing interface instantiations, the DUT instantiation, and pin-level connections between the interfaces and the DUT.
- Generation of the set of methods required by each transaction class, including support for metadata that is excluded from the automatically generated compare, pack, and unpack operations because it does not represent data transferred to and from the DUT (the motivation is the same as for field macros, but the code generation implementation is more transparent to the user)
- Generation of the register adapter for each agent and instantiation and connection of the register model, adapter, and predictor.

The Easier UVM Code Generator also contributes to productivity by generating working example code that can be used, extended or discarded subsequently:

- Default clock and reset generation circuitry in the test harness
- Default driver code to get the request transaction from the sequencer
- A default covergroup in the subscriber component containing a coverpoint for each transaction variable (excluding any metadata)
- A default write method in the subscriber component to sample the above covergroup
- Instantiation and configuration of the Syosil UVM Scoreboard [7], including any number of TLM connections to the DUT and to a reference model
- A default end_of_elaboration_phase method to dump the UVM component topology and the factory.
- A default report_phase method to report the final coverage for an agent
- A tree of default sequences consisting of a top-level virtual sequence, a virtual sequence for each env, and a sequence for each agent, each sequence having access to the associated configuration object.
- A default top-level virtual sequence that starts multiple synchronized volleys of low-level sequences across all agents.

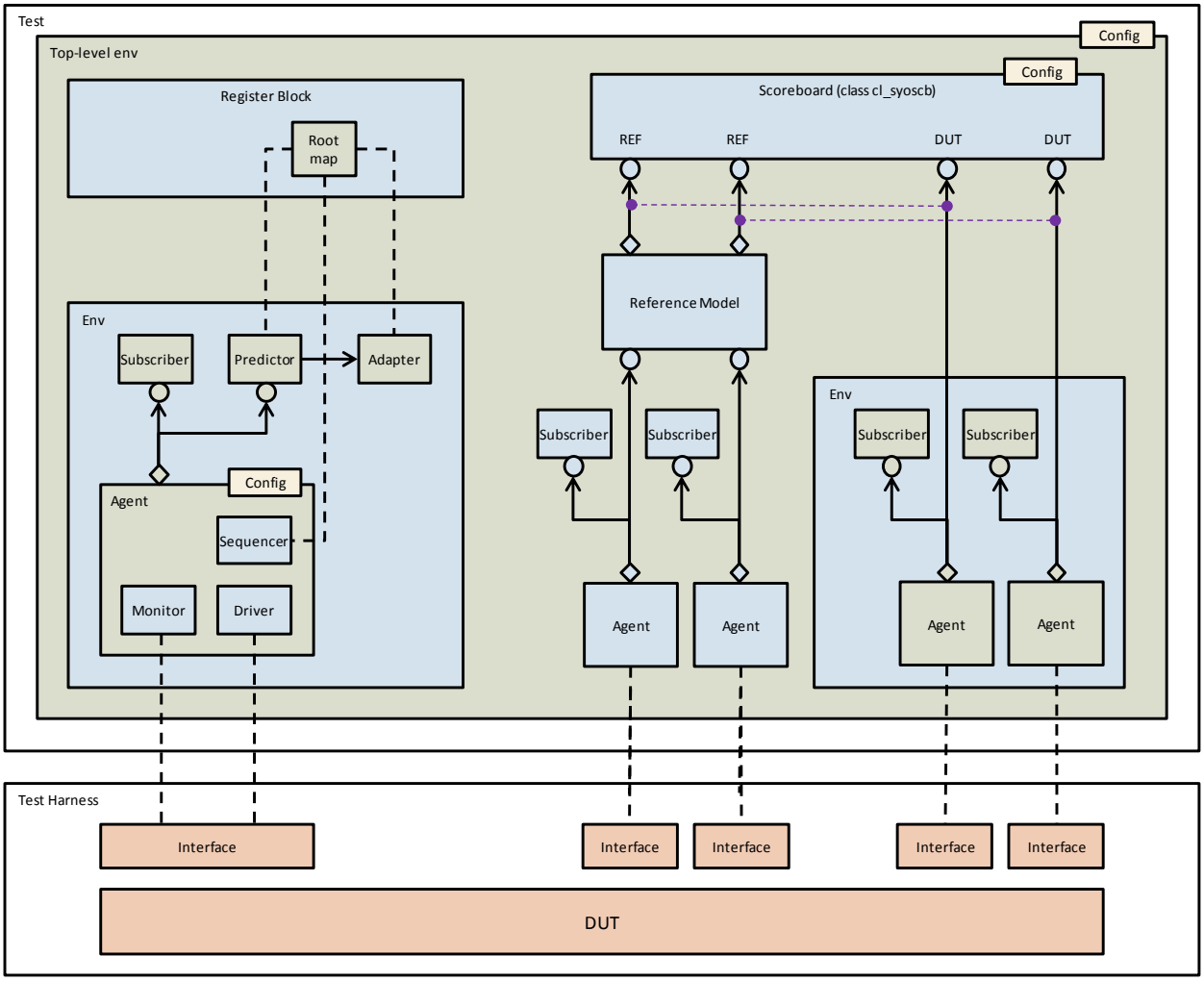

Figure 1. A Representative Structure from the Easier UVM Code Generator

Figure 1 shows an example of the kind of structure that the current Easier UVM Code Generator is capable of producing, including multiple DUT interfaces and associated agents, nested envs, and the instantiation of a register block, reference model, and scoreboard. Note that the Code Generator does not generate the contents of the register

block, reference model, or scoreboard. The register block would typically come from a separate generator, the reference model would be coded by hand, and the scoreboard is VIP available under an Apache 2.0 license.

VII. A UVM SUBSET?

Right from the beginning [1], one of the explicit goals of Easier UVM was to make life easier for the novice by deliberately restricting the number of UVM features being used, while at the same time allowing that any and all features of UVM could be used by experts, because UVM is the standard. Recently Sutherland and Fitzpatrick [8, 9] have published UVM-Light, which they call a subset of UVM for rapid adoption. In their paper, Sutherland and Fitzpatrick analyze the UVM constructs that appear in a simple, complete UVM example, and argue that this subset is sufficient to write "effective UVM testbenches". An analysis of the UVM constructs used by the Easier UVM Code Generator broadly supports this conclusion, with a few caveats.

Of the UVM constructs itemized in the second paper [9], all except three (phase_ready_to_end, response_handler, use_response_handler) are used by the Easier UVM Code Generator or closely associated user code. The Code Generator also uses a handful of other constructs that are not enumerated in the UVM-Light paper but were probably inadvertent omissions, namely uvm_macros.svh, uvm_phase, uvm_comparer, and a few verbosity levels such as UVM_LOW, UVM_MEDIUM, UVM_HIGH, bringing the feature count from the 65 claimed by Sutherland and Fitzpatrick up to around 70. The Easier UVM Code Generator then uses a further 20 features that are not mentioned as being part of UVM-Light, namely uvm_object, uvm_top, find, end_of_elaboration_phase, get_is_active, uvm_bitstream_t, uvm_factory::get, factory.print, find_override_by_type, do_print, do_record, uvm_recorder, `uvm_record_field, compare_field, pre_start, post_start, set_starting_phase, get_starting_phase, get_type_name, print_topology.

Broadly speaking, the message of Easier UVM is closely aligned with that of UVM-Light, namely, that a very small subset of UVM is sufficient not only to get started with UVM but to do a substantial amount of work. However, Easier UVM acknowledges that some users will need a richer API. In particular, some uses will need to take advantage of further methods that reveal or control the state of UVM objects, methods such as factory::find_override_by_type or uvm_sequence::get_sequencer or uvm_sequencer::user_priority_arbitration. Unfortunately, the list goes on and on, and given the basic principles of API design (such as completeness, consistency and orthogonality), we end up with something that contains the majority of the UVM standard, though not necessarily all of it.

VIII. HOW FAR CAN YOU TAKE UVM CODE GENERATION AND WHY WOULD YOU WANT TO?

In this section we review our experience of using the Easier UVM Code Generator and reflect on the two questions posed in the title of this paper, how far can you take UVM code generation and why would you want to?

In direct answer to the first question, by employing an algorithmic approach there is no hard upper limit on how far code generation can be taken. It is just a question of how much effort you choose to invest, given the immediate goals of whatever project you are working on. At Doulos, we have been able to use the Code Generator to generate source code for all the examples and exercises we use in our training classes. We have not always chosen to use the generated code in our training materials because sometimes we want to show alternative or more advanced approaches, but we have been able to achieve much better consistency across our examples and exercises.

This provides one answer to the second question: why would you want to? A lot of the value of code generation lies in the fact that it can generate a consistent code base, and this would be of value to *everybody*. The point about consistency is that its value increases the more widely it is applied. Hand generated code almost always contains inconsistencies, both internal inconsistencies within the code written by a single individual and inconsistencies across the code written by a team. Further, although certain inconsistencies might be debated as justifiable in the context of solving the problem at hand, most inconsistencies in any code base are *entirely gratuitous*. Our experience has always been that by ironing out gratuitous inconsistencies in style, the effort of any team can become a lot more focused and productive. Possibly depending on their cultural background, many engineers strive to inject their individuality and personality into their code by adopting their own idiosyncratic coding preferences. In terms of creating a maintainable code base, this is always a bad idea. Using a code generator to nail down the coding style used in the non-user-defined elements of the UVM code base might even give engineers a sense of common ownership over the coding style decisions that they have made as a team and then captured in the code generator script.

Our next observation based on experience is that a code generator never does everything you want it to do. This may be a cause of psychological resistance to the use of code generation for many engineers: "The code generator will never be able to write the code that I can write, so I don't want anything to do with it." Actually, my experience

is that the code generator writes better code than I do, or at least more consistent code, and there is no motivation to vary the style of user-defined code fragments too far from the auto-generated norm.

Allowing fine-grain control over the generated code is problematic because the author of the code generator cannot predict everything that the end user will need. We employ a number of mechanisms to give the user control over the code:

- Pre-defined insertion points for user-defined code fragments. This mechanism works well when the insertion points can be identified in advance, and it is easy to add extra insertion points (by modifying the code generator)
- Control file switches to turn off automatic code generation for a particular region of code, which permits the user to replace automatically generated code with a user-defined code fragment. This is a catch-all solution that gives the user total control of the generated code, but it is an all-or-nothing approach and a blunt instrument. Once the user has turned off automatic code generation for a region of code, the benefits of code generation are lost for that region.
- Users can extend an automatically generated class and override individual methods using the factory. This is a pragmatic solution in some situations, but is arguably a marginal abuse of object-oriented programming principles if used to obliterate the structure and function of a base class and replace it with something else entirely. It also suffers from the same disadvantage as the previous point in that the advantages of automatic code generation are lost for whichever methods are overridden, particularly if they happen to perform significant operations such as build_phase or connect_phase.

It is not only that users want to push the code generator to handle new scenarios. As well as that, the coding guidelines captured by the code generator will continue to evolve. There are very few coding guidelines that can really be set in concrete or apply to every situation that can arise, and so the code generator will need to evolve along with the guidelines.

In my opinion, the more general consequence of the observation that you can never anticipate everything that will be required of the code generator is that you should work continually on beautifying your code generator script so that it remains easy to change and easy to maintain. We found a few things to be key:

- Maintain a simple, consistent overall structure for the code generator script
- Take care over the naming of internal variables that represent control file settings
- Keep the code templates as close to plain text as possible so that they are readable and easy to update
- Keep the meta-instructions used to generate code from the templates as regular, consistent, and simple as possible. For example, have a simple, consistent way of vertically aligning text
- Build a regression test suite and run it after every change to the generator script

Our next observation is that you should not be tempted to modify the generated code, not if you ever want to generate it again. We learnt through experience that this should be regarded as a clear-cut rule. The first time you change the generated code, you have in effect burnt your bridges with the control files and the code generation process. This is fine if and when abandoning the code generator is what you want to do, not but before. You must be able to delete the entire output from the code generator and regenerate it from the control files, code fragments, and DUT alone. Apart from anything else, this allows you to run regression tests on the code generator script itself. It also allows you to regenerate your UVM code base if ever it becomes corrupt.

Our second answer to the second question “Why would you want to” is that there is enormous value in being able to automatically regenerate the code that stitches everything together, particularly the envs, interfaces, and top-level modules. Once you start modifying the generated code and burn your bridges to the code generator, you no longer have the option to regenerate the top-level. If you really do need to depart a long way from the generated code, a better solution is to continue to regenerate the top-level automatically, inserting dummy agents to represent your own VIP as necessary, then use the factory to swap-in your own VIP in place of the automatically generated components. You will not benefit from automation in any connections made to your own VIP, but you can continue to benefit from automation elsewhere in the code.

Our final observation is not unique to code generation. It is to do whatever it takes to maintain backward compatibility between the evolving code generator script and old versions of the control files. There is always a temptation to think “I’ll just make some improvements: nobody will notice.” Don’t do it! Not only will you upset any users you may happen to be cultivating, but you will also break all your own old scripts, including your regression tests. Always be prepared to jump through hoops rather than introducing backward incompatibilities.

Deprecate old features if you want to, but don't break old code. (I wish all standards committees would take this lesson to heart.)

IX. SUMMARY

Limitations and Advice

- There are no hard upper limits to the capabilities of UVM code generation. It is a return-on-investment decision how far you take code generation
- A code generator always needs to evolve to handle new use cases and changing coding guidelines
- You should work continually to make your code generator script increasingly easy to modify
- Do whatever it takes to keep the evolving script backward compatible with old control files
- You must be able to regenerate the entire code base at any time
- Do not modify the generated code until you are ready to burn your bridges to the code generator

Benefits

- There is huge value in the consistency of coding style achieved using a code generator
- There is huge value in automatically regenerating the code that stitches everything together

REFERENCES

[1] John Aynsley, Doulos, "Easier UVM for Functional Verification by Mainstream Users", DVCon 2011, San Jose.
[2] Detailed Explanation of the Easier UVM Coding Guidelines, Version 2015-02-23, http://www.doulos.com/knowhow/sysverilog/uvm/easier_uvm_guidelines/detail/
[3] John Aynsley, Dr Christoph Sühnel, "Easier UVM - Coding Guidelines and Code Generation", DVCon 2014, San Jose
[4] Jim McGrath, juvb11.pl and juvb12.pl, http://forums.accellera.org/files/category/3-uvm
[5] The Easier UVM Code Generator, http://www.doulos.com/knowhow/sysverilog/uvm/easier_uvm_generator/
[6] The Easier UVM Code Generator – Reference Guide, http://www.doulos.com/knowhow/sysverilog/uvm/easier_uvm_generator/ref/
[7] The Syosil UVM Scoreboard, http://syosil.com/index.php?pageid=33
[8] Sutherland and Fitzpatrick, "UVM Rapid Adoption: A Practical Subset of UVM", DVCon United States 2015
[9] Sutherland and Fitzpatrick, "UVM-Light, A Subset of UVM for Rapid Adoption", DVCon Europe 2015