Apples versus Apples HVL Comparison Finally Arrives

Comparing OVM SystemVerilog to OVM ***e***

Brett Lammers
Cadence Design Systems, Inc.
2670 Seely Avenue
San Jose, CA 95134
1-408-943-1234
brettl@cadence.com

Riccardo Oddone
Cadence Design Systems, Inc.
2670 Seely Avenue
San Jose, CA 95134
+39 (025) 755 8202
ric@cadence.com

ABSRTACT

Over the past few years the discussion of hardware verification languages (HVLs) has come full circle. At first, verification teams tried to assess the strengths and weaknesses of individual language features with the goal of creating their own verification libraries and environments but generally without the context of a reuse methodology. As these groups became more sophisticated and sought to exchange and reuse verification IP (VIP), they coalesced on the two IEEE standardized verification languages – 1800 SystemVerilog and 1647 e and moved toward the industry supported methodologies and libraries built with these languages. With the advent of a single methodology implemented in both languages – OVM multi-language – the discussion has returned to HVL features but now that the reuse methodology known, a clear apples versus apples comparison is now truly possible.

As mentioned, methodology is the biggest advantage verification engineers have to cut through the noise and put the language features in context. Possibly the most clear division is between design and verification which helps put into context the features of the verification languages – assertions, constraints, interfaces, etc. – as well as the design languages they interface to including IEEE 1364 Verilog, 1076 VHDL, and 1666 SystemC Another classic comparison is the efficiency of coding, but in the methodology context this is split between the test writer and the verification IP developer. This leads to the context of reuse and the language elements that enable verification engineers to account for change within a project, through project integration, and between projects. When the e and SystemVerilog are set into these methodology elements, features comparisons like AOP versus OOP, the use of factory patterns, randomization/generations schemes, SVA versus e assertions, tool support, and more become apparent and compelling.

The choice of HVL was once a murky process which resulted more in a vendor choice then an optimized technology selection. With the advent of HVL standardization and the popularity of consistent, open, interoperable methodologies, verification engineers can once again start the debate of language merits. The difference now is that whether that debate ends with SystemVerilog or e, the verification engineering team can make the selection on technology merit and maximize the productivity, predictability, and quality of their projects.

Keywords

HVL, SystemVerilog, e, OVM, multi-language, OOP, AOP

1 INTRODUCTION

The purpose of this document is to explain the main comparison points between OVM ***e*** and OVM SystemVerilog when used for verification. Knowing it is hard to carve out the time to sit down and read a lengthy document on the subject, this summary document is intended to give you, as a verification manager or expert verification engineer, an overview of the concepts involved as well as some links to additional reference material. This document assumes some basic knowledge of OVM SystemVerilog and OVM ***e*** as well as OOP fundamentals. Therefore, it is not intended to replace the language manuals or methodology guides for either language.

2 WHY IS THE VERIFICATION LANGUAGE CHOICE IMPORTANT?

A programming language is, of course, how to instruct a computer to execute a certain task. However, look at any experienced programmer and you realize that their favorite programming language is much more then just a means to communicate with a computer. A programmer's favorite language and programming paradigm shapes how they think about and tackle programming problems. This of course is no different in verification. If it has not already, the verification language choice will affect the verification team's view of verification and how they attack the problem. Before we dive into the more specific comparison points, let's remember some the characteristics of a verification project and how the language choice affects those characteristics.

2.1 Verification vs. Design

This is not referring to hardware design versus hardware verification but rather the difference between creating something and verifying that it works. When creating something the problem is fairly well bounded by some set of requirements or assumptions. However, the verification problem is very much unbounded as it needs to model all the different situations and interactions that the design will encounter. This can result in a large number of required variations or scenarios. It is important that your verification language matches this concept and gives you the power and flexibility to create as many of these situations and interactions in the most efficient manner possible.

2.2 Single application vs. multiple applications

In verification, each test case that is developed is really its own application. Depending on the complexity of the design, each test case may need a significant amount of specific tailoring to present different behavior to the device under test. Ideally, a verification language will allow you to share as much functionality between every different test case while minimizing the amount of code needed to specify the differences between test cases.

2.3 Need for Efficient Coding

It would seem to follow that the less actual verification code that a verification engineer has to write, the quicker that the verification environment will start testing the design. As mentioned previously the verification task is unbounded but the schedule is not. The time taken to create the verification environment takes away from the time actually testing the design. Because of this, a verification language needs to support automation and code reuse to make the verification engineer as efficient as possible in getting to the actual testing. However, it is important that this automation and reusability does not diminish the ability for the verification engineer to control the environment. Without the control, it will be difficult to target the specific corner cases that may be needed to really stress the design. In a similar fashion, it is also important that by raising the level of efficiency through automation, the code does not become obscure and hard to debug. If it does, this will also slow the critical process of developing the verification environment.

2.4 Accounting for Change

A verification project is very fluid and somewhat unbounded. There is always more to verify. This again ties back to the idea of creating something vs. verifying that it is correct. Verification environments also need to quickly adjust to enhancements to the environment as well as design changes that occur over the course of the project, or from project to project. Adapting to these changes in the most efficient and safe manner possible requires reusing as much code as possible and rewriting as little as possible. It is important that the verification language and the associated methodology support these changes without jeopardizing the integrity of any existing verification code.

2.5 Making use of non-verification resources

Verification teams often need to make use of other resources outside the team, like designers, to complete the verification. In many cases, a designer will not have this kind of familiarity with the verification environment, nor do they have the time to really dig in and learn the code set that makes up the verification environment. However, to be successful in helping to achieve verification complete they will need to interact with the environment in a detailed way. This means that the verification language, the supporting methodology, and the test writer API need to facilitate both ease of use as well as a high level of control.

2.6 Availability of Engineers and Tools

When the verification team forms it often pulls existing human resources and tools together to meet a specific time and resource budget. The existing knowledge of the engineers and the existing capabilities of the tools often create the basis for the HVL decision. That base decision is then weighed against the verification project goals along with the training and tooling necessary to achieve those goals. Additionally, other contributing factors like how many design and verification languages will come from existing IP and how the tooling environment is able to support that language structure further drive the ultimate decision. In some cases, the language choices are fully within the control of the team, and in some cases those choices may be dictated by the availability of verification IP.

2.7 Availability of Verification IP

One of the fastest ways to save time on a verification project is to be able to reuse existing verification code. Reusing code, both saves the development time of creating that code, as well as provides confidence through prior use. Of course, in order to make use of existing IP it has to exist and it needs either to be written in the language of choice or have some sort of multi-language interoperability support. A common reuse case is reusing an entire standalone verification environment inside another one, as is needed when moving from block level verification to chip level verification. When making this progression, often times the reuse of the block level environment must be configurable in many different aspects to achieve the different verification goals of both the block and system level environment. It is important that the verification language facilitates the developer of the block level verification IP to reach the proper level of detail for block level verification, while also giving the system level developer the control to configure that lower level IP.

2.8 Performance

It should come as no surprise that when using a constrained random environment the more tests that can be run with different and meaningful behavior the more coverage of the design there will be. However, the number of simulation machines, licenses, and time to complete the regression is limited. This means that the faster an individual test runs the more tests can complete. This is of course an important metric but it is also important to weigh any tradeoffs that may come with that faster simulation time. It might seem at first desirable to invest one time in a faster environment, considering that it would need to run hundreds or thousands of times per regression. However, there are other factors to also consider such as overall development time, the likelihood that bugs will be introduced when making enhancements or doing maintenance, and the overall effectiveness of the environment in reaching your coverage goals. These kinds of non runtime performance metrics are not what you normally think of when evaluating performance, but are equally critical in reaching the overall goals of verification complete for a project. Ideally, a verification language gives the verification engineer the best of both worlds: The ability to develop the environment quickly while carefully targeting the appropriate verification goals, as

well as the runtime performance to get all the testcases done in a reasonable amount of machine time.

3 WHAT ARE THE TECHNICAL COMPARISON POINTS?

Now that we have some ideas as to where language matters let's take a look at some of the differences between OVM SV and OVM ***e.*** The following is a list of the important comparison points that we will cover in more detail later in the detailed sections of the document. For each of the points in the list this section will explain how that point applies to verification.

3.1 Aspect Oriented Programming (AOP) in *e* vs. Object Oriented Programming (OOP) in SystemVerilog

AOP is a key difference between ***e*** and SystemVerilog and very applicable to the verification problem. AOP can be thought of as a super set of OOP, in the sense that a user generally has to apply OOP concepts to successfully apply AOP. Both methodologies share the concepts of objects and enable the user to organize their code around those objects. Figure 1 illustrates the relationship between AOP and OOP. Because of this relationship, AOP languages like ***e*** support both OOP and AOP.

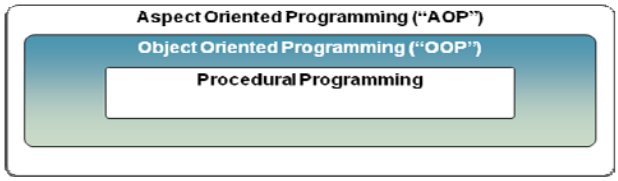

Figure 1 - Programming Paradigm Relationships

However, AOP goes beyond OOP in providing the user more flexibility to organize code modules not only by objects but also by cross-cutting-concerns. A cross-cutting-concern is functionality that touches multiple objects in the code set. In verification, these cross cutting concerns can include DUT related functionality such as operation modes as well as verification related functionality like coverage collection or checking. Essentially, the test itself is a cross-cutting-concern as it configures and constrains many objects within the testbench. Figure 2 illustrates this concept. Each of the colored bands traversing the different verification environments represents a concern or aspect that is shared across all of those components.

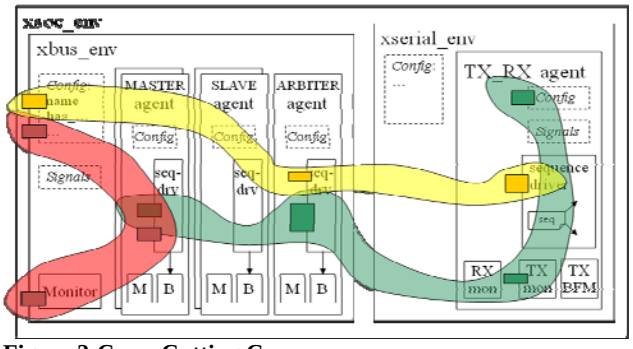

Figure 2-Cross Cutting Concerns

In order to manage these cross-cutting-concerns, an AOP language like ***e*** allows the user to extend objects in the environment from within different modules without creating new types. An AOP compiler then collects up the various extensions to form the final runtime objects with all the appropriate concerns included. This concept is extremely useful when maintaining code or reusing existing code. AOP is not unique to ***e***, and has been around in the software industry for a while now. Even though AOP has been around in the software industry for quite some time it seems that this idea of test writing in verification presents a problem that is uniquely suited to AOP. A reference for this topic is the book *Aspect Oriented Programming with the **e** Verification Language* by David Robinson[1].

Since AOP is a superset of OOP, it is important to realize that AOP requires some of the same software architecture techniques used in OOP. Just like in OOP, it is good practice to go through some planning to organize both objects and their associated concerns into the appropriate modules. This will prevent code that is difficult to read and hard to maintain. Many OOP users would argue that the more strict nature of OO is actually advantage as it allows for rigid control over the code's functionality and design. While that is true, if carefully managed, it is extremely powerful to use the flexibility of AOP extensions to methodically update the environment with concerns that were missed in the initial planning or are the result of changes that have occurred in the project. This sort of "after-the-fact" manipulation can be difficult in a standard OOP environment. However, it is also true that the same power that comes through this flexibility can also become a detriment if not properly managed through the proper methodology and proper planning. Figure 3 shows an example of this organization and how a base monitor module can be augmented by other modules that add aspects that may be important to different use models of that monitor. In the figure each of the boxes with colored bars below the monitor box represents a separate module that extends the base monitor and adds the associated concern.

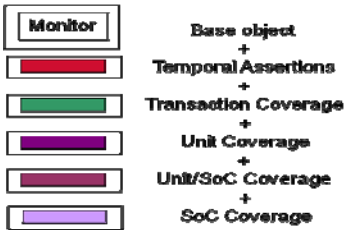

Figure 3 - Organizing Concerns

To further illustrate the update concept just explained, image that a verification engineer is working on testing an SOC and would like to make use of the monitor but needs to add some functionality to gather SOC level coverage. In this case, the engineer would just add the module represented by the last box in the figure and update the base monitor for SOC level coverage

3.2 OO Factory Pattern

This is a OO design pattern, used in other OO programming languages, that allows an OVM SystemVerilog

user to capture some of the AOP advantages just discussed. The factory allows the user to register various types such that when objects are defined they use the factory registered type. If set up properly, the type registrations can be overridden in such a way that, when a declaration calls for the original factory type it actually becomes the new type. This concept is extremely powerful in SystemVerilog and like AOP allows the code developer some flexibility to replace object types at runtime. However, it is important to realize that the factory concept does not cover all aspects provided by AOP. SystemVerilog Factories are an advanced topic in SystemVerilog that requires careful planning to be used properly. Since the factory pattern is not built into the language, as AOP is in ***e***, one of the most important steps in this planning process is the decision to consistently use factories in the code. If this is not decided upfront and adhered to throughout the code, the potentially error prone activity of retrofitting existing code to use factories can be costly. Also, it is important, to have this upfront planning to keep the number of registered factory overrides controlled. Without this planning, the number of registered factory patterns can become unwieldy and like uncontrolled AOP usage, result in difficult to read and maintain code. For more information on SystemVerilog factories refer to the OVM SV documentation that can be downloaded from OVMWorld (www.ovmworld.org).[2] Another interesting comparison point related to factories and general OOP inheritance in SystemVerilog is the fact that certain verification elements cannot be redefined or reused through the factory or even OO inheritance as they can in ***e***. For example, functional coverage constructs. Using AOP in ***e***, a user can extend functional coverage to add additional coverage items or refine goals. This ability becomes extremely important when reusing environments from block to system level or even reusing third party verification IP that needs to be configured to a specific design.

3.3 Randomization and Generation Schemes

There are a number of differences between ***e*** and SystemVerilog in the area of randomization and generation. For the purpose of this discussion, it is important to separate generation from randomization. Generation refers to the engine or process that creates the randomized variable or randomized structure. Randomization refers to the process by which the user controls how an entity will be randomized and the actual values that the variables will arrive at. In the area of randomization, the ***e*** language implements a concept referred to as "infinity minus". This means that in ***e*** one must specify that a given field or variable is *not* randomized. As with most languages in the world, in SystemVerilog, the verification engineer must specify which fields are to be randomized. This requires careful up front planning in order to determine the proper fields and their interactions. It might seem like a small difference at first, but the "infinity minus" approach acts like a built in safety by creating interactions that were never originally thought about.

Another important difference between ***e*** and SystemVerilog in the area of generation is memory allocation. In SystemVerilog, memory must be manually allocated for a given object before that object can be randomized. In ***e***, generation automatically allocates memory saving the code writer a little code for each field. However, the code savings is not really the key advantage. Instead, the real advantage is how this automatic allocation allows you to easily model complex transaction scenarios and really enables the "infinity minus" approach for complex compound structures. Consider the following example: Imagine modeling a set of CPU instructions in SystemVerilog. Table 1 shows a few hypothetical CPU instructions and how they could have different operand contents

Table 1 - Example CPU Instructions

opcode	operands		
wr_mem	address	data_h	data_l
jmp	address		
add	dest_addr	value1	value2
rd_mem	address		
…	…	…	…

In this case, you wish to create a list of CPU instructions where each instruction is a different opcode, and therefore as shown has a different structure and a different memory footprint. This is extremely difficult and inefficient to model when manual allocation is required. One option would be to create some procedural code to manually allocate each list item individually. In ***e***, this operation is as trivial as creating a list of homogeneous structures and constraining the subtype field. To further illustrate the difficulty, imagine that we want to add another instruction subtype. In SystemVerilog, the user would have to add the new type and update the manual code that created our list of CPU instructions to allocate the new type. In ***e***, the user would only have to add the new type and the code dealing with the list would automatically pick up the new type seamlessly.

3.4 "When subtyping" in *e*

This concept is unique to ***e*** even among other AOP languages. "`when` subtyping" allows the user to define subtypes with their own specific characteristics based conditionally on the value of a field in the base type. Then using random generation and simple constraints on these "when determinant" fields the user can easily change the structure from one subtype to another. To illustrate this concept imagine the CPU instruction example, just discussed, which has different field definitions and structure based on the value of an opcode field.

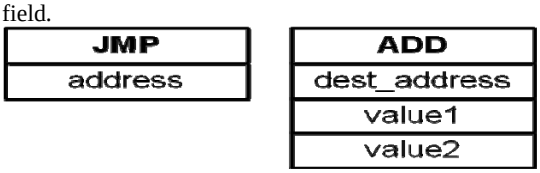

Figure 4 - JMP and ADD structures

Figure 4 illustrates the idea that each instruction is still a CPU instruction, but `when` a particular instruction is a *JMP* instruction it has a different structure then when it is an *ADD* instruction. When combined with the "infinity minus" randomization and automatic memory allocation, discussed earlier, `when` subtyping is an extremely valuable and easy way to express variation from one transaction to the next. IN the situation where we want to send different CPU instructions representing some program streams, we would want to generate

one instruction after another worrying only about the kind and not the associated structure. This "variation modeling" is extremely difficult to capture in an OOP language like SystemVerilog. There are however, a number of OO design patterns to mimic this capability. One solution is to create a flattened object which is a union of all the subtypes. Based on the value of some type field, the code using that flattened object would interpret it differently. Another possible solution is to create a new class for each subtype. Unfortunately, these solutions can become cumbersome as the number of variations grows but is none the less successful if managed appropriately.

3.5 Design Related Assertions

Both ***e*** and SystemVerilog support assertions. However, SystemVerilog assertions (SVA) have some advantages over those created in ***e***. One strong advantage is the ability to use SVA's in multiple verification flows. SVA's can be used in formal analysis, they can be used in the traditional simulation flow, and they can be synthesized for use in hardware acceleration environments. Another reason SVA's have some advantage over ***e*** assertions is the idea of maintaining assertions within the design by the designers. This is a great area to share some of the verification work with the design team without forcing them to learn a completely new language or new programming paradigms like AOP or OOP. Designers can help by creating both checking and coverage related assertions, which inherently reflect the design intent, as they are creating the design.

It is important to remember that SVAs also work very nicely in conjunction with an ***e*** testbench environment. Sort of the best of both world type scenario. In fact, there are a number of hooks added into Specman to allow SVAs to "call back" into the ***e*** testbench for additional processing.

3.6 Macros

Traditionally, a lot of programmers shudder at the thought of using macros in their code because they are often hard to debug and often affect the code's readability. However, they play an important role in both the SystemVerilog and the ***e*** languages. In OVM SystemVerilog, over 350 different macros help programmers in accomplishing common verification tasks and simplifying the coding effort needed to achieve these tasks. In ***e***, macros provide a powerful way to, in a sense, extend the language itself. Somewhat in contrast to macros in most other languages, ***e*** macros are not just text substitution, but rather appear very naturally just like other language constructs. In fact, macros are so natural that probably many ***e*** users that don't realize that some of the most common functionality is actually implemented using ***e*** macros. In addition to the natural feel, macros in ***e*** can be debugged with much the same capability as normal code. This additional debug support and the natural look and feel remove the old stigma that historically surrounds macros. For more information on ***e*** macros and how to use them, refer to the Macros chapter of the ***e*** Language Reference manual in the Cadence help documents.[3]

3.7 Tool Support

While the focus of this document is intended to be on the languages themselves it is also important to look at the tools that support these languages. One area where tool support greatly affects the ability to successfully use a language is code debug. Debugging has always been a huge portion of the verification effort and debugging contradicting constraints can be one of the more complicated and time consuming debug activities. The Intelligen debugger in Specman provides the user with an easy to navigate GUI debug view of all the information related to the generation of fields. This tool allows the user to browse through interactions between constraints and see how various values were chosen, In contrast, most SystemVerilog simulators have only recently gone beyond only printing out a detailed error message when a problem occurs and started adding constraint debugger functionality. It is difficult without theses debugging tools to dig into the environment and see why generation occurred the way it did.

Of course, in addition to generation debug, debugging the verification environment and debugging the design are also very important for verification success. Often times, these two activities go hand in hand but in some cases done by different resources. Many times a verification engineer needs to debug through a good portion of the design under test to find a bug in the verification environment. The reverse is also true of a designer that needs to gather debug information from the verification environment to isolate a design bug. Again, this means it is important to have the proper tools to make this effort as efficient as possible.

3.8 Compiled vs. Interpreted Mode Support

This concept involves the tradeoff between runtime performance and code control. In general, all SystemVerilog code, the testbench, the design, and the tests are all compiled together into "executable code". Compiled code of course has the advantage of running fast and therefore is desirable for runtime performance. However, having to compile in all of the tests at once can present some compile time performance problems if the number of tests is large. Specman takes further advantage of ***e***'s AOP characteristics and allows the user to mix and match interpreted code with compiled code. This allows the user to layer an interpreted mode test on top of a compiled environment which gives the perfect blend between compile time performance and runtime performance. Another area where this mix can be useful is in debug. Imagine debugging a failing test case that was layered on top of a large compiled verification environment. The problem is found to be something in the base compiled code. Instead of recompiling that code, the user can take advantage of AOP and layer on an interpreted mode fix to the compiled code without recompiling. This can often save precious debug time while trying to test out a proposed fix.

4 WHICH LANGUAGE FITS BEST?

In this section, let's look at a few common scenarios that you might encounter and why a certain language may be a better fit then others.

4.1 Raising the Level of Reuse

One of the biggest challenges in reuse is creating something flexible enough to meet the needs of multiple users in multiple situations with the least amount of maintenance to the core code. Figure 5 illustrates how a smaller block level piece of IP can be reused at the system level as well as across the different systems in the larger ecosystem.

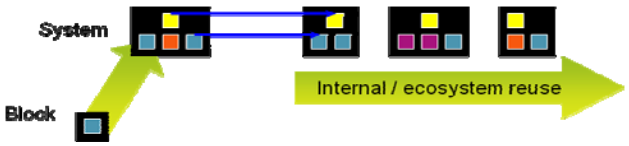

Figure 5 - IP reuse

It should be obvious that the concepts and methodologies provided in OOP as well as verification methodologies like OVM facilitate this reuse. However there are some characteristics of verification reuse that can benefit from the additional flexibility and control of an AOP language like ***e***. Such examples are sharing code between projects, layering constraints onto existing data items, or creating a commercial VIP with IP that needs to be hidden. While the concept of the SV factory and OOP objects can facilitate some of this reuse it can become very difficult if the shared code needs to remain static, is invisible to the user, or needs to express many independent variations. To really achieve these kinds of reuse cases, it is extremely advantageous to go beyond the capabilities of standard OO software design and even the SystemVerilog factory to a language that supports AOP. Whether you are creating standalone IP that you deliver to other groups or you just want to increase the productivity of your verification group by increasing reuse, ***e***'s AOP characteristics likely make it the better choice.

4.2 Full Chip Verification Environments

Similar to the previous reuse discussion there are scalability issues that require additional configurability and control when moving from the block level to the full chip level. It is important to factor in ***e***'s maturity for this case. Having been around longer, and already used by verification engineers to solve the full chip verification problem, it has picked up some additional methodology to support full chip issues such as reset testing. If you are looking to address this full chip verification case, it would be good to look into OVM ***e*** and see how it goes beyond OVM SV and ***e***RM to solve the problems presented in full chip verification. For more details, check out Cadence's OVM ***e*** reference manual in the OVM ***e*** package delivered as part of the OVM Multi-language Release. The Library can be downloaded from the contributions area on OVMWorld (www.ovmworld.org)[2] For cases needing to support high levels of reuse, ***e*** may be a better choice.

4.3 Designers Having Dual Responsibility for Design and Verification

In general, this is going to be a challenge when adopting any of the advanced verification languages for advanced verification. SystemVerilog having roots in Verilog seems like an obvious choice here. Assuming that these design resources do not have the bandwidth to take on a software design project, it may be better to focus them on the simpler *module based* SystemVerilog environment. This will allow them to pick up some quick and dirty verification enhancements like randomization and coverage without having to learn more strictly software concepts such as OOP or AOP. While SystemVerilog is likely the better choice for this case, there are a couple of additional notes to keep in mind:

- What is the long term goal/plan? It is likely that there will be a future desire to reuses the verification code created by the designers from one generation of the design to the next and maybe even across groups. To be most efficient and enable this later reuse, it is best to create the module based environments following the OVM for SystemVerilog. The OVM specifies an architecture, terminology, and process that will help facilitate this later reuse.
- Once created, these SystemVerilog environments can be reused in an ***e*** environment for example at the system level. This reuse is again facilitated by the multi-language support in the OVM.
- If this module based environment is intended for reuse by multiple project and to possibly become verification IP, it may be a good idea to consider the use of ***e*** for the same reasons mentioned above in the "Raising the level of reuse" discussion.
- A designer who tries to implement an OOP based environment without proper training will likely experience some difficulty. Advanced verification requires knowledge of software development disciplines, like OOP and AOP, to really be successful and must not be overlooked.
    - Performing verification using a different language then the design language can help reveal bugs that could have otherwise been missed. Using the design language could pose the same problem as having the designers verifying their own blocks. The same language might have implementation or behavior details that end up hiding bugs.

4.4 Simple, less complex designs

Since the design is simple, the verification environment is likely simple also. This may mean that it will be harder to realize the real power and efficiency gain that ***e*** provides in a more complex environment. The same is likely true of an advanced class based SystemVerilog environment. For this use case, either language is a good choice form a technical perspective. Therefore, you might want to let other criteria, such as existing resource experience, guide your decision.

4.5 Tapping into the Existing Ecosystem

One very attractive feature of OVM SystemVerilog is that it is supported by multiple simulator vendors. Historically, ***e*** has only been supported by Specman. However, ***e*** is an IEEE standard language and is open to be supported by any supplier. There are cases, for example on government projects, where it is required that the verification environment code itself be run on two different engines. As Cadence Specman is currently the only official ***e*** engine, it may not be the best choice in this scenario. However, there are a couple of important things to remember with respect to existing ecosystems around SystemVerilog and ***e***:

1. Do not to confuse this requirement with multi-vendor simulation support. Specman and therefore ***e*** does interoperate very well with non-Cadence simulators and has been doing it for quite some time. Refer to the "Supported Simulators" section of the Specman Integrators Guide in the Cadence Help for a compatibility list.

2. The ***e*** language is an IEEE standard.[5] While there is not yet officially another verification engine available, other companies are currently providing ***e*** based utilities like parsers, development tools, and linters.

3. Since ***e*** coding with ***e***RM, now OVM ***e***, began five years before a commercial methodology for SystemVerilog came into existence, there is a lot of silicon proven Verification IP available in ***e*** that is not yet available in SystemVerilog. As mentioned earlier the ability to reuse this existing Verification IP ecosystem can provide additional confidence that your verification environment is correct.

5 CONCLUSIONS

5.1 Pick a Strong Methodology

Both SystemVerilog and ***e*** provide a lot of power to help you achieve verification success. However, as with most powerful things, this power needs to be accompanied by the correct knowledge and methodology or it can backfire and become ineffective. It is important when choosing either language, that you accompany this choice with the proper training. It is also important to choose a verification methodology like OVM that guides you to create consistent and organized verification environments that can be more easily reused later on. In evaluating the different methodologies available, it is important to take into account the reuse capability that OVM multi-language support provides. This multi-language support gives you the freedom to most efficiently make use of existing talent within your team, pick the best language for the task at hand, and then reuse work from both languages across projects and up through full chip verification.

5.2 OVM SystemVerilog

There is no doubt that you can create a successful constrained random coverage driven verification environment using SystemVerilog. Its historical roots in design and similarity to Verilog make it an attractive option for designers responsible for both design and verification. It also provides a lot of strength in the area of assertions as SVA's can be used directly in formal verification, simulation, hardware accelerators, and even in conjunction with ***e*** testbenches. When used in conjunction with strong OOP techniques, and with a strong verification methodology like OVM, SystemVerilog is very successful in creating an advanced verification environment.

5.3 e

There are several technical advantages that make the ***e*** verification language more efficient and more reusable. When used correctly, the AOP and when subtyping capabilities provide an enormous amount of power in the form of flexibility and control. You will find this power critical to more efficiently accomplishing long term and higher levels of reuse. These features also help engineers better manage the functionality that touches the various objects in your verification environment, one of the most important of which is the testcase itself. In the area of coding efficiency, in general, it will take less code then other verification languages to capture a given task in the ***e*** language. It should stand to reason that less code can often lead to less errors, which in turn can lead to less costly debug time. These advantages result in ***e*** also being a great choice for advanced verification environments but also making it better suited for full-chip verification and higher levels of reuse across multiple projects or multiple groups.

6 ACKNOWLEDGMENTS

Special thanks to the following and many others for providing useful feedback to the early drafts of this document.

Shlomi Uziel – Cadence Design Systems
Michael Stellfox – Cadence Design Systems
Matan Vax – Cadence Design Systems
Zeev Kirshenbaum - Cadence Design Systems

7 REFERENCES

[1]Robinson, D. 2007, *Aspect-Oriented Programming with the e Verification Language: A Pragmatic Guide for Testbench Developers*. Elsevier Inc.

[2] OVMWorld web site. http://ovmworld.org

[3] ***e*** Language Reference manual

[3] IEEE Standard For SystemVerilog - Unified Hardware Design, Specification and Verification Language, IEEE Computer Society, IEEE, New York, NY, IEEE Std 1800—2005

[4] IEEE Standard for the Functional Verification Language 'e', IEEE Computer Society, IEEE, New York, NY, IEEE Std 1647—2006

[5] IEEE1647 Working Group website
http://www.eda.org/twiki/bin/view.cgi/P1647/WebHome