Universal Hardware Data Model

Alain Dargelas, PhD,
Data Model Solutions, LLC,
Santa Clara, CA, USA
alain.dargelas@gmail.com

Henner Zeller, Google,
Mountain View, CA, USA,
hzeller@google.com

***Abstract*—The Universal Hardware Data Model (UHDM) [1] open-source project aims at enabling open-source Electronic Design Automation (EDA) tools to support the entire SystemVerilog 2017 Standard [2]. On one side parsers like SureLog [3] parse and populate the UHDM model and on the other side client tools like Synthesis, Simulation, Linter and so on read back from the compiled model and perform their respective tasks.**

Keywords—SystemVerilog 2017, Open-source Verilog Parser, Persistent Data Model, Verilog Procedural Interface.

I. INTRODUCTION

The EDA open-source ecosystem lacks at present the complete support for the SystemVerilog 2017 language. The goal of the UHDM Data Model is to fill this void and enable the entire ecosystem to support SystemVerilog in its entirety.

To that extend a complete SystemVerilog 2017 Preprocessor/Parser/Compiler/Elaborator has been developed: SureLog [3]. SureLog parses the user's SystemVerilog and performs the tasks of compiling and elaborating the design and testbench then populates the UHDM data model which is persisted on disk. UHDM is then used in development adaptations of applications like Yosys [4], Verilator [5] and is translated into their respective native data structures by translation layers like Yosys-UHDM integration [6], Verilator-UHDM integration [6], or used as a standalone data structure by the client application. The ecosystem is also open for other parsers/compilers to populate the UHDM model.

At the time of this publication we are working toward Synthesizing and Simulating the OpenTitan Root of Trust Design [7] using the SureLog-UHDM-Verilator/Yosys flow to validate the complete stack.

II. UHDM

A) SystemVerilog Object Model

Since the beginning of the Verilog language, an integral part of the Verilog Standard is the Verilog Object Model. We have decided to follow the SystemVerilog Object Model as closely as possible as the Schema for the UHDM Model. One of the advantages is the widespread knowledge of the Verilog Object Model and its modern interface, the Verilog Procedural Interface (VPI) [8]. The Standard Verilog Object Model was designed as an API to Simulators and is fully elaborated and bit-blasted, we had to take some digressions like making the Elaboration optional post re-load and avoid bit-blasting as much as possible, as bit blasting makes a lot of client applications not efficient and has to be delayed in the flow or avoided as much as possible.

B) UHDM Features

The UHDM model is captured in a Yaml-like [9] markup language, and represents the entire schema present in the SystemVerilog Object Model in pages 973-1050 of the IEEE Std 1800-2017, one yaml file per diagram in the model.

Examples of UHDM models captured in Yaml:

```
- class_def: instance
  - extends: scope
  - property: definition_name
    vpi: vpiDefName
    type: string
    card: 1
- group_def: stmt
  - class_ref: scope
  - class_ref: atomic_stmt
  - class_ref: expr
  - group_ref: assertion
```

We had to support several concepts in order to properly capture all the types of relations present in the diagrams: Abstract class (class def), Concrete object (obj def), Inheritence (extends), Group (group def, similar to void* with runtime checks for group membership), Composition (class ref/object ref/group ref).

From this Yaml meta model, a code gererator developed by the authors generates the following code automatically:

- C++ implementation (Classes)

- Serialization/deserialization using the Capnp library [10]. To be noted that the Capnp code is hidden from the API user, and the deserialized UHDM data structure (C++ classes) is read/writeable (unlike Capnp typical data structure use model which are read-only after deserialization).

- A Walker that creates a humanly readable text dump of the UHDM data model

- A C++ Listener Design Pattern over the entire model

- An optional Elaborator/Uniquification that can be invoked pre-serialization or post-deserialization

- The corresponding C Standard VPI interface as a facade to the C++ model

C) Deviations from the Standard

As mentioned previously, there are a few instances where the Simulation-centric bit blasted Verilog Object Model was not convenient. This is where we took some liberties to deviate from the Standard.

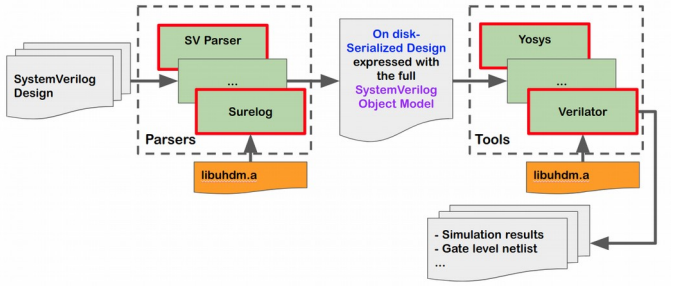
This publication resulted (in part) from work supported by Google.
The content is solely the responsibility of the authors and does not necessarily represent the official views of Google.

- The “ref obj” is used in UHDM everywhere a reference to a named object is necessary instead of presenting the bonded object itself, the ref obj is presented to the user and the vpiActual property of the ref obj points to the object the ref obj is binding to. In the Standard VPI presented by commercial EDA tools this indirection is not present, but comes at the price of having another intermediate data structure to perform the binding behind the scenes.
- UHDM is not bit-blasted, unlike the Standard Model. Arrays for instance are kept as arrays and not individual bits of the array. The memory savings both in the persisted representation and at runtime is colossal. Client applications can perform their own bit-blasting on demand.
- By default UHDM is not elaborated, again deviating from the Standard, but an optional Elaboration is offered and it can be invoked before serialization or after deserialization.
- A few Groups of objects are extended to support “ref obj” or similar more encompassing object types.
- The package import statement is made explicitly part of the model, it facilitates symbol lookup in the client applications.
- A unique ID, a ClientData (void*) pointer and UHDM object type enums are offered as convenient extensions for client applications that wishes to use the UHDM data model as their main runtime datastructure (free standing). As a side note, at the time of this writing there are a couple of open-source projects that are in the process of creating their own applications based on UHDM and they make use of these extensions.
- A “design” vpiHandle is added as the top level handle of a design so multiple designs or partitions of a single design can coexist at the same time. This also opens the possibility for supporting multiple language (VHDL/Verilog) designs in the future. All objects of a design root at that object. In comparison the standard VPI has a NULL top level object to iterate top level constructs ie.: *vpi_iterate(vpiModule, NULL)*.
- All of the deviations are documented in the Yaml models.

D) UHDM in the Compiler flow

UHDM comes in the form of a dynamic or static C++ library that is used in the SureLog parser (or other parser) to generate the persistent SystemVerilog Object Model after Parsing, Compilation and partial Elaboration (SureLog implementation). The same libuhdm.a library is then used in the respective client applications like a Simulator (Verilator) or a Synthesis tool (Yosys) to retrieve the persisted object model and populate the respective client application data structures.

Illustration of the flow:



E) Client-side APIs

1. The C++ API is comprised of a set of classes, that represent
- virtual classes (literally using virtual classes), i.e. the “instance” class in the Verilog Object Model
- classes that represent the concrete objects in the model like the “module” object.
- getter and setter methods that follow the VPI naming scheme.

C++ API header example:

```
namespace UHDM {
  class instance : public scope {
  public:
    bool VpiDefName(const std::string& data);
    …
  };
  class module : public instance {
  public:
    virtual const BaseClass* VpiParent() const;
    virtual const std::string& VpiFile() const;
    …
  };
};
```

2. The VPI C API implements the standard *sv_vpi_user.h* functions like *vpi_get*, *vpi_get_str*, *vpi_get_value*, *vpi_scan*, *vpi_iterate*, *vpi_get_by_name* for all objects in the SystemVerilog Object Model. The integer values returned from vpi_get or passed as arguments follow the *vpi_user.h* and *sv_vpi_user.h* defines.

3. A C++ Listener Design Pattern is offered on top of the C++ and VPI API allowing the client application that is interested in a convenient way to skip over relations and be able navigate the data model.

Example of custom listener code:

```
class MyVpiListener : public VpiListener {
protected:
  void enterModule(const module* object,
                   const BaseClass* parent,
                   vpiHandle handle,
                   vpiHandle parentHandle) {
    const char* parentName =
                 vpi_get_str(vpiName, parentHandle);
  }
   …
};

int main (int argc, char** argv) {
  UHDM::Serializer ser;
  std::string uhdmfile = argv[0];
  std::vector<vpiHandle> designs =
                         ser.Restore(uhdmfile);
  MyVpiListener* listener = new MyVpiListener();
  listen_designs(designs,listener);
}
```

III. SURELOG

A) *Preprocessor, Parser*

We mentioned that the first implementation of a parser to generate the UHDM Object Model is the SureLog parser. SureLog is an Antlr4.7 [11] based multi-threaded or multiple processes preprocessor and parser for the entire SystemVerilog 2017. The grammar files have been tested on a wide range of open-source cores and SystemVerilog UVM testbench code.
Both the preprocessor and the parser are incremental, they only recompile source code that changed and persist the syntax trees on disk using a mechanism developed by the author on top of Antlr. This makes up for the relatively slow first compile but very fast subsequent incremental compiles. The syntax trees are represented with a child-sibling ID scheme, 0 terminated, which allows for elegant and fast tree traversal used in the subsequent compilation and elaboration phases.
The parser precompiles the UVM and OVM packages syntax trees which allows for fast recovery when compiling a user design. It takes about 70 seconds on a single thread to precompile the UVM package and about 300ms to retrieve it from disk.
The preprocessor/parser supports both the “Interpreted” Verilog compilation semantic and the “Separate Compilation Unit” semantic through an option. It also supports Libraries and Configurations.

B) *Compiler*

The compiler currently compiles all the Synthesizable subset of the language, a fair amount of Assertions, and Classes definitions, it creates placeholders for Constraints and other Testbench related concepts in the UHDM model. The final goal is to compile UHDM models for the entire Testbench aspects of the language as well.

C) *SureLog and UHDM Elaborators*

The SureLog Elaborator supports all flavors of parameter passing, including defparams. It does perform generate statement evaluations and hierarchy tree expansion. As an example, the resulting UHDM model only contains the active branch in an if-else generate statement, SureLog does not populate UHDM with the inactive branch of the statement. We have tested it on designs like BlackParrot [12] and OpenTitan [7] which make a good use of these features. The SureLog Elaborator only uniquifies the instances though, it does not uniquify nets.

After the SureLog Elaboration, the model is still a Folded Model. The task to unfold the model completely is left for the UHDM Elaborator which creates unique net IDs and deep clone all the statements in the instance tree.

D) *Design Coverage*

SureLog produces a form of Coverage of objects represented in UHDM in regard to the original syntax tree, highlighting places that are not compiled or elaborated, or still unsupported statements in the SureLog to UHDM translation.
Legend:
- Grey is source text that has a corresponding object covered in the UHDM Model
- White text is not processed at all (No AST representation like comment sections, or unaccounted for statements like “end statement” that does not need a corresponding object in the UHDM model)
- Red is source text with no representation in UHDM but present in the Parser AST. In the case below the else branch of the if-else generate statement is not active hence not populated in the UHDM model. Only the if branch (Deemed active by the SureLog Elaborator) has corresponding objects in UHDM. In that case, it is OK for the code not to be in UHDM. We still mark it in red to warrant visual inspection. Users can ask themselves the question: “Is this code really intended to be left out?”. On the other hand unsupported statements are not OK and require attention on our end when supporting new designs.

```
233:     always_ff @(posedge clk_i or negedge
234:       if (!rst_ni) begin
235:         mult_state_q <= MULL;
236:       end else begin
237:         if (mult_en_internal) begin
238:           mult_state_q <= mult_state_d;
239:         end
240:       end
241:     end
242:
243:     // States must be knwon/valid.
244:     `ASSERT_KNOWN(IbexMultStateKnown, mul
245:
246:   // The fast multiplier uses one 17 bit
247:   // and MULH instructions in 4 cycles.
248:   end else begin : gen_multdiv_fast
249:     logic [15:0] mult_op_a;
250:     logic [15:0] mult_op_b;
251:
252:     typedef enum logic [1:0] {
253:       ALBL, ALBH, AHBL, AHBH
254:     } mult_fsm_e;
255:     mult_fsm_e mult_state_q, mult_state_d
256:
257:     // The 2 MSBs of mac_res_ext (mac_res
258:     // 1. The 2 MSBs of the multiplicants
```

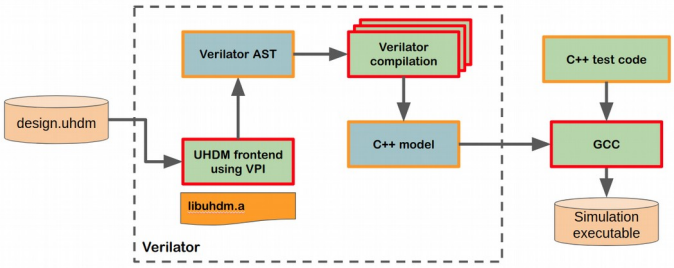
A convenient hyperlinked top level page is generated with all the files compiled and their UHDM coverage.
In the example below, the low coverage is due to inactive branches in the generate statements. A Quick view allows the reader to browse only the “uncovered code” allowing identification of unsupported constructs. This mechanism allows us to quickly identify the parts of the compiler that requires our attention when supporting a new RTL design.

IV. SURELOG-UHDM-VERILATOR

At the time of this writing we are actively working on simulating the OpenTitan core design using the SureLog-

UHDM-Verilator flow. The Verilog model converges in Verilator and we are making it working module by module.

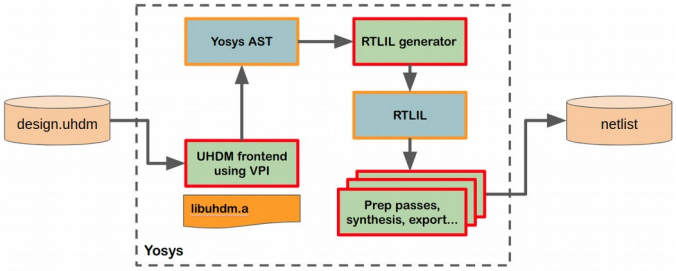
Verilator integration:



V. SURELOG-UHDM-YOSYS

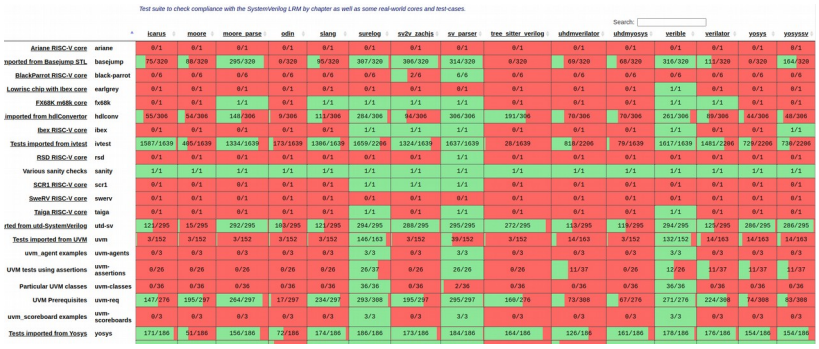
At the time of this writing we are actively working to Synthesize the OpenTitan core design using the SureLog-UHDM-Yosys flow. All modules in the design compile and we are at the stage where we are debugging logic errors module by module.

Yosys integration:



VI. SV-TESTS COVERAGE

SureLog is running part of the sv-tests [13] regression framework and enjoys the top rank in terms of numbers of passing tests among all open-source parsers. It is also the slowest due to the use of the C++ Antlr runtime and the full compilation and serialization it does perform.
sv-tests HTML report:



Test suite to check compliance with the SystemVerilog LRM by chapter as well as some real-world cores and test-cases.

Search:

#		icarus	moore	moore_parse	odin	slang	surelog	sv2v_zachjs	sv_parser	tree_sitter_verilog	uhdmverilator	uhdmvpssys	verible	verilator	yosys	yosyssv
Ariane RISC-V core	ariane	0/1	0/1	0/1	0/1	0/1	0/1	0/1	0/1	0/1	0/1	0/1	0/1	0/1	0/1	0/1
nported from Basejump STL	basejump	75/320	68/320	295/320	0/320	95/320	307/320	306/320	314/320	0/320	69/320	68/320	316/320	111/320	0/320	164/320
BlackParrot RISC-V core	black-parrot	0/6	0/6	0/6	0/6	0/6	0/6	2/6	0/6	0/6	0/6	0/6	0/6	0/6	0/6	0/6
Lowrisc chip with ibex core	earlgrey	0/1	0/1	0/1	0/1	0/1	0/1	0/1	0/1	0/1	0/1	0/1	1/1	0/1	0/1	0/1
FX68K m68k core	fx68k	0/1	0/1	1/1	0/1	1/1	1/1	1/1	1/1	0/1	0/1	0/1	1/1	1/1	0/1	0/1
imported from hdlConvertor	hdlconv	55/306	54/306	148/306	9/306	111/306	284/306	94/306	306/306	191/306	70/306	70/306	261/306	89/306	44/306	48/306
Ibex RISC-V core	ibex	0/1	0/1	0/1	0/1	0/1	1/1	1/1	1/1	0/1	0/1	0/1	1/1	0/1	0/1	1/1
Tests imported from ivtest	ivtest	1587/1639	405/1639	1334/1639	173/1639	1306/1639	1659/2206	1324/1639	1637/1639	28/1639	818/2206	79/1639	1617/1639	1481/2206	729/2206	736/2206
RSD RISC-V core	rsd	0/1	0/1	0/1	0/1	0/1	0/1	0/1	1/1	0/1	0/1	0/1	0/1	0/1	0/1	0/1
Various sanity checks	sanity	1/1	1/1	1/1	1/1	1/1	1/1	1/1	1/1	1/1	1/1	1/1	1/1	1/1	1/1	1/1
SCR1 RISC-V core	scr1	0/1	0/1	0/1	0/1	0/1	1/1	1/1	1/1	0/1	0/1	0/1	0/1	0/1	0/1	0/1
SweRV RISC-V core	swerv	0/1	0/1	0/1	0/1	0/1	0/1	0/1	0/1	0/1	0/1	0/1	0/1	0/1	0/1	0/1
Taiga RISC-V core	taiga	0/1	0/1	0/1	0/1	0/1	1/1	0/1	1/1	0/1	0/1	0/1	1/1	0/1	0/1	0/1
rted from utd-SystemVerilog	utd-sv	121/295	15/295	292/295	103/295	121/295	294/295	288/295	295/295	272/295	113/295	119/295	294/295	125/295	286/295	286/295
Tests imported from UVM	uvm	3/152	3/152	3/152	3/152	3/152	146/163	3/152	89/152	3/152	14/163	3/152	132/152	14/163	14/163	14/163
uvm_agent examples	uvm-agents	0/3	0/3	0/3	0/3	0/3	3/3	0/3	3/3	0/3	0/3	0/3	3/3	0/3	0/3	0/3
UVM tests using assertions	uvm-assertions	0/26	0/26	0/26	0/26	0/26	26/37	0/26	26/26	0/26	11/37	0/26	12/26	11/37	11/37	11/37
Particular UVM classes	uvm-classes	0/36	0/36	0/36	0/36	0/36	36/36	0/36	2/36	0/36	0/36	0/36	36/36	0/36	0/36	0/36
UVM Prerequisites	uvm-req	147/276	195/297	264/297	17/297	234/297	293/308	195/297	295/297	160/276	73/308	67/276	271/276	224/308	74/308	83/308
uvm_scoreboard examples	uvm-scoreboards	0/3	0/3	0/3	0/3	0/3	3/3	0/3	3/3	0/3	0/3	0/3	3/3	0/3	0/3	0/3
Tests imported from Yosys	yosys	171/186	51/186	156/186	72/186	174/186	186/186	173/186	184/186	164/186	126/186	161/186	178/186	176/186	154/186	154/186

Future work includes optimizing the Antlr runtime for parsing speedup and finishing the complete language support in the compiled UHDM model along with ensuring complete support of all the open-source popular designs and their UVM tests.

REFERENCES

[1] *UHDM:* https://github.com/alainmarcel/UHDM

[2] *SystemVerilog 2017:* http://ecee.colorado.edu/~mathys/ecen2350/IntelSoftware/pdf/IEEE_Std 1800-2017_8299595.pdf

[3] *SureLog:* https://github.com/alainmarcel/Surelog

[4] *Yosys:* http://www.clifford.at/yosys/

[5] *Verilator:*https://www.veripool.org/wiki/verilator

[6] *UHDM - Verilator and Yosys integration:* https://github.com/alainmarcel/uhdm-integration
https://github.com/antmicro/verilator/tree/uhdm-verilator
https://github.com/antmicro/yosys/tree/uhdm-yosys

[7] *OpenTitan design:* https://opentitan.org/

[8] *VPI:* https://ieeexplore.ieee.org/document/496013

[9] *Yaml:* https://yaml.org/

[10] *Capnp:* https://capnproto.org/capnp-tool.html

[11] Antlr: https://www.antlr.org/index.html

[12] Blackparrot: https://github.com/black-parrot

[13] sv-tests; https://github.com/SymbiFlow/sv-tests