Visualizing SystemVerilog and UVM

Jamie Ridgeway, Paradigm Works, Inc., Fort Collins, CO, USA
(Jamie.Ridgeway@paradigm-works.com)

***Abstract*-Conveying the implementation of developed code in a verification project is an especially fraught task. That is, as a newcomer to an existing project, available documents are often skimmed or disregarded in favor of just stepping through the code line-by-line. This is primarily because documentation has no hard value to the project compared to bug hunting. Even employing existing tools to ease the documentation burden, such as NaturalDocs and Doxygen, testbench documentation is nearly universally incomplete or out of date versus the actual implementation. Furthermore, these tools do not specifically address the predominate verification domain language, SystemVerilog, and technique, the universal verification methodology (UVM). We introduce three free engineer-developed tools, svdiag.pl, svpre.pl, and uvmtopo.pl, that can aid the in exploration and understanding of existing developed code as well as provide coherent diagrams of actual implementation for documentation.**

I. Introduction

Conveying a verification engineer's intent of their implemented test code is especially fraught. This is because any such document of the testbench implementation, or individual tests' implementation, has no real value. This is indicative in the project schedule affording no time for viable testbench documentation. The actual testing of the hardware design under verification is far more important than the documentation of its test implementation. As such, verification testbench documentation is nearly universally incomplete or out of data versus its implementation.

We do not imply that no documentation exists for verification in a hardware design project. It is imperative to itemize features from the hardware specification that must be tested to release the design to production, be that into an ASIC, an FPGA, or any number of other production target platforms. This list of features is the *verification test plan* and is presented in plain language in some document and/or spreadsheet. The verification team breaks down the test plan into a *feature coverage plan* and *functional coverage plan*. Some features may be only tested via standalone directed tests reporting pass or fail. Others can be covered within a constrained-random testbench employing functional coverage to report on features exercised. The collection of directed tests and functional coverage points presented in plain language, in some document, usually a spreadsheet as the feature coverage plan guides overall verification management. Electronic design automation (EDA) vendors assist this task with their *verification plan management toolsets*, which allows the plain language plans to tie symbolically to directed test names or functional coverage and constrained-random test names. Together, the engineer developed verification test plan and EDA vendors' runtime reporting capabilities during test regression, are used to show when verification is complete. None of the verification plans and reporting structures properly document the implementation of the testbench or individual tests. Of all these documents, the engineer-facing testbench documentation is often the least valuable.

We consider the following cases to be a complete set[1] of events that require a viable testbench documentation specification:

1. The customer will integrate the IP-level testbench directly in their own testbench,
2. The testbench is developed as general purpose to be reused in design-specific projects,
3. The already produced design has shown to have bugs requiring testbench verification, or
4. An engineer is added to an existing verification project.

The testbench document becomes a deliverable to the external customer in (1), thus conveying it real value. The testbench, in (2) is general purpose to be used by disparate teams. Documentation becomes a deliverable, albeit internal. When a production design has bugs, hopefully a member of the original verification team can aid in testing the design again, otherwise this becomes a number (4) event: someone unconnected to the project is added to the project. In this case, viable documentation, if available, should lead to faster productive outcomes for that engineer.

In this paper, we describe several engineer-developed and free tools written in Perl: svpre.pl, svdiag.pl and uvmtopo.pl [1]. These tools are presented as aids in understanding existing code within a testbench as well as for use for testbench documentation. These tools were developed out of necessity because viable testbench documentation often does not exist. The svpre.pl tool is a general purpose SystemVerilog pre-processor meant to declutter the various

[1] "Complete" should be considered somewhat farcical – our point is that it is extremely rare that testbench documentation has real value to a verification project versus DUT bug hunting.

`include, `ifdef and `define instances that exist in the code base [2]. The svdiag.pl tool, converts SystemVerilog class declarations to diagrams styled like those from unified modeling language (UML) [3], though not truly UML, via the GraphViz dot program [4]. Finally, the uvmtopo.pl repurposes svdiag.pl to be a visualization tool of the actual runtime universal verification methodology (UVM) testbench [5], as reported in the simulation log.

II. Related Work

Code documentation tools do already exist that may beautifully present information. Two examples are NaturalDocs [6] and Doxygen [7]. They both can generate HTML documentation that includes class structure information as well as description of implementation. Their primary caveat is that any meaningful description is incumbent upon the verification engineer documenting their code, in-line, such that it conforms to the documentation tool's descriptive grammar. Consider the example a very basic AXI protocol sequence item presented in Code 1.

```
1   class axi4_seq_item extends uvm_sequence_item;
2     `uvm_object_utils(axi4_seq_item)
3
4     // Variable: addr
5     // Address targeted for reading or writing.
6     rand bit[31:0] addr, data;
7
8     // Function: do_print
9     // UVM compatible printing.
10    extern virtual function void do_print(uvm_printer printer);
11    extern virtual function string convert2string();
12
13  endclass
```

Code 1: NaturalDocs usage for an example AXI4 Sequence Item.

Here, the sequence item has only four class members: two random 32-bit variables, `addr` and `data`, and two printing functions. The in-line comment format might be familiar as NaturalDocs is the preferred documentation tool of the UVM library [5].[2] Refer to Figure 1 for NaturalDocs generated web page for the code as per the in-line comments.

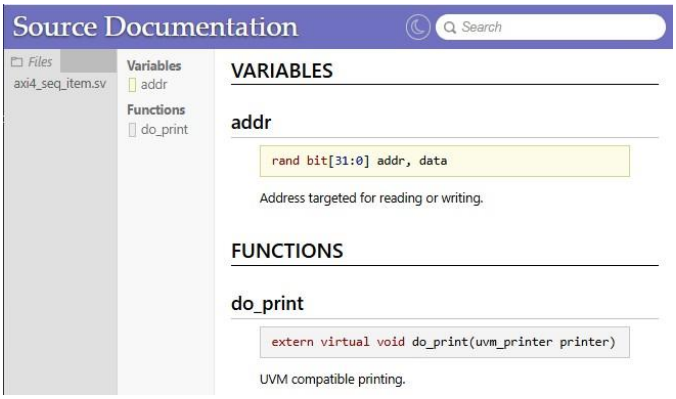


Figure 1: NaturalDocs documentation for axi4_seq_item class in Code 1.

Clearly the generated documentation for the example `axi4_seq_item` is lacking. Consider a hardware IP customer who is incorporating the testbench into their own for system-level testing. Is there anything in Figure 1 that would prove useful for a customer when using the `axi4_seq_item` class? Or does this presentation provide anything beyond what the code, itself, provides? We argue, no.

Doxygen is another widely available documentation tool. However, like NaturalDocs, it suffers from similar problems: using the code comments, themselves, to generate the descriptions for each class member, though its grammar is more robust. A distinct advantage that Doxygen offers is its capability to generate visual diagrams utilizing GraphViz's dot program. Figure 2 is an example of a Doxygen/dot generated inheritance diagram for Code 1. This

[2] The UVM library employs an extended version of NaturalDocs with broad SystemVerilog support. As far as we are aware, this extension is not available in any NaturalDocs version publicly available.

visual representation, backed up with textual descriptions of each class, targets two types of learning: visual and read/writing, as part of the visual-aural-reading/writing/kinesthetic (VARK) learning models [8]. We see the visual representation as key to quickly understanding testbench architecture, and it is the primary style for our toolset.

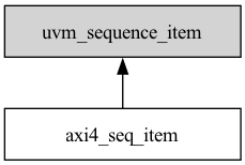


Figure 2: Doxygen generated inheritance diagram for axi4_seq_item class in Code 1.

Both documentation tools, NaturalDocs and Doxygen, also have some practical concerns. Their goal is the generation of *comprehensive* documentation for the whole project. To properly deploy a documentation tool, a dedicated resource must understand its capabilities and how they may best be used for the verification project. Essentially, this is a learning curve for the engineer tasked with generating the documentation. Furthermore, verification engineers have their own learning curve in the using the documentation tool's grammar within code comments. As we have shown in the code example Code 1, it is easy to simply skip the documentation step, whether that be in-line code comments or a separate document. In contrast, our toolset parses the SystemVerilog in Code 1 and generates a diagram representation, as in Figure 3.

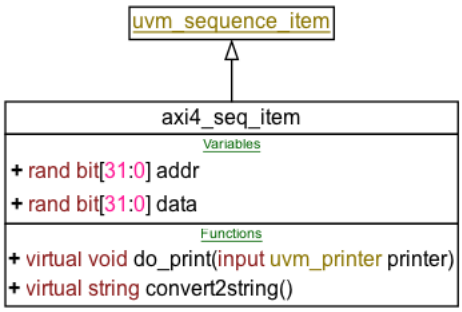


Figure 3: svdiag.pl and dot generated diagram of Code 1.

Finally, installation of either Doxygen or Naturaldocs may come with its own set of caveats. Does the tool support my operation system? Does the tool require installation of additional packages to operate? Will the IT department be necessary to properly install this tool? Is there licensing fee for the tool? Does the usage license allow the team to publish, internally or externally, the documentation without additional steps or fees?

Our focus differs from these full-featured documentation toolsets in that we only consider the visualization of code and its relationships. We want the verification engineer to use diagrams to better understand existing code or their own implementations, and as a vehicle for conveying information, written – a testbench document – or in discussion with others. The diagrams are meant to be easy to generate and, thus, may be temporary. But the information they convey is more likely to persist for a verification engineer tasked for environment development or maintanence.

III. SystemVerilog Visualization Toolset

While each widely available documentation tool can generate viable and coherent documentation, given the proper environment (management acceptance, dedicated resources, and team adoption), we had a very limited set of goals for our documentation tool.

1. Easy to install and use,
2. Parse native SystemVerilog code, including compile-time macros, to
3. Generate diagrams that are reminiscent of UM [9]L-Style,
4. Generate diagrams for all code to show class relationships, and
5. Only focus on diagrams rather than comprehensive documentation.

Having a tool that was easy to install – doesn't require IT or management assistance – was particularly important. For our toolset we opted for generic PERL that is dependent only on core PERL package modules generally available in any PERL distribution. Currently, the only PERL packages required to employ our toolset are:

- Getopt::Long,
- FileHandle,
- FindBin.

Additionally, the only package required to create the visualization is:

- GraphViz dot.

Our toolset translates from SystemVerilog to GraphViz DOT language, specifically for directed acyclic graphs, digraphs [4]. The dot program is commonly available on most Linux distributions and, if it is not, is free and easy to install on an engineer's Microsoft Windows© or Apple MacBook© laptops. The dot program, itself, satisfies points (3) and (4) by converting the tools' generated dot language files to diagrams.

A. svdiag.pl

The primary diagram tool, svdiag.pl, is essentially a "lite" cross-compiler from declarative SystemVerilog to GraphViz DOT language. The dot program converts svdiag.pl output to UML-style diagrams, as in Figure 3.

We employ several features in the diagram for a viewer to quickly understand the class or classes. First, inheritance is indicated by the outlined arrow pointing from the derived class to the base class. The class name is indicated clearly at the top and in its own row of the table representing the class. Class members are grouped together, such as Variables and Functions, with the contents of each group appearing in coding order (Tasks appear separately). In Figure 3 functions `do_print()` and `convert2string()`, for example, are not in alphabetical order in the diagram but as the engineer coded them in Code 1. All members of this class have a public access level and are indicated with a preceding plus sign (+). Protected members are shown with a hash mark (#) and private with a dash (-). SystemVerilog keywords are indicated in red while known UVM key classes are indicated in gold, and string and number literals are highlighted in pink. When the inheritance tree reaches a base class that is unknown, i.e., not processed by svdiag.pl, it is shown underlined. For example, `uvm_sequence_item` is underlined because svdiag.pl did not process this class and thus it is considered unknown, but also gold because svdiag.pl recognizes it as a UVM library key class.

1) Parse Methodology

Our lite cross-compiler tool focuses on declarative features in SystemVerilog Part One: Design and Verification Constructs. No static or runtime analysis of functional code is performed. Instead, only static analysis of structural verification code is processed, ignoring all others. This is most clearly seen from Code 1. The declaration of the `axi4_seq_item` class is sufficient to generate the diagram in Figure 3. Furthermore, svdiag.pl ignores all compiler directives, except for file inclusion. For example, no information from `` `uvm_object_utils() `` is represented in the diagram above. The methods `do_print()` and `convert2string()` are part of the SystemVerilog grammar, Annex A.1.9, as prototypes [2]:

```
class_method ::= extern { method_qualifier } method_prototype ; | …

method_prototype ::= task_prototype | function_prototype

function_prototype ::=
  function data_type_or_void function_identifier [ ( [ tf_port_list ] ) ]
```

In the grammar for SystemVerilog, rule names are on the left-hand side of `::=` and they are composed of the rules indicated on the right-hand side. For example, for any class member task or function, the syntax must adhere to the `class_method` rule, which is in turn composed of further rules in the grammar. The pipe token `|` separates disparate options for the rule. In the `class_method` rule, above, this rule option for method prototype declarations, must have the `extern` keyword followed by a `method_prototype` rule and ending with a semicolon token. Both the keyword and token are shown in `red`. The `method_qualifier` rule is optional here, braces `{ }` indicate rules that may be skipped or repeated, and is defined as any of the following keywords, according to the grammar rules:

```
method_qualifier ::= [ pure ] virtual | class_item_qualifier

class_item_qualifier ::= static | protected | local
```

The brackets `[ ]` indicate a rule that is optional, but, unlike braces, cannot not be repeated. For example, the `pure` keyword is optional but may not be repeated. All SystemVerilog syntax represented by the grammar above is parsed by svdiag.pl and, other than `extern`, displayed in the resulting diagram.

It is also common to fully define the task or function body in place within the class declaration (i.e., between `class` and `endclass`). The grammar allows for this with the following rules (showing function below, refer to the grammar for tasks) :

```
class_method ::= { method_qualifier } task_declaration |
                 { method_qualifier } function_declaration

function_declaration ::= function [ lifetime ] function_body_declaration

function_body_declaration ::= … |
    function_data_type_or_implicit [ … ] function_identifier ( [ tf_port_list ] ) ;
        { block_item_declaration } { function_statement_or_null }
        endfunction [ : function_identifier ]

lifetime ::= static | automatic
```

Here, svdiag.pl parses the function identifier, return type, optional lifetime attribute, and port listing, if one exists. However, the parser ignores any `block_item_declaration` and `function_statement_or_null`, instead scanning for `endfunction`. In this manner, ignoring runtime executable code in any construct, we have the parser focus only on declarative statements within SystemVerilog.

From the parsed code, svdiag.pl builds an internal model consisting of class structures with their declarative contents and represented as a set of PERL5 objects (i.e., blessed hashes). A class structure object is its own namespace. The script ensures unique symbol naming within that namespace, be it a function, task, or a class member variable. All classes are afforded their own namespace, even it if is not declared within the parsed code. Referring to Figure 3, both classes are represented by instances of the same class structure object in the internal model. An attribute in the class structure identifies if the instance is fully defined or considered an orphan. That is, the class name identifier was parsed but never declared, as seen in line 1 of Code 1. However, that same orphan structure can be fully defined if the class declaration is parsed at any time. For example, if the following code was presented to svdiag.pl:

```
`include "axi4_seq_item.svh"
`include "uvm_sequence_item.svh"
```

The `` `include `` is always supported by svdiag.pl. In the first line, parsing the code in `axi4_seq_item.svh` would result in two unique classes in the internal model, `axi4_seq_item` and `uvm_sequence_item`, with the latter marked as an orphan. In the second line, the `uvm_sequence_item` class is fully declared, and the internal model orphan is updated to a regular class instance in the model.

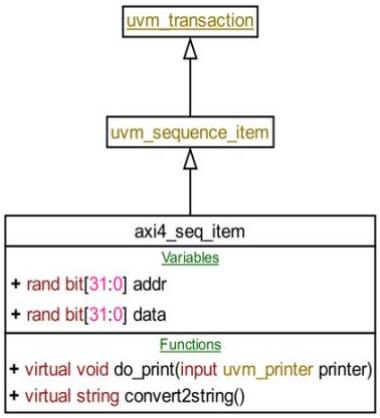


Figure 4: Both `axi4_seq_item` and `uvm_sequence_item` are known but `uvm_transaction` is an orphan class.

Now, in Figure 4, `uvm_sequence_item` class name is not underlined, indicating it is not an orphan class. However, it does extend from `uvm_transaction`, which is now displayed as an orphan class. The svdiag.pl script by default displays the contents of all fully declared classes. In Figure 4, we have employed a svdiag.pl command-line option to show the contents of the class in question, while indicating only the class names for its known lineage.

2) Translation Methodology

We take three avenues to translation and output for the svdiag.pl parsed internal model: code generation, display options, and diagram physical size.

First, each class structure, and each component within, can represent itself in the DOT language. DOT provides both overall diagram structure information, represented by the arrows between objects, as well different ways to format and label within each object. In our diagrams, each class is a box containing nested HTML tables, while each class item is represented as one row in a table. For an example of the DOT language representation of `axi4_seq_item` refer to Code 2, below.

```
1  // Take class: axi4_seq_item
2  // Take ancestor: uvm_sequence_item
3  // Take orphan ancestor: uvm_transaction
4  c4 [ label = <<table title="uvm_transaction" … </table>> ];
5  c92 [ label = <<table title="axi4_seq_item" port="P">
6    // Title row/cell table
7    <tr><td>
8      <table><tr><td>axi4_seq_item</td></tr></table>
9    </td></tr>
10   // Variables row/cell table
11   <tr><td><table>
12     <tr><td>Variables</td></tr>
13     <tr><td port="v93">+ rand bit&#91;31:0&#93; addr </td></tr>
14     <tr><td port="v94">+ rand bit&#91;31:0&#93; data </td></tr>
15   </table></td></tr>
16   // Functions row/cell table
17   <tr><td><table>
18     <tr><td>Functions</td></tr>
19     <tr><td>
20       + virtual void do_print(input uvm_printer printer)
21     </td></tr>
22     <tr><td>+ virtual string convert2string()</td></tr>
23   </table></td></tr>
24 </table>> ];
25 c3 [ label = <<table title="uvm_sequence_item" … </table>> ];
26 c3:P -> c92:P [dir=back arrowtail=empty];
27 c4:P -> c3:P [dir=back arrowtail=empty];
```

Code 2: DOT translation snippet for `axi4_seq_item` class.

The svdiag.pl internal model takes care to uniquely identify each class object with a label, `c92` for `axi4_seq_item` class in Code 2. By default, the dot program may layout arrows wherever convenient. However, here we enforce a particular location on the object for the arrow tail or head with a port label. Both the outer table `P`, in line 5, as well as individual cell member details (`td`), `v93` in line 13 and `v94` in line 14 have labels. Connections between objects and considering their ports are shown in lines 26 and 27 (refer to uvmtopo.pl, section IV.A., to see how variable ports are used). Note that while the color formatting is shown in Code 2 as it is displayed in Figure 4, additional HTML tags are required in the DOT language file. Refer to Appendix section VIII.A. for full DOT file generated svdiag.pl and used for Figure 4.

The script also generates the DOT code to encapsulate the various class objects and arrows between. The target diagram is directed acyclic graph, or diagraph. While each PERL class structure, and each member within, generates the DOT code to represent themselves, the top-level svdiag.pl object generates the surrounding DOT code for a well-formed digraph.

```
1  digraph G {
2    node [fontname="arial", fontsize=10, shape="plaintext" ];
3    // optional print-size
```

```
4     nodesep=0.3;
5     ranksep=0.4
6     // class objects
7     // connections/arrows
8   }
```

Code 3: Outer digraph code generation template.

Second, line formatting is layered on top of code generation to allow for centralized control. The line formatter handles both SystemVerilog keyword and UVM key class coloring. Also, by default, each table class member is output in a single table row in the diagram. For methods, especially, this can result in very long lines of text as each method argument's direction, input, output, or reference, and optional default value is displayed in the diagram. The width of the generated box, as per the number of characters displayed in one row before a line break, is controlled is indicated in Table I.

Table I: svdiag.pl display options for each class member.

Command-line option	Result	Default
`-max-chars <num>`	Sets the number of characters on one line	unlimited
`-nocolor`	Disable SV and UVM keyword coloring	enabled

Third, and finally, svdiag.pl considers the overall generated diagram size. By default, the generated diagram is expected to be examined only digitally – via a PDF or picture viewer. This is desired as it is possible to zoom in to understand to a potentially large and complex diagram. In this case, line 3 in Code 3 is empty – no print size information is provided or necessary. Some preset sizes are built in to allow for constraining the resulting diagram to poster, US letter, US legal, and A4; as well as free-form indicating width and length on the command-line, Table II.

Table II: svdiag.pl options for physical diagram size.

Command-line option	Diagram size	Generated code (default)
`-page [landscape_]poster[,fill|compress|auto]`	36 in. x 48 in.	`size="35,47"; ratio=fill;`
`-page [landscape_]letter[,fill|compress|auto]`	8.5 in. x 11 in.	`size="7.5,10"; ratio=fill;`
`-page [landscape_]A4[,fill|compress|auto]`	8.3 in. x 11.7 in.	`size="7.3,10.7";ratio=fill;`
`-page <width>,<length>[,fill|compress|auto]`	W+1 in. x L+1 in.	`size="W,L"; ratio=fill;`

With the `landscape_` prefix for the print size, the width and length values are rotated – a landscape orientation. The attributes, `fill`, `compress`, and `auto`, determine how compact the resulting diagram is on the page. When indicating the fill attribute for a small diagram, such as Figure 4, the class boxes would be laid-out equidistance top-to-bottom on the page with long arrows in-between, refer to Appendix section VIII.B. However, the layout space is quickly filled as more classes are parsed and displayed.

3) Use Models

The svdiag.pl script has two primary purposes: generate clean readable diagrams for documentation and explore the codebase to obtain a better understanding. When targeting documentation, it is best to only parse the code that should be shown. This provides the most control over how the diagram is generated. If the dot program is available on your installation, then svdiag.pl can call it directly to generate diagrams in PDF, PNG, or JPG format. These, in turn, can either be directly imported into a document or a screen snippet taken for importation. In all cases, the accompanying `.dot` file is generated which can be edited directly, if necessary (such as removing the date from the bottom of the diagram). The svdiag.pl always shows the dot command to generate the target output diagram file.

However, when seeking obtain a better understanding of an existing codebase parsing as much code as possible in is preferred. For example, the UVM library may be pre-processed and parsed, wholesale, and diagram generated.[3] Of course, the display area is overwhelmed with classes and inheritance trees. A commercial PDF viewer is recommended to explore the generated codebase diagram by zooming into the diagram to find pertinent classes and their relationships. The program evince is often already installed on Linux distributions [9]. However, as the diagram increases in complexity and size, we have seen that evince cannot zoom in well enough as commercial PDF viewerd

[3] Using the pre-processor is necessary as macros are used in declarative structural code throughout the library. Note that this process can take 8 seconds or more, depending on your machine.

to see the necessary detail. Nonetheless, a generated PDF and viewer of choice is the most straightforward way to see the entire codebase at one time. For example, the following commands can pre-process, parse, and generate a PDF of the entire UVM codebase.

```
svpre.pl -nouvm-macro-inst -macro-inst -keep-ws-read -o uvm_pkg.out.sv uvm_pkg.sv
svdiag.pl -pdf -page landscape_letter,compress uvm_pkg.out.sv
…
[INFO ] svdiag: Generate : uvm.out.dot
[INFO ] svdiag: write digraph
[INFO ] svdiag: Executing: dot -T pdf -o uvm.out.pdf uvm.out.dot

PDF created, view via:
  > evince uvm.out.pdf
```

Refer to Appendix section VIII.C. for the resulting diagram, with `uvm_void` as the tree root.[4]

Table III: svdiag.pl options for class content detail.

Command-line option	Result	Default
`-class <classname>[,classname[,…]]`	Indicate set of focus classes	All classes are part of focus set
`-[no]vars`	Disable show variables	All class variables are shown
`-[no]types`	Disable show types	All class typedefs are shown
`-[no]constraints`	Disable show constraint names	All class constraint names are shown
`-[no]methods`	Disable show tasks and functions	All class methods are shown
`-hide`	Alias for disabling all class detail	All detail is shown
`-hide-lineage`	Disable all class detail except for focus set	All detail on all classes is shown
`-access <public|protected|local>`	Set the minimum level of access level to display for all class members	`local`, i.e., show everything

The svdiag.pl script provides runtime options to list only the classes that are of interest as well as their full lineage. Furthermore, the lineage classes may contain detail or no detail (i.e., class name and class attribute only). Finally, fine-grain control is available for the level of class member detail desired, as shown in Table III.

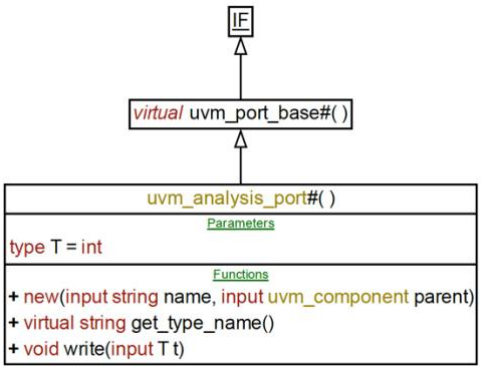


Figure 5: Result of focusing svdiag.pl on just the `uvm_analysis_port` tree from the UVM library.

For example, the entire UVM library may be parsed, similar to the command(s) above, but also specifying for svdiag.pl `-class uvm_analysis_port` to focus the output yields Figure 5. The user can combine the various use cases and

[4] Also available at: https://github.com/svloghack/svdiag/blob/main/doc/DVCon2026/appendix/C/uvm_lib.pdf.

command-line options to generate diagrams of the codebase necessary for documentation or learning and understanding.

B. svpre.pl

A SystemVerilog pre-processor, svpre.pl, can be employed to expand compile-time macros as well as filter out unused code, e.g. `` `ifdef `` … `` `else `` … `` `endif ``. Often `` `defines `` are employed throughout the codebase. These can hide potentially critical code for understanding the behavior of a class. Or, simply, macros could be used to enable or disable features of a reused class for specific project capabilities. For example, in Code 1 line 2, the `` `uvm_object_utils() `` macro contains types, variables, and methods that are not shown in Figure 3. By default, svdiag.pl ignores macro instantiations, though it always considers `` `ifdef/`ifndef `` constructs. In Figure 6, the `axi4_seq_item` class code was first pre-processed then a diagram generated:[5]

```
svpre.pl -I $UVM_HOME/src -macro-inst -keep-ws-read -keep-com \
         -o axi4_seq_item.out.sv axi4_seq_item.svh
svdiag.pl -o axi4_seq_item.pdf axi4_seq_item.out.sv
```

The pre-process step instantiates all known macros (`-macro-inst`), preserves whitespace for readability (`-keep-ws-read`) as well as any embedded comments (`-keep-com`). The `axi4_seq_item` class is output to the file, `axi4_seq_item.out.sv`, should appear as close to the original file as possible, but would also expand `` `uvm_object_utils() `` as defined in `$UVM_HOME/src/macros/uvm_object_defines.svh` (assuming `UVM_EMPTY_MACROS` is not defined). Refer to the appendix section VIII.D. for the output of svpre.pl execution for the `axi4_seq_item` class. Then, the svdiag.pl step, along with the dot program, generated the diagram in Figure 6.

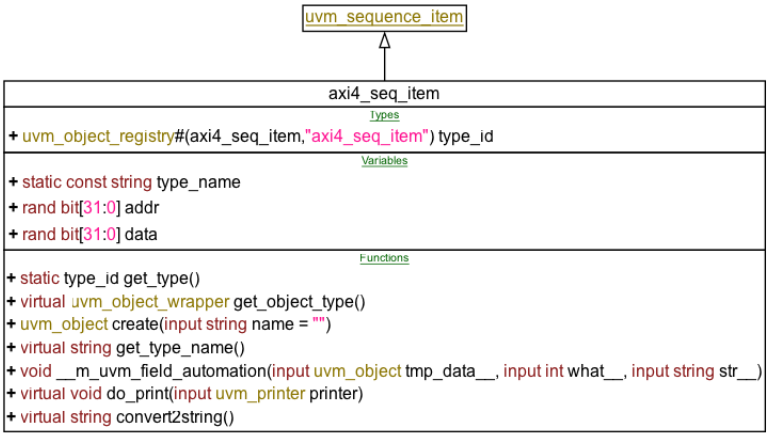


Figure 6: Diagram generated by svdiag.pl and dot after enabling pre-processing with svpre.pl.

The svpre.pl pre-processor differs from commercial pre-processors in that its primary goal is understanding the existing codebase. Of course, svpre.pl pre-processes necessary macros and constructs, but it also aims for readability based on the compile-time options provided. Consider a common usage of macro instantiation for parameter passing:

```
1   virtual class param_base#(int val, type T); endclass
2   class my_a extends param_base#(`PROJECT_A_PARAMS); endclass
3   class my_b extends param_base#(`PROJECT_B_PARAMS); endclass
```

Here, the opaque macro, in lines 2 and 3, provides the very implementation of the base class. While hunting down the `` `define `` for each project may be a one-time effort, svpre.pl is can simplify this action, especially for the newcomer.

Visually parsing through code can become cumbersome. Our pre-processor takes care to preserve comments and, as much as feasible, existing whitespace indentation. The pre-processor does not reformat code, though spaces may

[5] By default, svpre.pl does **not** instantiate macros to allow the user high fidelity control. Use `-macro-inst` to all macros, and `-macro-inst -no-uvm-macros` to only instantiate user macros.

be inserted between lexicographical tokens. Instead, it determines what code, comments, and whitespace will pass through as the user desires and after `macro instantiation and `ifdef/`ifndef resolution.

Both the svpre.pl pre-processor and svdiag.pl translator instantiate the same lexicographical interpreter (lexer). As such, the same command-line arguments are available to both and include finer grain control. For example, by default svdiag.pl does not instantiate any macro, thereby simply removing the code from the output, though `ifdef/`ifndef blocks are still resolved. An option is available to instantiate user macros but ignore UVM macros, or the vice versa. And all known macros and their definitions, if any, as well as the paths of all encountered `include, may be written to a file. The goal of a separate svpre.pl pre-processor furthers understanding of the existing codebase.

Combining the two scripts, svpre.pl and svdiag.pl, provides a powerful toolset to understand and visualize codebases. It can process an entire set of code and then be directed to generated diagrams based on a subset of that code, no need to separate the code out from the whole package. In Figure 7, in the entire UVM library [5], was pre-processed, parsed, and then a diagram generated by focusing only on a subset of uvm_reg and related classes, hiding the contents of the classes themselves to only show inheritance.

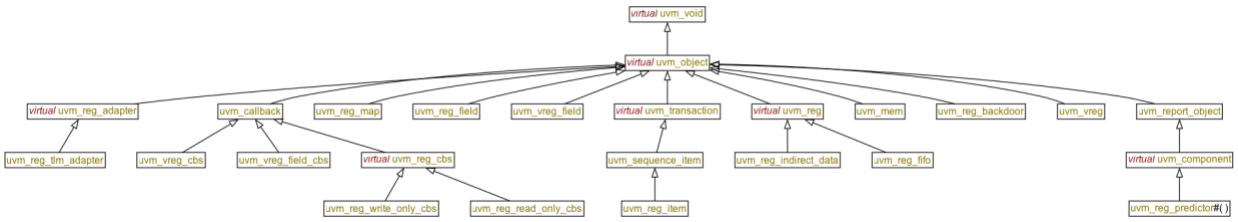


Figure 7: Class inheritance tree for uvm_reg and related classes.

IV. Visualizing the UVM Testbench

A. *uvmtopo.pl*

The uvmtopo.pl script employs the same backend abstract syntax tree handling found in svdiag.pl but replaces the SystemVerilog lexer and parser front ends with a UVM topology table parser. At SystemVerilog simulation runtime, the current testbench topology may be reported to the simulation log by:

```
// Generally, during end_of_elaboration_phase
uvm_root utop = uvm_root::get();
utop.print_topology();
```

The uvmtopo.pl script searches the provided log for the key phrase (though it is configurable):

```
reporter [UVMTOP] UVM testbench topology:
```

and consumes all text until the end of the table (i.e., the entire log need not be parsed). Each class, class type, and all class members are parsed from the topology table indicated in the simulation log, an internal model built, then output in the same visualization DOT language as svdiag.pl. Class member current assigned values are not reported.

Consider the topology report in Code 4. This is a common testbench construction with an environment instantiation constructing two agents, one active and one passive, and a scoreboard.

```
1  # UVM_INFO verilog_src/uvm-1.2/src/base/uvm_root.svh(579) @ 0: reporter [UVMTOP]
   UVM testbench topology:
2  # ----------------------------------------------------------------
3  # Name                         Type                      Size  Value
4  # ----------------------------------------------------------------
5  # uvm_test_top                 test                      -     @362
6  #   m_env                      m_environment             -     @376
7  #     m_agt_active             m_active_agent            -     @394
8  #       a_port                 uvm_analysis_port         -     @585
9  #       m_drv                  m_driver                  -     @549
```

```
10 #          rsp_port            uvm_analysis_port        -     @566
11 #          seq_item_port       uvm_seq_item_pull_port   -     @557
12 #        m_mon                 m_monitor                -     @575
13 #          m_port              uvm_analysis_port        -     @600
14 #        m_seqr                m_sequencer              -     @424
15 #          rsp_export          uvm_analysis_export      -     @432
16 #          seq_item_export     uvm_seq_item_pull_imp    -     @538
17 #          arbitration_queue   array                    0     -
18 #          lock_queue          array                    0     -
19 #          num_last_reqs       integral                 32    'd1
20 #          num_last_rsps       integral                 32    'd1
21 #      m_agt_passive           m_passive_agent          -     @404
22 #        m_mon                 m_monitor                -     @614
23 #          m_obs_port          uvm_analysis_port        -     @639
24 #        m_port                uvm_analysis_port        -     @624
25 #      m_sb                    scoreboard               -     @414
26 #        sb_exp_imp            uvm_analysis_imp         -     @648
27 #        sb_obs_imp            uvm_analysis_imp_obs     -     @657
28 # ----------------------------------------------------------------
```

Code 4: An example UVM testbench topology, sim_tb_2agents.log.

The visualization of the testbench is shown in Figure 8 and generated from Code 4 with the uvmtopo.pl script:

```
uvmtopo.pl -pdf sim_tb_2agents.log
[INFO ] uvmtopo: open file: sim_tb_2agents.log
[INFO ] uvmtopo: processed 1 files
[INFO ] uvmtopo: wrote file: sim_tb_2agents.dot
[INFO ] uvmtopo: wrote file: sim_tb_2agents.pdf, view via:

    > evince sim_tb_2agents.pdf
```

The root of the class tree is always at the bottom of the diagram with the `uvm_test_top test`. Only the variables group of class members is indicated per definition, as this is what is reported in the topology table. While class inheritance cannot be ascertained from the topology table, class member instantiations to their definitions are indicated via an arrow, working upward in the diagram. Again, UVM library key classes are presented in gold. Often the data member type is indicated as integral, shown in **red**, as an opaque classification of any SystemVerilog integral type.

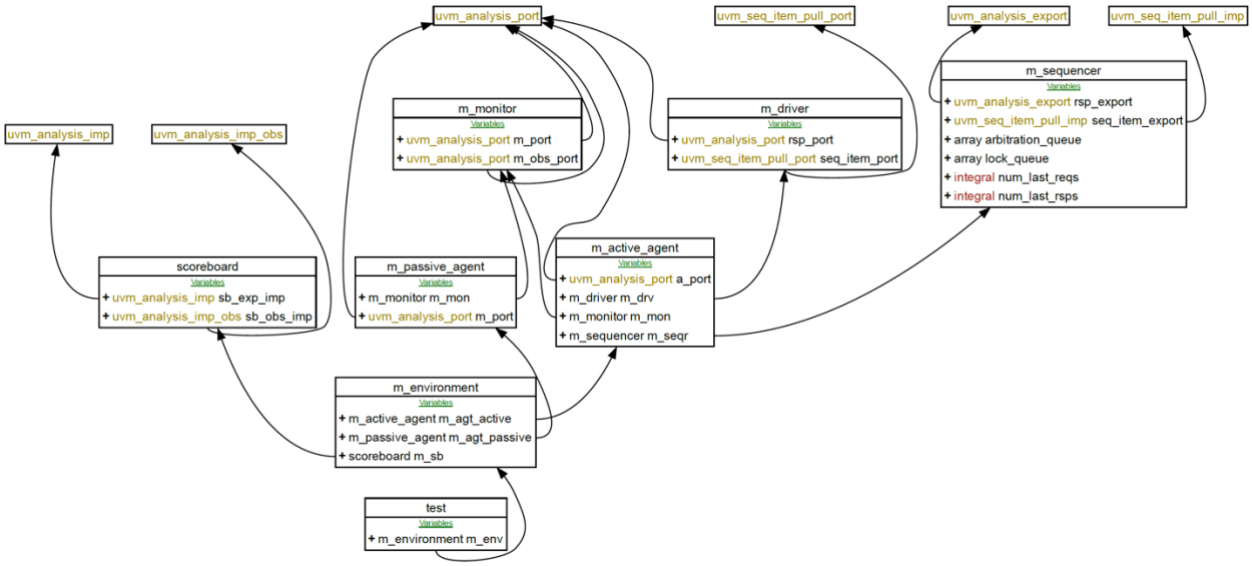


Figure 8: UVM testbench architecture of Code 4 as generated from uvmtopo.pl.

The UVM topology diagram is wholly dependent upon what is reported in the table. This, in turn, is dependent upon the type declared in the class, either by UVM macro instantiation or hand coded. In Figure 6, the `uvm_object_utils() expansion included the class-type information:

```
1  const static string type_name = "axi4_seq_item";
2  virtual function string get_type_name () ;
3    return type_name;
4  endfunction
```

If this class were reported in the topology, then it would be indicated as something like the following:

```
#      m_curr_item              axi4_seq_item            -      @1237
```

The m_curr_item is the variable member is some enclosing class, and axi4_seq_item the type as reported by the get_type_name() class function. However, this can become problematic for parameterized classes. For example, if the axi4_seq_item class were to parameterize the size of the address and data, then:

```
1  class axi4_seq_item#(int MAX_WIDTH = 31) extends uvm_sequence_item;
2    `uvm_object_param_utils(axi4_seq_item)
3    rand bit[MAX_WIDTH-1:0] addr, data;
4  endclass
```

But the `uvm_object_param_utils() definition does not include the type_name variable member nor get_type_name() function. As such, this same class instance in the topology would appear as potentially slightly misleading:

```
#      m_curr_item              uvm_sequence_item        -      @123
```

While an instance of, for example, axi4_seq_item#(64) is a uvm_sequence_item, the topology does not show the full picture. Care should be taken to properly report this class by, possibly, hand-coding the two class members.

```
1  const static string type_name = {$sformatf("axi4_seq_item#(%d)", MAX_WIDTH)};
2  virtual function string get_type_name();
3    return type_name;
4  endfunction
```

The system task, $sformatf, may be resolved at compile time and the constant static string assignment made. Now, the topology would indicate the more useful and accurate class type, as well as propagating to the UVM testbench topology diagram:

```
#      m_curr_item              axi4_seq_item#(64)       -      @123
```

Like svdiag.pl and svpre.pl, the uvmtopo.pl script can parse the entire topology table but focus the output on a specific class name or a set of class names, as well as a specific reported type or set of reported types. As the UVM testbench grows in complexity, this feature is necessary to home in on specific portions of the testbench, as necessary.

V. Future Work

The svpre.pl and svdiag.pl scripts were developed based on a common UVM test bench implementation and SystemVerilog observed. However, we expect to grow the toolset to represent latest techniques. The following are current limitations, but likely not a comprehensive set as it is biased towards the authors' coding capabilities.

SystemVerilog interface classes are currently not supported by the scripts. These constructs have become an integral part of testbench architecture and should be represented properly in the diagramming.

SystemVerilog packages are currently not supported in the internal model used by svdiag.pl. The keywords are parsed but ignored as its own namespace. The effect is a single union namespace for all packages provided in a single parse. For example, given one set of code with the same class name in differing packages, svdiag.pl will abort the

parse because duplicate names within one namespace is illegal. In a future version, svdiag.pl will support packages that may be visualized usefully.

Our SystemVerilog pre-processor, svpre.pl, does not support `` `line `` output. It is common split the compilation effort into two stages: a pre-processor to shape the input code and a compiler to translate the code into an executable (e.g., a simulation, for our purposes). This results in the code losing semblance to the original file and line number, which would make compile-time debug difficult. The `` `line `` compiler directive allows the compiler to retain that context. Neither svpre.pl nor svdiag.pl support generation or instantiation of the `` `line `` directive..

A subset of compiler delimiters is supported by the scripts. While not all are applicable to testbench structure, they are simply ignored for now. SystemVerilog delimiters not handled include: `` `undef ``, `` `undefineall ``, `` `resetall ``, `` `begin_keywords ``, `` `end_keywords ``, `` `celldefine ``, `` `endcelldefine ``, `` `default_nettype ``, `` `nounconnected_drive ``, `` `pragma ``, `` `timescale ``, and `` `unconnected_drive ``. Refer to the SystemVerilog standard Section 22. Compiler Directives for more information [2].

No functional coverage is yet represented by the scripts. Considering their integral importance to the project, this may be an omission to the understanding of the overall verification project cycle.

VI. Conclusion

The tool suite, svpre.pl, svdiag.pl, and uvmtopo.pl provide a powerful way to visualize both SystemVerilog existing codebases as well as the runtime simulated testbench architecture. They each may be used stand-alone, for example using only svpre.pl to pre-process a file to examine compile-time macro expansions, or in conjunction to give the engineer a clear picture of the testbench. These tools are free to access and use and available on our GitHub site: https://github.com/svloghack/svdiag.

VII. References

[1] L. Wall, "Perl," 2025. [Online]. Available: https://www.perl.org.

[2] IEEE, SystemVerilog IEEE 1800-2017, 2017.

[3] Wikipedia, "Unified Modeling Langauge," [Online]. Available: https://en.wikipedia.org/wiki/Unified_Modeling_Language. [Accessed 2 Nov 2025].

[4] J. Ellson, E. Gansner, Y. Hu and S. North, "dot - graphviz," version 10.0.1, 2024. [Online]. Available: https://www.graphviz.org.

[5] Acellera, "Universal Verification Methodology, version 1.2," 2014.

[6] G. Valure, "NaturalDocs," version 2.3.1, 2025. [Online]. Available: https://www.naturaldocs.org.

[7] D. van Heesh, "doxygen," version 1.14.0, 2025. [Online]. Available: https://www.doxygen.nl.

[8] Wikipedia, "Learning Styles," [Online]. Available: https://en.wikipedia.org/wiki/Learning_styles. [Accessed 2 Nov 2025].

[9] M. Kretzschmar, M. P. Gritti, J. Blandford, N. V. Shmyrev, B. Clark, C. G. Campmos and W. Bolsterlee, "Gnome/evince," 2018. [Online]. Available: https://gitlab.gnome.org/GNOME/evince.

VIII. Appendix

A. *svdiag.pl DOT output for Code 1.*

The DOT language output for class `axi4_seq_item` was generated via:

```
svdiag.pl -I $UVM_HOME/src -I $UVM_HOME/src/seq -hide-lineage \
          -font 10 -class axi4_seq_item -pdf axi4_seq_item.svh
```

Note, that `` `include "uvm_sequence_item.svh" `` was inserted in the file just prior to the class declaration. The include paths on the command-line were necessary to process that file and its included files.

```
1  //
2  // svdiag.pl Copyright (c) 2022-2025 Jamie Ridgeway <svloghack@gmail.com>
3  // Generated on  Sun Nov  2 10:11:22 2025 by jamie
4  //
5  // __GLOBAL__: classes: 2.
6  // __GLOBAL__: orphan classes: 1.
7  digraph G {
8      node [ fontname="arial", fontsize=10, shape=plaintext ];
9      nodesep=0.3
10     ranksep=0.3
11     // Take class: axi4_seq_item
12     // Take ancestor: uvm_sequence_item
13     // Take orphan ancestor: uvm_transaction
14     c4 [ label =
15         <<table title="uvm_transaction" border="0"
16                 cellpadding="0" cellspacing="0" cellborder="1" port="P">
17         // Title row/cell table
18         <tr><td>
19             <table border="0" cellborder="0" cellpadding="0" cellspacing="0">
20                 <tr><td align="center" balign="center" cellpadding="1"
   cellspacing="1"><u><font color="gold4">uvm_transaction</font></u></td></tr>
21             </table>
22         </td></tr>
23         </table>>
24     ];
25     c92 [ label =
26         <<table title="axi4_seq_item" border="0"
27                 cellpadding="0" cellspacing="0" cellborder="1" port="P">
28         // Title row/cell table
29         <tr><td>
30             <table border="0" cellborder="0" cellpadding="0" cellspacing="0">
31                 <tr><td align="center" balign="center" cellpadding="1"
   cellspacing="1">axi4_seq_item</td></tr>
32             </table>
33         </td></tr>
34         // Variables row/cell table
35         <tr><td>
36             <table border="0" cellborder="0" cellpadding="0" cellspacing="0">
37                 <tr><td align="center" balign="center" cellpadding="1"
   cellspacing="1"><font point-size="8"
   color="darkgreen"><u>Variables</u></font></td></tr>
38                 <tr><td align="left" balign="left" cellpadding="2"
   cellspacing="1" port="v93"><b>+</b> <font color="firebrick4">rand</font> <font
   color="firebrick4">bit</font>&#91;<font color="deeppink1">31</font>:<font
   color="deeppink1">0</font>&#93; addr</td></tr>
39                 <tr><td align="left" balign="left" cellpadding="2"
   cellspacing="1" port="v94"><b>+</b> <font color="firebrick4">rand</font> <font
   color="firebrick4">bit</font>&#91;<font color="deeppink1">31</font>:<font
   color="deeppink1">0</font>&#93; data</td></tr>
40             </table>
```

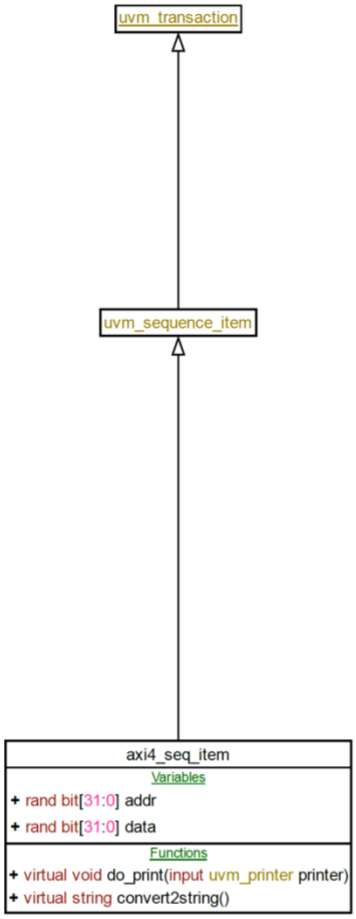
```
41          </td></tr>
42          // Functions row/cell table
43          <tr><td>
44              <table border="0" cellborder="0" cellpadding="0" cellspacing="0">
45                  <tr><td align="center" balign="center" cellpadding="1"
    cellspacing="1"><font point-size="8"
    color="darkgreen"><u>Functions</u></font></td></tr>
46                  <tr><td align="left" balign="left" cellpadding="1"
    cellspacing="1"><b>+</b> <font color="firebrick4">virtual</font> <font
    color="firebrick4">void</font> do_print(<font color="firebrick4">input</font>
    <font color="gold4">uvm_printer</font> printer)</td></tr>
47                  <tr><td align="left" balign="left" cellpadding="1"
    cellspacing="1"><b>+</b> <font color="firebrick4">virtual</font> <font
    color="firebrick4">string</font> convert2string()</td></tr>
48              </table>
49          </td></tr>
50          </table>>
51      ];
52      c3 [ label =
53          <<table title="uvm_sequence_item" border="0"
54                  cellpadding="0" cellspacing="0" cellborder="1" port="P">
55          // Title row/cell table
56          <tr><td>
57              <table border="0" cellborder="0" cellpadding="0" cellspacing="0">
58                  <tr><td align="center" balign="center" cellpadding="1"
    cellspacing="1"><font color="gold4">uvm_sequence_item</font></td></tr>
59              </table>
60          </td></tr>
61          </table>>
62      ];
63
64      // axi4_seq_item extends uvm_sequence_item
65      c3:P -> c92:P [dir=back arrowtail=empty];
66      //c92:v93 -> null; // (bit[31:0]) addr not defined in scope
67      //c92:v94 -> null; // (bit[31:0]) data not defined in scope
68
69      // uvm_sequence_item extends uvm_transaction
70      c4:P -> c3:P [dir=back arrowtail=empty];
71      //c3:v8 -> null; // (uvm_sequencer_base) m_sequencer not defined in scope
72      //c3:v9 -> null; // (uvm_sequence_base) m_parent_sequence not defined in
    scope
73
74      label="\n\nsvdiag.pl (C) 2022-2026 Jamie Ridgeway\nGNU General Public
    License\nGenerated by jamie on 01.03.2026\n";
75      fontname=arial;
76      fontsize=7.0;
77      fontcolor=lightgray;
78  }
```

B. Diagram for Code 1 with page size US letter and "fill" attribute

The `axi4_seq_item` class output, below, was generated with the following command:

```
svdiag.pl -I $UVM_HOME/src/base -I $UVM_HOME/src/seq -I $UVM_HOME/src -hide-lineage \
          -page letter,fill -font 10 -class axi4_seq_item -pdf axi4_seq_item.svh
```

Note, that `` `include "uvm_sequence_item.svh" `` was inserted in the file just prior to the class declaration. The include paths on the command-line were necessary to process that file and its included files. In the diagram, `uvm_transaction` is not known, and thus underlined, because it was not included in the parse, while `uvm_sequence_item` is known but details hidden (`-hide-lineage`). The large spacing between boxes is due to filling the available US letter-sized space (`-page letter,fill`).

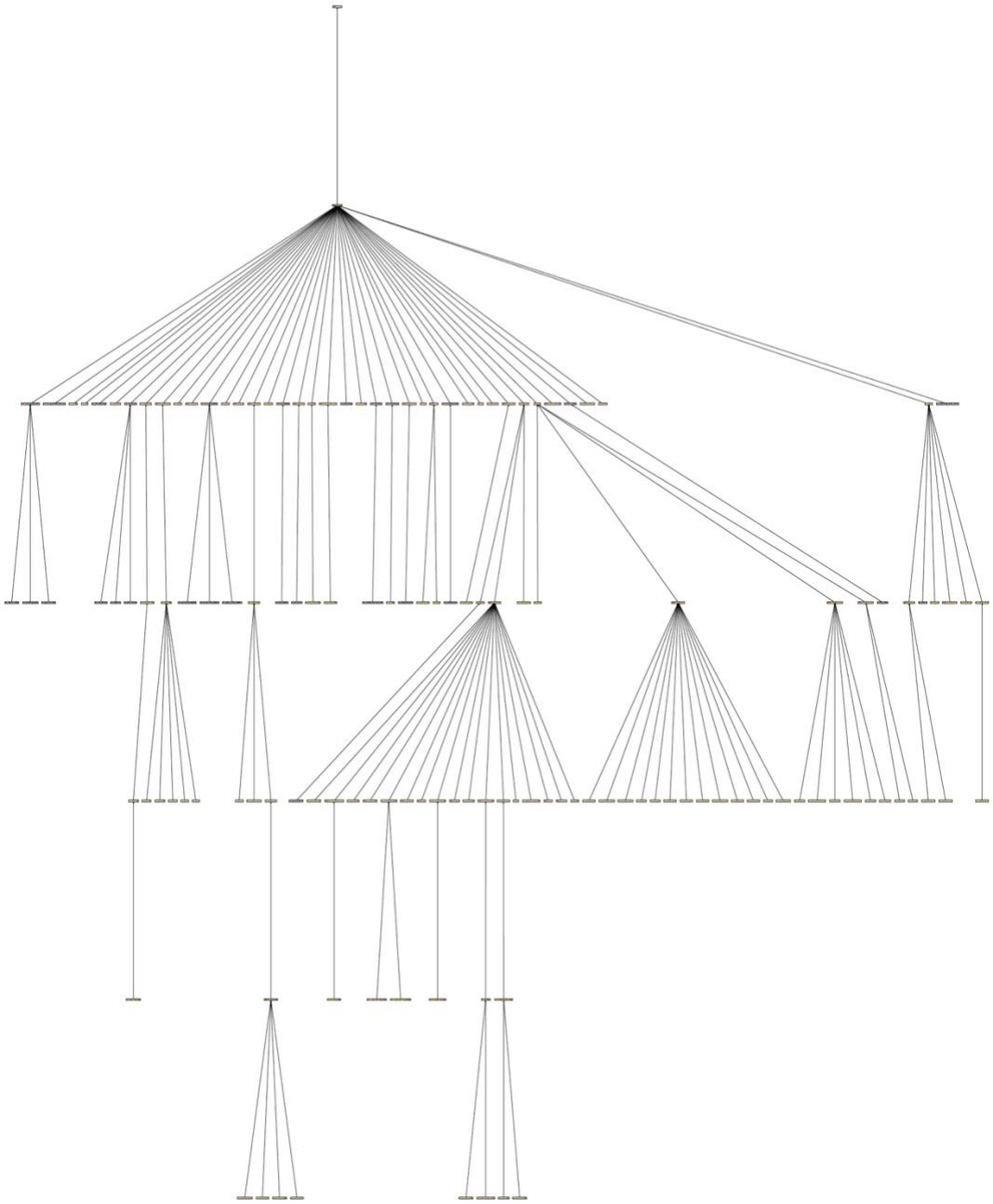


C. Diagram for UVM Library with page size US letter

A file, uvm_lib.sv, containing only `` `include "uvm_pkg.svh" `` for the UVM 1.2 version library, was pre-processed, parsed and diagram generated (showing only inheritance) via the commands following commands:

```
svpre.pl -I $UVM_HOME/src -macro-inst -o uvm_lib.out.sv uvm_lib.sv
svdiag.pl -hide-detail -class uvm_void -page letter,fill -o uvm_lib.pdf uvm_lib.out.sv
```

The tree root is the `uvm_void` class, the remainder of the tree is its lineage (able to see the original with a PDF viewer zoom, refer to our GitHub repo at https://github.com/svloghack/svdiag/tree/main/doc/DVCon2026/appendix/C).



D. Pre-processing axi4_seq_item.svh keeping whitespace and comments

The `axi4_seq_item` class output, below, was generated with the following command:

```
svpre.pl -I $UVM_HOME/src -macro-inst -keep-ws-read -keep-com \
         -o axi4_seq_item.out.sv axi4_seq_item.svh
```

Note, that `` `include "uvm_macros.svh" `` was inserted in the file just prior to the class declaration. The code below represents the output for just the `axi4_seq_item` class itself, from Code 1. The generated file also included all comments from `uvm_macros.svh`, and those were ignored here, for clarity.

```
1   class axi4_seq_item extends uvm_sequence_item ;
2
3
4
5      typedef uvm_object_registry # ( axi4_seq_item , "axi4_seq_item" ) type_id ;
6      static function type_id get_type ( ) ;
7        return type_id :: get ( ) ;
8      endfunction
9      virtual function uvm_object_wrapper get_object_type ( ) ;
10       return type_id :: get ( ) ;
11        endfunction
12
13     function uvm_object create ( string name = "" ) ;
14       axi4_seq_item tmp ;
15       if ( name == "" ) tmp = new ( ) ;
16       else tmp = new ( name ) ;
17       return tmp ;
18        endfunction
19
20     const static string type_name = "axi4_seq_item" ;
21     virtual function string get_type_name ( ) ;
22       return type_name ;
23        endfunction
24
25     function void __m_uvm_field_automation ( uvm_object tmp_data__ ,
26                                              int what__ ,
27                                              string str__ ) ;
28     begin
29       axi4_seq_item local_data__ ;
30       typedef axi4_seq_item ___local_type____ ;
31       string string_aa_key ;
32       uvm_object __current_scopes [ $ ] ;
33       if ( what__ inside { UVM_SETINT , UVM_SETSTR , UVM_SETOBJ } ) begin
34          if ( __m_uvm_status_container . m_do_cycle_check ( this ) ) begin
35              return ;
36          end
37          else
38              __current_scopes = __m_uvm_status_container . m_uvm_cycle_scopes ;
39       end
40       super . __m_uvm_field_automation ( tmp_data__ , what__ , str__ ) ;
41       if ( tmp_data__ ! = null )
42              if ( ! $cast ( local_data__ , tmp_data__ ) ) return ;
43
44       end
45     endfunction
46
47      // Variable: addr
48      // Address targeted for reading or writing.
49      rand bit [ 31 : 0 ] addr , data ;
50
51      // Function: do_print
```

```
52      // UVM compatible printing.
53      extern virtual function void do_print ( uvm_printer printer ) ;
54      extern virtual function string convert2string ( ) ;
55
56  endclass
```