CREATING USEFUL CODING GUIDELINES FOR A VERIFICATION ENVIRONMENT

Ambar Sarkar, Principal Consulting Engineer, Paradigm Works Inc., Andover, MA

Abstract

Coding guidelines are considered useful for commercial software development environments. However, verification engineers often fail to adopt coding guidelines as they are not considered worth the extra effort. The author suggests that even when not followed strictly, many of these guidelines can save the team significant time and effort in the long run. A case study is presented, with a set of guidelines that can be used by similar projects as a starting point.

Introduction

Commercial software development projects adopt coding guidelines to promote maintainability, reuse, and improved performance of the software. These guidelines are usually a collection of coding rules that are followed by the development team. However, due to various schedule pressures and the extra effort required to adhere to these rules, verification engineers often perceive such guidelines as a waste of time, especially when not strictly followed. Our experience, however, proved otherwise; the observed benefits of adopting coding guidelines definitely outweigh the perceived disadvantages.

For a recent verification project, we looked around for a set of coding guidelines for the chosen hardware verification language (HVL). The HVL chosen for the project was Vera[2,5]. We were unable to find a set of guidelines that matched our project needs. We therefore created a set of rules based on existing guides for related languages[3,4], our past verification and software-development experience, input from project members, and various other sources[5].

We soon found out that some guidelines mattered more than others. Some needed to be modified to reflect specific project needs. Some proved vital--we couldn't even have linked all our code together if these guidelines were not followed. Some guidelines helped us in handling tool issues while some others saved us time by making it easier to debug and make code changes. In some cases, having a consistent rule was more important than the actual convention selected. And there were a few cases where following the guideline had no perceptible impact on our project.

This paper relates our experiences in creating and applying Vera coding guidelines on a real project, and identifies a set of guidelines that is a good starting point for similar projects elsewhere. A complete list of these guidelines can be found in [1].

The remainder of the paper is organized as follows. First, we describe the perceived benefits and disadvantages of using coding guidelines. Second, we describe how we derived our current set of coding guidelines. Third, we discuss the content and organization of the coding guidelines. Finally, we describe our experiences, followed by our conclusion.

Perceived benefits and disadvantages of using coding guidelines

Using a well-chosen set of coding guidelines generally results in code that is much more readable and understandable, especially to other developers when the original developer may be unavailable. As a verification environment often uses verification IP from different sources, enforcing specific conventions enables easier integration between diverse components. In addition, such guidelines can help in handling idiosyncratic language semantics and tool issues in a uniform way, while also potentially promoting efficient coding practices.

Choosing an appropriate set of rules is critical for a project, given the typical negative perceptions regarding coding guidelines in a schedule-driven environment. Adherence to coding guidelines is often perceived as a waste of time because some extra effort may be required. Some rules may seem artificial and may seem to require extra coding without any apparent benefit. Some rules may not even be applicable for a given project. For legacy and externally developed software, these rules cannot be applied since one may not wish or be able to make changes to such software. We created our rules with all such objections in mind, and after a few revisions, we were able to agree on a set of rules that address such issues.

Choosing coding guidelines for the verification environment

A brief description of the verification environment follows. The design under verification was a SOC targeted for the embedded networking applications. The design consisted of both externally and internally developed RTL, with a significant portion of it being external design IP. The verification environment consisted of both internally and externally developed verification software.

The verification team consisted of seven experienced verification engineers, with about half of them contractors. Familiarity with a HVL varied from those who had never used anything else other than a HDL to those who are experts in Vera. Half of the verification team members were experienced in other HVLs. Many team members lacked experience in object-oriented programming, while some were experts. The goal of this team was to maintain this environment and verify a number of similar SOCs down the road.

The guidelines were created based on other languages similar to Vera. Specifically, we borrowed some from C++/Java guidelines as they are object-oriented and have many similar language constructs. We considered various guidelines suggested in several sources such published books on Vera[2], mailing lists[5], numerous published articles, and knowledge from experts within our company. We also received feedback from the tool vendor regarding the coding guidelines. While we borrowed ideas that seemed applicable to Vera, we often modified these guidelines to suit our project needs.

We wanted to select coding guidelines that were easy to follow and intuitive. However, adherence to the coding guidelines was not made mandatory. While it was highly encouraged, we left it to the verification engineer to follow the guidelines. This decision was made to ensure that one followed the guidelines because it made sense and

did not seem to require too much effort. As we observe later, following the guideline strictly mattered more in some cases than others.

Coding guidelines contents and organization

Each guideline was stated imperatively, followed by an explanation whenever necessary. Next an example that illustrated the rule was provided. A summary of these rules is presented in Table 1. The complete guidelines can be found in [1].

Guideline #	Guidelines
Category 1) Programming Methodology Guidelines	
CG1	Declare a class type and the implementation of its methods (tasks and functions) separately
CG2	All class data members should be declared local or protected.
CG3	Never use constant literals directly in the code.
CG4	Always use virtual ports and pass bind names as references into tasks
CG5	Do not sample signals in expressions
CG6	Remember object handles and not the objects are passed as arguments in Vera
CG7	Avoid unnecessary use of non-blocking drives in testbench code
CG8	Always use skew when sampling or driving signals into the DUT.
CG9	Do not rely on the order of execution of threads.
CG10	Avoid shared variables across multiple threads, use shadow variables whenever feasible.
CG11	Always check returns for system calls such as alloc, new
Category 2) Naming Programming Constructs	
CG12	Use Hungarian notation for all names except for macros and enumerated constants.
CG13	Use uppercase and underscores for all macros and enumerated constants.
CG14	Use upper case for the first letter of the name of each type declaration..
CG15	Use lower case for the first letter of any variable
CG16	Use lower case for the first letter of any bind instantiation
CG17	Name tasks or functions appropriately.
CG18	Give meaningful name to the task or function arguments in prototype declarations
CG19	Use appropriate suffixes for class names of the verification components.
CG20	Use unique names for enumerated constants.
CG21	Use appropriate conventions when naming interfaces, virtual ports and port bindings.
CG22	Use appropriate naming conventions for static and shared objects.
CG23	Use appropriate naming conventions for global variables.
Category 3) File Organization	
CG24	Generate header files automatically for classes.
CG25	Every file should have the standard file header at the top.
CG26	Define one class per file, especially for large classes
CG27	Name the file based on that class.
CG28	Every header file, including class definition files, must have a read-once latch.
CG29	Keep port declarations with the class that uses that port as a parameter in its methods
CG30	Keep bind declarations in files that pass binds as parameters to tasks.
CG31	Avoid unnecessary #include's
CG32	All RTL hierarchy paths should be declared in a separate header file.
CG33	Use appropriate filename extension conventions.
Category 4) Error And Debug Messaging	
CG34	Use a standard format for all error messages.
CG35	Use a standard debug message format.
Category 5) Performance Considerations	
CG36	Minimize use of dynamic binding to ports (using signal_connect).
CG37	Avoid using associative arrays for ranges of less than 1000 elements.

Table 1. Summary of Vera coding guidelines

The guidelines were grouped roughly into the following five categories:

Category 1) Programming Methodology

These rules suggest how the HVL code should be organized, as well as rules on using specific HVL constructs. Most of these rules are primarily applicable to Vera, though similar rules may be adopted by other HVLs . Here is an example of such a rule:

Rule: Do not sample signals in expressions
Explanation: According to Vera semantics, a signal is sampled immediately (asynchronously) if it is embedded in an expression. If not embedded in an expression, the signal is sampled in the next clock edge. Since in general most sampling is done synchronously, sampling asynchronously may lead to race conditions and therefore unexpected behavior. Therefore, it is recommended that one separate the sampling of a signal from its use.
Example:

Instead of :

```
dataPls1 = dut.$Data + 1'b1;
```

Use:

```
data = dut.$Data;
dataPls1 = data + 1;
```

Category 2) Naming Programming Constructs

These guidelines recommend how to name various Vera programming constructs. Most of these rules were derived from guidelines in other programming languages such as C++/Java. An example of such a rule is given below:

Rule: Use appropriate suffixes for class names of verification components. The following siffixes are recommended:

Bfm	for Bfm objects. Eg. Spi4Bfm
Chkr	for Checker objects. Eg. Spi4Chkr
Mntr	for Monitor objects. Eg. AhbMtr
Cvg	for Coverage objects. Eg. UsbCvg

Category 3) File Organization

This category recommends how to organize the code among various files. It also includes suggestions for naming various files, their suffixes, and helpful compilation conventions. Many of these rules were tool specific. Here are a couple of examples:

Rule: Define one class per file, especially for large classes.
Explanation: This principle improves code readability, and avoids unnecessary recompilations.

Rule: Name a file based on the class it defines.
Explanation: This rule helps in quickly identifying which file contains what class.

Category 4) Error and Debug Messaging

This category specifies error and debug message formats as well as their contents. These rules are independent of the HVL. Here is an example:

Rule: The following format is recommended for error messages:

```
[Timestamp]     <ErrorType>:  ObjectName:<tab> Message summary in one line
                <Details>
```

Example:

[120333] ERROR: Dma Channel 1: Failed to see DONE_
Dma Status Register Contents:
....

Category 5) Perfomance Considerations

This category covers rules that had potential impact on the tool performance. Examples are rules for hooking up the testbench with the RTL, suggestions on choice of appropriate data structures, etc. We had few rules in this category, primarily because further analysis of the verification code was needed to identify appropriate rules. As a start, we had a couple of rules suggested by the vendor. Here is an example:

Rule: Minimize use of dynamic binding to ports (using signal_connect).
Explanation: Use static binding (using interfaces) instead, whenever feasible. The static interface method uses the direct kernel access routines which are faster. If one does not need to dynamically change the binding of a port, using interface files is preferable.

Experiences with coding guidelines

Table 2 provides a summary of our experiences over a span of about ten months. For each adopted guideline, it specifies how strictly each guideline was followed, as well as whether adherence to the guideline proved useful.

A rule was considered to have been followed strictly when the verification engineer generally followed the rule with very few exceptions. A rule was considered to have been followed reasonably if it was followed at least half of the time. Otherwise, we considered the rule to have been abandoned.

A rule was considered useful if we felt like we saved time in the long run or if we lost some time unnecessarily by not following it. On the other hand, even if a rule appears useful at first glance, but our project did not seem to take advantage of it, we considered it belonging to the "Does not matter" category.

Among the thirty-seven guidelines, thirteen were strictly adhered to. Fifteen were adhered to reasonably well, though there were some exceptions. The remaining seven were not followed much. In terms of usefulness, about twenty-one rules proved extremely beneficial in our project, while eight proved to be fairly useful. Out of these twenty-one rules, two were not followed but were discovered to be fairly beneficial in hindsight. The remaining ten proved not to be of use within our project, but may have been useful in other environments.

Experience with Programming Methodology Guidelines

The programming methodology guidelines were useful in helping the relative newcomers to Vera. The guidelines helped them to avoid some of the common pitfalls that such a user can make. An example was CG5, which recommends that the signals should not be sampled in expressions. In Vera, sampling signals in expressions causes the signals to be sampled asynchronously, which could have caused unintended race conditions between the sampled event and the remaining testbench activities. As most of our observed events were synchronized to a clock, by adopting CG5 we were able to avoid any such confusion.

Guideline #	Adherence			Benefits		
	Strict	Reasonable	None	Very	Somewhat	Does not matter
Category 1) Programming Methodology Guidelines						
CG1		Y			Y	
CG2			Y			Y
CG3	Y			Y		
CG4			Y	Y		
CG5	Y			Y		
CG6		Y		Y		
CG7	Y			Y		
CG8	Y			Y		
CG9	Y			Y		
CG10			Y			Y
CG11			Y	Y		
Category 2) Naming Programming Constructs						
CG12		Y			Y	
CG13		Y		Y		
CG14		Y			Y	
CG15		Y			Y	
CG16		Y				Y
CG17	Y			Y		
CG18	Y			Y		
CG19		Y			Y	
CG20	Y			Y		
CG21		Y				Y
CG22			Y			Y
CG23			Y		Y	
Category 3) File Organization						
CG24	Y			Y		
CG25		Y				Y
CG26		Y		Y		
CG27		Y		Y		
CG28	Y			Y		
CG29		Y		Y		
CG30		Y		Y		
CG31			Y			Y
CG32			Y			Y
CG33	Y			Y		
Category 5) Error And Debug Messaging						
CG34	Y			Y		
CG35		Y				Y
Category 6) Performance Considerations						
CG36	Y			Y		
CG37			Y			Y

Table 2. Summary of experiences

Another example of a very useful programming guideline was CG11. This rule specified that one should always check the result of a system call, whenever possible. While this is a fairly common-sense rule, it required some extra work on the programmer's part. Some of them decided to skip the check on some system calls that seemed fairly robust. Unfortunately, as it turned out, one of our verification IP providers changed some of their constraint class names in a new release. Once we upgraded to this release, things stopped working mysteriously. This wasted about one day’s worth of a single verification engineer before we realized the problem, as it wasn't obvious where the failure occurred. Regardless of the actual source of the problem, adhering to CG 11 strictly would have saved us significant time.

Interestingly, CG2 proved to be one that did not seem to matter in practice. It required all member variables of a class to be declared local or private, and required one to define access methods for each variable. This ensured that any user of the object would obtain a handle to the variable through an access method. By doing so, one can prevent any external class from accessing any variable without the parent class being aware of it.

In practice, however CG2 proved to be more strict than necessary. Instead, the programmer decided which variables were better declared private, and the rest were left declared as public. By picking carefully which variables should or should not be exposed for access to other objects, we still followed the principle of information hiding, while saving extra typing that the rule required.

Experiences with Naming Programming Construct Guidelines

The rules in this category were quite useful for the reason of code readability and maintainability. While few rules in this category were strictly followed, majority were followed reasonably well. The key point was to choose consistent and meaningful names for these constructs. Such names helped in quickly grasping the functionality of each object.

An example of extremely useful guideline is CG20, which required giving unique names for enumerated constants. This enabled us to integrate with verification IP from multiple sources without any fear of name-space collisions.

Some of the coding guidelines ended up not mattering much as they were not followed strictly. CG16, CG21, and CG22 dictated rules regarding how exactly to name various Vera type instances so that one could easily understand the exact type and function of the object from its name. In practice, since these rules were not followed strictly, one had to always look up the objects context to determine its true functionality.

In summary, while most of the rules related to naming constructs are useful, some of them are rendered useless if not followed strictly. Some guidelines, such as CG13, CG17, CG18, and CG20 are still very useful even when they are not followed strictly.

Experiences with File Organization Guidelines

The rules on file organization were extremely beneficial. We would not have been able to compile and link all the verification code from various verification IP vendors with our own code easily without CG24, CG28, CG29, and CG33. The primary advantage of these guidelines was being able to compile and link verification related software from various sources.

The guidelines CG24, CG28, and CG33 were critical for creating common scripts that compiled and linked all our code. For example, CG28 ensured that the contents of an include file did not get included more than once even if the file itself was included more than once in the hierarchy of include path of a file.

CG29 and CG30 would have been useful in avoiding unnecessary recompilations whenever changes were made to the interface files or the associated pin names in the RTL. Specifically, CG30 made sure that if there were changes in the RTL pin names, these caused recompilation in only those files that included the interface declarations. Unfortunately, this rule was not strictly followed, causing many unnecessary

recompilations in the early stages of the project, when the various port and signal names were not fully decided upon.

Some guidelines did not matter much. For example, CG31 required that one avoid unnecessary includes, as they would cause unnecessary recompiles. In the sophisticated software environment that we were using, it was often a significant amount of effort trying to identify which ones were strictly necessary and which weren't. CG32 required that all HDL path hierarchies be separated into one file. In practice, having the interface file contain any reference to the HDL paths effectively performed this task, so it was not necessary to create yet another file with just the HDL paths.

Experiences with Error and Debug Messaging Guidelines

These rules primarily recommend the format for error and debug messages. The format for error messaging proved fairly useful, as it let us write automated scripts that quickly identified the failures from the log files. On the other hand, the debug messaging format was not used much, even though convenient APIs and classes were provided to make it easier to follow. Interestingly, the programmers chose to use printf statements and deleted or commented the printfs when they felt they had debugged the problem satisfactorily.

Experiences with Performance Consideration Guidelines.

We did not have many rules in this category. Rules CG36 and CG37 were recommendations by the tool vendor. CG36, which recommended how to hook up the HDL with Vera, was adopted and had saved us some time according to the Vendor.

Interestingly, CG37, which recommended avoiding the use of associative arrays for small, non-sparse ranges was not followed in spite of potential performance consequences. This was done for a couple of reasons. First, it is a convenient construct to use in many situations where the exact size of the array may not be known, but array itself was not sparse. Second, it was premature to think about optimization without doing an overall performance analysis and addressing the bigger bottlenecks first.

Conclusions

Coding guidelines are essential for verification projects. These rules help us work around tool limitations, handle language idiosyncrasies, integrate software from diverse sources, and improve maintainability of the verification software. However, it is important to realize that depending on the project, some rules may be more important than others. While some rules are not beneficial if not followed strictly, we note that many rules provide value even if they are followed to some extent. Failure to adhere to some of these rules can cost the team significant time, and the benefits observed easily justify the effort spent in adhering to the guidelines.

Acknowledgements

I would like to express my sincere thanks to the NetSilicon verification team, Synopsys, and my Paradigm Works colleagues for their various insights and feedback.

References

1. Vera Coding Guidelines , Paradigm Works, Inc., Dec 2002
   http://www.paradigm-works.com/technology/vera-coding-guidelines.pdf
2. Art of Verification with VERA, Haque, Khan, Michelson, Verification Central, Sept 2001
3. Programming in C++, Rules and Recommendations, Henricson and Nyquist, Ellemtel Telecommmunications Systems Laboratories, 1992
4. Code conventions for the Java programming language, Sun Microsystems, http://java.sun.com/docs/codeconv/
5. OpenVera website: http://www.open-vera.com