How to Overcome Editor Envy: Why Can’t My Editor Do *That?*

Dillan Mills, Chuck McClish
Microchip Technology Inc.
2355 W. Chandler Blvd.
Chandler, AZ 85224
dillan.mills@microchip.com
charles.mcclish@microchip.com

***Abstract*-The vast majority of programming languages today have advanced open-source editor platforms. Due to its complexity and skillset of the typical user, SystemVerilog is severely lacking in this regard. Many developers are unaware of the capabilities other language editor environments offer, while others are aware but unsure how best to implement these features or become overwhelmed with the scope of the task and abandon the project. This paper will outline the desired features of a SystemVerilog editor environment and make the case for implementing this functionality in the Language Server Protocol.**

I. Introduction

In most programming languages today, robust editor support is standard. Whether it is class hierarchy browsing, intelligent code completion, or in-code documentation, these features are all expected in a 21st century integrated development environment (IDE). In this sense, SystemVerilog is unique. Support for these features, and many others, is not standard across the editor landscape. Why is this? There are two likely reasons:

1. The Backus-Naur form (BNF) grammar of SystemVerilog is massive compared with other languages. It includes the synthesizable subset, classes, packages, assertions, coverage, user-defined protocols (UDPs), specify blocks, and more. To create a truly functional, spec compliant editor, all of this information needs to be taken into account. This is not trivial! Table I, below, shows a comparison of the SystemVerilog [1] and Python [2] specifications:

TABLE I
LANGUAGE COMPLEXITY COMPARISON

	Languages	
	SystemVerilog (1800.2017)	Python (3.9)
Number of Reserved Keywords	248	35
Number of Lines in BNF	1848	416
Number of Pages of Specification	1315	171

2. Hardware engineers specialize in hardware. Unlike software engineers, the languages we use in our day jobs are not applicable to developing software debug tools.

Another interesting feature of SystemVerilog vs other languages is the proprietary nature of the language itself. Many other languages have massive amounts of open source code available and communities that support them. SystemVerilog is almost exclusively proprietary. Because of this, many hardware developers are not open source oriented, meaning that projects that benefit the larger community do not make it out of the closed corporate bubbles where these developers work. This is not conducive to developing and maintaining open source SystemVerilog tools.

These are challenges, but they are not insurmountable. We develop the chips that run all the software on the planet. Surely, we can make a better IDE for ourselves! So, what is the best path forward?

II. Language Server Protocol

A common problem for all editors is what can be described as an M×N problem. There are ‘M’ number of languages and ‘N’ number of editors. In order to support all languages in all editors, the functionality needs to be completed M×N times. This results in a fractured editor landscape and a lot of wasted time and resources either maintaining and implementing functionality or giving up and using editors with poor feature sets. Until relatively recently, this was the common mode of operation for all editors.

Enter the Language Server Protocol (LSP). Initially published in 2016, this open source protocol is developed and maintained by Microsoft, CodeEnvy (makers of the Eclipse IDE), and RedHat. The protocol defines a set of JSON remote procedure calls between a language server and client (i.e. an editor). This changes the M×N problem to an M+N problem because creating an LSP compliant server for a language can be used on any LSP compliant editor. In addition, because each LSP server is performing the same type of tasks across various programming languages, there are now many examples of how to implement an LSP server, and software development becomes more unified.

So, the big question is: why do we care? Today, there are various solutions that attempt to provide editor support for SystemVerilog:

- Emacs, VIM, and many others have basic syntax support.
- Visual Studio Code (VS Code) and Sublime Text 3 are more context aware of what is going on, but do not fully understand the language itself.
- Eclipse has several free and paid add-ons' that provide varying levels of support.

None of these solutions are complete. In addition, utilizing these different features requires migrating the user base to different editors. Unless an engineer is forced to use a specific editor, most prefer to just keep using what they know. This results in fragmentation across a team, where some engineers utilize more advanced features that other engineers could have access to but choose not to.

By providing an LSP server for SystemVerilog, full support for all of the features of the language would be provided to all of the editors listed above and more. If you like your editor, you can keep it. This not only supports the needs of the power users, but helps developers of all skill levels, without impacting their standard work environment.

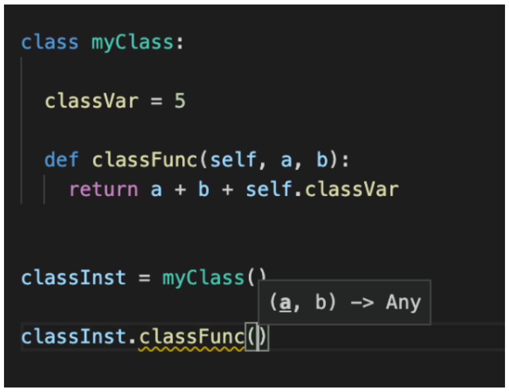
All this sounds good in theory, but how does it work? When an editor opens a file within a project, the relevant source files are evaluated by the language server. In the case of SystemVerilog, this means reading in the relevant *.f files and/or configurations. After evaluation, the language server has a full list of all the symbols, tokens, and their relationships. Inside of the editor, its LSP client can continuously ping the server for information using the context of where edits are taking place. For example, the cursor could 'hover' over a class instance. This triggers a request to be sent from the LSP client to the server asking for the properties of this class and any documentation available. Because human input through a mouse or keyboard is slow in terms of compute time, this process is transparent to the user. In essence, the system is searching for answers to the questions you have not asked yet.

III. Out of the Box Features

So, let's say an LSP server for SystemVerilog was created, what types of basic features would an LSP server provide to an editor out of the box? Since many languages have common traits with SystemVerilog, it is relatively straight forward to model what capabilities an LSP client would contain by using another language as an example. In the examples below, Python is used to model the types of features that would be available in both systems, while mockups are used in places where Python doesn't make sense.

A. Symbol Lookup

This is the most basic (and one of the most useful) functions an LSP server should provide. All code is composed of tokens (reserved keywords, syntax) and symbols (everything else). The first questions any developer has when they open up an existing source file or are debugging their own, are related to the symbols: *What is the definition of this module? Where is this signal referenced? Where is this class declared?* Because the LSP server understands all of the symbols in the code base and their relationships to one another, it is a trivial matter to send a query from the client to request this information as seen in Figure 1 below:

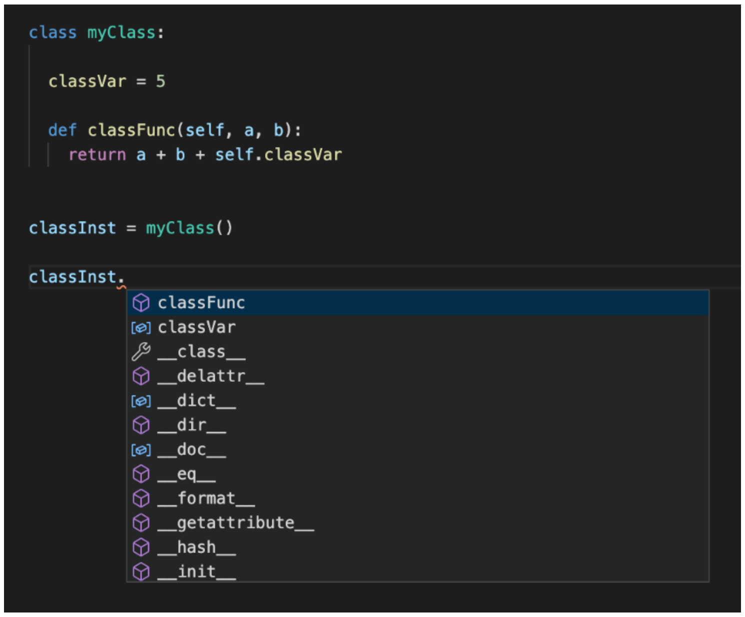


```
class myClass:

  classVar = 5

  def classFunc(self, a, b):
    return a + b + self.classVar


classInst = myClass()
                       (a, b) -> Any
classInst.classFunc()
```

Figure 1. Example of Symbol Lookup (Function Parameters)

B. *Intelligent Code Completion*

The first example of code completion can be seen in Figure 2 below:



```
class myClass:

  classVar = 5

  def classFunc(self, a, b):
    return a + b + self.classVar


classInst = myClass()

classInst.
```

Figure 2. Example of Symbol Lookup (Functions, Variables)

In this example, a symbol is starting to be typed. After each typed character, the editor pings the language server for any symbols *relative to the context* containing this partial match. The context sensitivity here is very important as several editors have native code completion, but because they are not communicating with a full language server, they typically regex the entire project looking for matches. This can lead to false positives in terms of completion causing subtle and difficult to debug issues.

C. *Macro Expansion and 'Peeking'*

Another example of code completion is macro expansion (also known as macro peeking). On mouse or cursor hover in the editor, the true text of a macro can be seen without having to dig through the code to find the original definition. In addition, it is possible to use the editor to replace an instance of a macro with the true text.

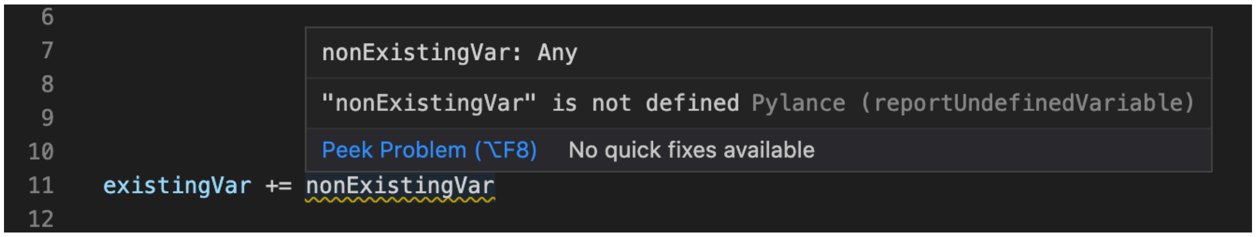
Macros are very useful to abstracting away low level details and making flexible code that can be modified at compile time, however, they can lead to dense, confusing code that requires a lot of back and forth between the code being developed and the code that contains the macro definitions. This feature provides immediate feedback to the developer without having to walk through the code base looking for definitions.

D. *Quick Jumping from Log Files or Built-in Command Prompt*

After compiling or running a simulation, if you have messages pointing to a specific line of code, this would let you quickly jump to that line. This would work from a logfile, or from the command prompt built into some compliant editors. Additionally, with the macro support, this would allow you to jump to the true line of origin, instead of the pre-macro-expanded line.

E. *On-the-Fly Linting*

On-the-fly linting provides immediate feedback as you are writing your code. The linting will check for any issues that would result in a compiler warning error and will indicate it by highlighting the problem code. This effectively removes the churn of "write code, compile, fix errors, repeat" by indicating what compile errors are present in-line in near-real time. Additionally, many editors such as VS Code include a "Problems" tab that lets you quickly jump to any warnings or errors, which further reduces debug time.



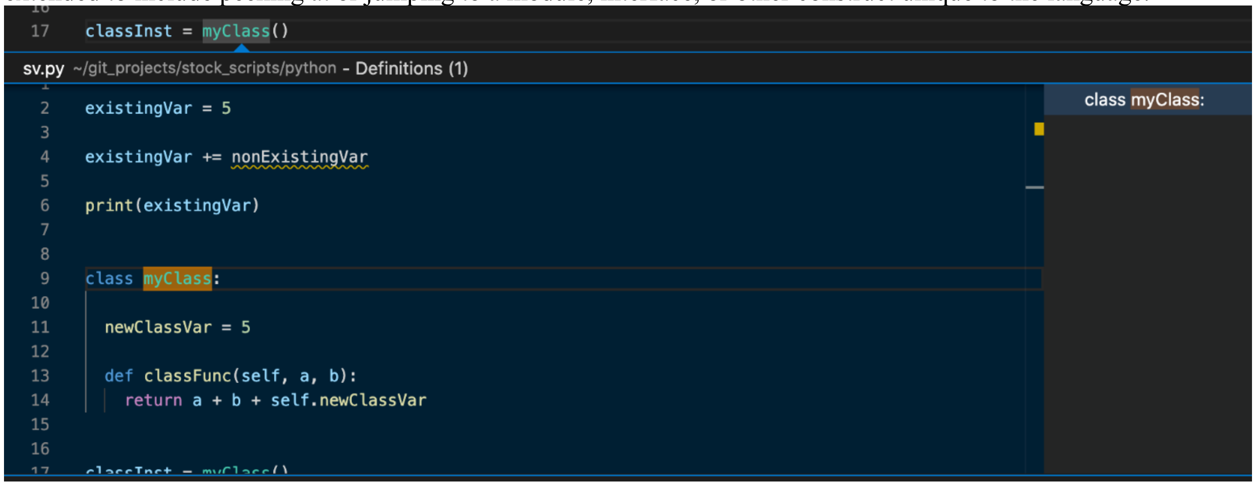
```
nonExistingVar: Any
"nonExistingVar" is not defined Pylance (reportUndefinedVariable)
```

```
existingVar += nonExistingVar
```

Figure 3. Example of LSP Linting

F. *Hierarchy References*

In standard object-oriented programming languages, hierarchy references allow you to quickly peek at or jump to a source class from an object, or to a parent class from a child class. In a SystemVerilog implementation, this can be extended to include peeking at or jumping to a module, interface, or other construct unique to the language.



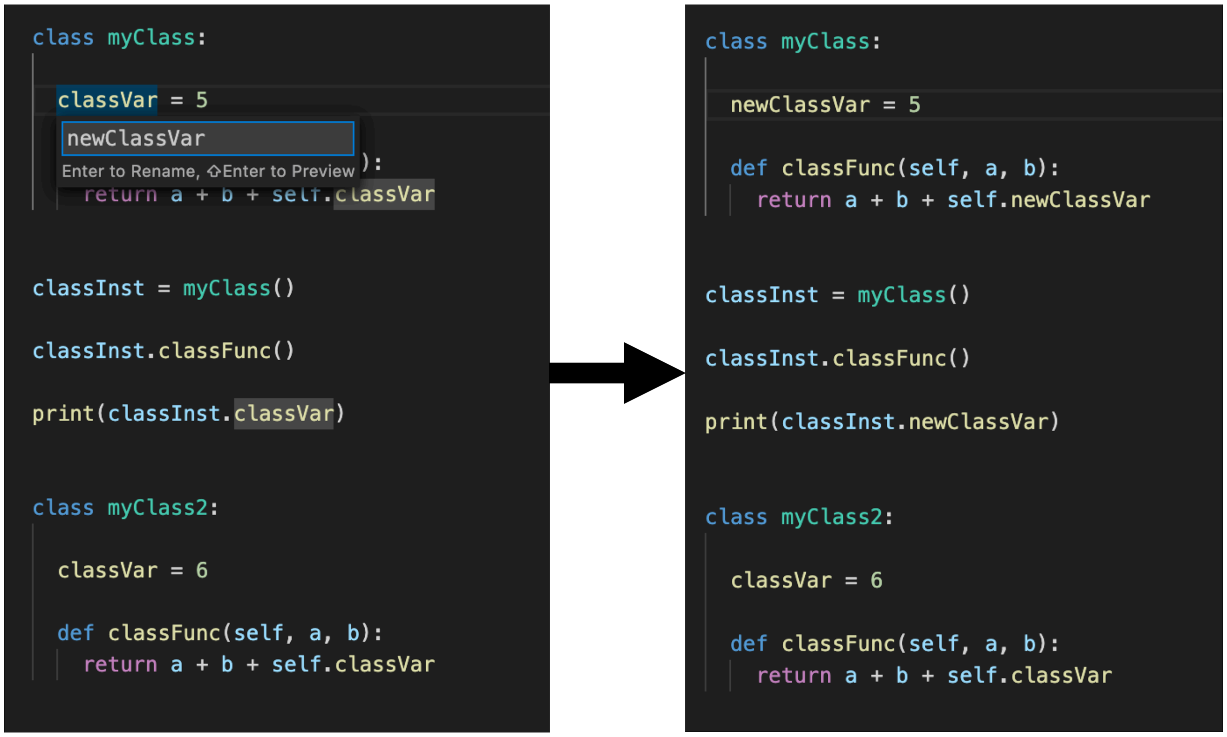
```
classInst = myClass()
```

```
existingVar = 5

existingVar += nonExistingVar

print(existingVar)


class myClass:

  newClassVar = 5

  def classFunc(self, a, b):
    return a + b + self.newClassVar


classInst = myClass()
```

Figure 4. Example of LSP Hierarchy Peeking

G. *Refactoring*

Refactoring capabilities in an editor make it easy to reformat code in various ways. One common way included in many LSP instances is the ability to rename a variable *in context*. The tool will find every reference to the specific variable and rename them all at once. Note that this is smarter than a simple "find keyword/replace keyword" operation. The editor will be able to intelligently determine if the keyword found is related to the original variable you selected for refactoring, or if it is a variable with the same name in a different scope. This refactoring works with variables, functions, classes, and in a SystemVerilog LSP, can additionally work with signals, modules, ports, and any other named component of your design and testbench. It would intelligently handle updating a netlist of a port name changed to make sure your design doesn't inadvertently break.



```
class myClass:

  classVar = 5
  newClassVar
  Enter to Rename, ⇧Enter to Preview ):
    return a + b + self.classVar


classInst = myClass()

classInst.classFunc()

print(classInst.classVar)


class myClass2:

  classVar = 6

  def classFunc(self, a, b):
    return a + b + self.classVar
```

```
class myClass:

  newClassVar = 5

  def classFunc(self, a, b):
    return a + b + self.newClassVar


classInst = myClass()

classInst.classFunc()

print(classInst.newClassVar)


class myClass2:

  classVar = 6

  def classFunc(self, a, b):
    return a + b + self.classVar
```

Figure 5. Example of LSP Context-Aware Refactoring

H. Standardized Auto-indent

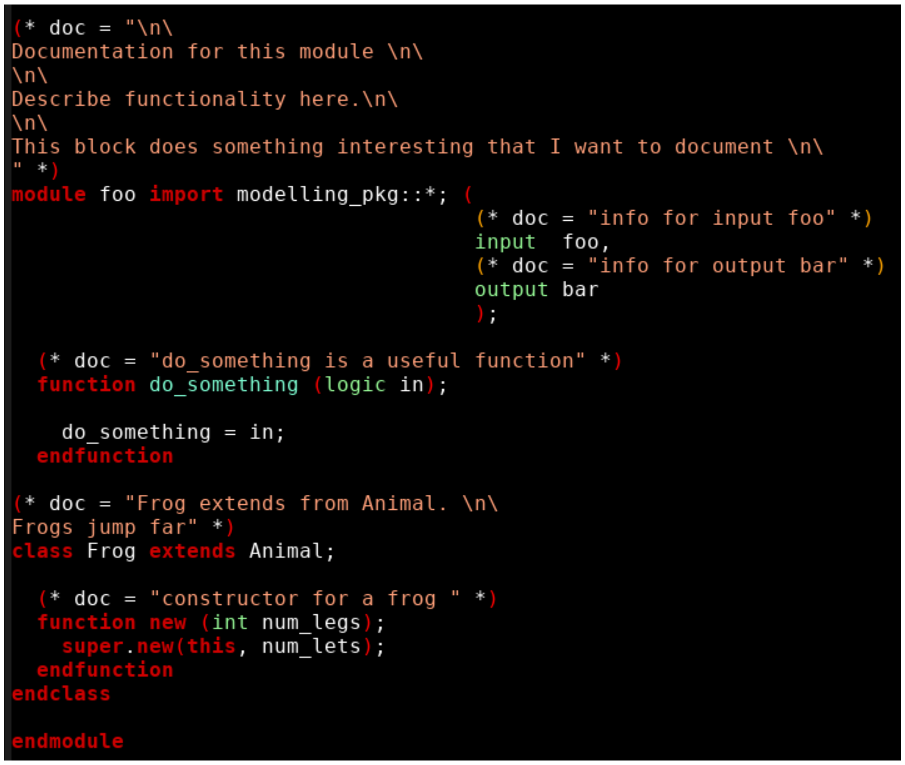
Every editor seems to have their own style for indenting code, and many editors miss some less-common aspects of the language, resulting in code that either requires lots of maintenance to keep formatted, or ends up difficult to read. Additionally, code shared between a team can become a train-wreck of indentation, as each engineer applies their own tool-specific formatting after opening a file. By centralizing the indentation to the language server, this problem goes away. Everybody on your team will indent files exactly the same, and the language server will make sure all your code gets indented properly. Ideally, this indentation could be controlled so a team could customize the indentation features to their liking.

IV. SystemVerilog Specific Features

I. In-line Documentation

One notable feature missing from the SystemVerilog language itself is in-line documentation. Languages like Python or Common Lisp contain built-in docstrings that are easily parsed by an LSP server and returned to a client on cursor hover. This is an extremely useful language feature. In the SystemVerilog UVM library, documentation is provided utilizing the NaturalDocs format. While this is better than nothing, it is not a component of the language itself and is not mentioned in any of the relevant language standards. This makes it difficult to develop a documentation parsing engine in the LSP server itself. So that raises the question, what mechanisms do we have in SystemVerilog that can be utilized for in-line documentation?

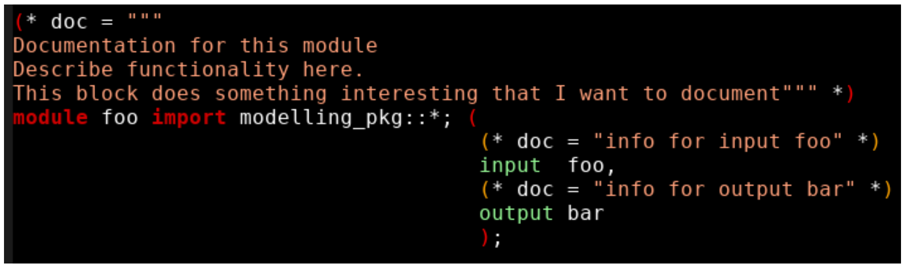
Enter SystemVerilog attributes. Attributes can be applied to nearly all the relevant components of the SystemVerilog language: module definitions and instances, signal declarations, class definitions and objects, task/function definitions, etc. Attributes themselves are simply a key-value pair. With a string value being the documentation itself, all that is required is a common key to indicate that the string provided is documentation and not something else. This paper proposes that a special key ‘doc’ be utilized to indicate a string is documentation. From this, an LSP server will easily be able to attach the relevant documentation to the relevant code snippet. In practice, the code would look like Figure 6 below:



```
(* doc = "\n\
Documentation for this module \n\
\n\
Describe functionality here.\n\
\n\
This block does something interesting that I want to document \n\
" *)
module foo import modelling_pkg::*; (
                                     (* doc = "info for input foo" *)
                                     input  foo,
                                     (* doc = "info for output bar" *)
                                     output bar
                                     );

  (* doc = "do_something is a useful function" *)
  function do_something (logic in);

    do_something = in;
  endfunction

(* doc = "Frog extends from Animal. \n\
Frogs jump far" *)
class Frog extends Animal;

  (* doc = "constructor for a frog " *)
  function new (int num_legs);
    super.new(this, num_lets);
  endfunction
endclass

endmodule
```

Figure 6. Documentation using attributes, today's SV standard

This works, but multiline comments are very cumbersome without a multiline string operator. To make inline documentation easier, this paper proposes that the SystemVerilog standard be amended to include the """ operator to create multiline strings/comments. The resulting example code would look like Figure 7 below:



```
(* doc = """
Documentation for this module
Describe functionality here.
This block does something interesting that I want to document""" *)
module foo import modelling_pkg::*; (
                                     (* doc = "info for input foo" *)
                                     input  foo,
                                     (* doc = "info for output bar" *)
                                     output bar
                                     );
```

Figure 7. Documentation using attributes using multi-line string operator

J. Signal Tracing

SystemVerilog, being an HDL, describes physical logic and wires. As such, there is the concept of drivers and receivers on a net or variable. This is different from software languages. In graphical simulation debug tools, there is a concept of driver/receiver tracing. With an LSP server, this functionality can be provided within editors as well. The LSP is flexible in that additional, language specific functionality can be supported. This does require modifications to the client to support, but this feature would be extremely valuable. No longer would a developer need to compile and simulate a design just to trace signals around a design.

K. Premium Mixed Language Support

In addition to SystemVerilog support, there are additional LSP servers that could be developed. Several possibilities that quickly come to mind:

- UPF LSP server: power awareness in an editor!

- VHDL LSP and SystemC LSP servers that link to a running SV LSP server to enable multi-HDL language support
- Portable Stimulus Standard LSP server
- Standard Delay Format LSP and Liberty Model LSP servers: timing awareness in an editor!
- SPICE LSP server to provide mixed signal awareness in editor
- Tool specific TCL LSP servers to support backend synthesis and APR tools

All of these features (and more) could be premium add-on tools that vendors could sell. Once the developer community is used to developing with LSP capable editors, this opens up an entirely new market for EDA vendors to develop products. This melds well with the current mantra of *collaborate on standards, compete on products*.

L. Testbench Builder

Many vendors already have a testbench builder application. We are not suggesting reinventing the wheel here, but developers hate leaving their environment and context switching to run a different tool. We propose the vendors could add hooks to their testbench builders so they can be run from within an LSP environment.

M. Debugger Support

Editors such as VS Code have a robust debugger for stepping through and debugging simulations. The interactive command-line simulations from the vendors include support for stepping through code as well, and these features are available in the GUI interactive simulations. We propose adding support for the LSP debuggers so you can step through a simulation and view registers or other allocated variables without needing to launch a full graphical debug environment and context switch. This will likely be more practical for verification testbenches, since stepping through the many parallel processes of a design could be overwhelming. Additionally, it would be nice if the simulation stayed coupled with the GUI window so you could quickly jump back to the waves if needed. Essentially, the goal would be to replace the simulation GUI code viewer with your LSP-compliant editor, but still keep the simulation GUI features available.

N. Assertion Writer

Being a language unto itself, SystemVerilog Assertions can be difficult to master and understand. In particular, writing assertions from scratch can be somewhat difficult. With an LSP that understands SVA, it opens up a lot of possibilities for the developer. For example, having context aware code completion available to show valid tokens or symbols for a particular location would be very helpful. In addition, flagging easily identifiable mistakes could also be very useful. For example, flagging assertions that would spawn many hundreds of threads with a warning could be useful.

O. Coverpoint Expander

Given a bus signal, have the option to generate a few pre-defined cover bin sets, such as:

- Each bit
- Evenly divided bins
- Each bit magnitude group (4-7, 8-15, 16-31, etc.)
- Others that can be configured through some form of template

P. Context-aware Linting

In addition to normal on the fly *syntax* linting, it would be very beneficial to have *context-aware* linting. This type of linting would flag legal SystemVerilog code but constructed in such a way that would result in issues for simulation or synthesis.

- Randomization issues or constraints that are impossible to meet
- Forever loops that would halt the simulator
- Case statements in always blocks that result in the creation of latches at synthesis
- Flip flops with async set/reset pins that are coded incorrectly

V. Conclusion

A SystemVerilog LSP server would be a great addition to the hardware developer's toolkit. In addition, the more this language goes from a hardware design language to a verification language, tools like this are no longer nice to have, but a necessity. Accellera is in a unique position to facilitate the development of such a tool. This organization has shown time and again that it can leverage expertise across a very closed-off industry and bring state of the art standards, libraries, and workflows to fruition. For the benefit of all hardware design and verification engineers, the authors of this paper strongly suggest that a committee be formed (or become part of the responsibilities of an existing committee) to develop a SystemVerilog LSP server.

VI. References

[1] IEEE Std 1800-2017, *Standard for SystemVerilog – Unified Hardware Design, Specification, and Verification Language.*

[2] "The Python Language Reference," [Online]. Available: https://docs.python.org/3/reference/index.html.