Universal Hardware Data Model

Alain Dargelas
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 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, Linters and so on read back from the compiled model and perform their tasks.**

Keywords—SystemVerilog 2017, Open Source Parser, Persistent Data Model

I. INTRODUCTION

The EDA Opensource 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 Preprocessor/Parser/Compiler/Elaborator has been developed: Surelog [3]. Surelog produces the data that populates the UHDM Data Model which is persisted on disk. UHDM is then used in client 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 system is also open for other parsers/compilers to also 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.

- A Serialization/deserialization using the Capnp library [10]. To be noted that the Capnp code is hidden from the user and the deserialized UHDM data structure is read/writeable

This work is funded by Google.

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 applications not efficient.

B) UHDM Features

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

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

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

- a C++ implementation (Classes)

(unlike Capnp typical data structures which are read-only after deserialization).

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

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

- An optional Elaborator that can be invoked post-deserialization

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

C) Deviations from the Standard

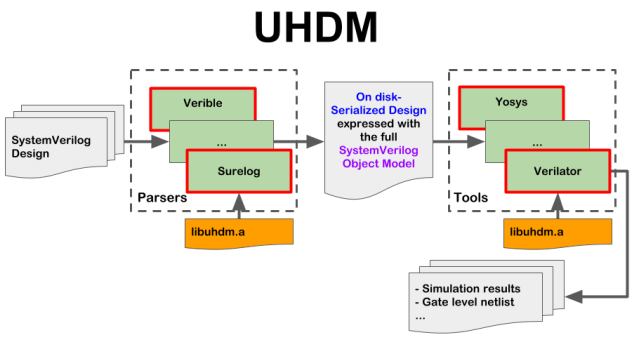
As mentioned previously, there are a few instance where the Simulation-centric bit blasted Verilog Object Model

was not convenient. This is where we took some liberties to deviate from the Standard.
- 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 points to the object the ref obj is binding to. In EDA tools this indirection is hidden, but comes at the price of having another intermediate data structure.
- 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 “obj ref” or similar more encompassing object types.
- The package import statement is 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 model. 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.

All of these deviations are documented in the Yaml model.

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. 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.



E) Client-side APIs

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 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.

```
namespace UHDM {

  class instance : public scope {
  public:

    bool VpiDefName(const std::string& data);
    …

  class module : public instance {
  public:

    virtual const BaseClass* VpiParent() const final { return
vpiParent_; }

    virtual const std::string& VpiFile() const final;
    …
```

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 defines.

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.

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

int main (int argc, char** argv) {
  UHDM::Serializer serializer;
  std::vector<vpiHandle> restoredDesigns =
serializer.Restore(fileName);
  MyVpiListener* listener = new MyVpiListener();
  listen_designs(restoredDesigns,listener);
  return 0;
}
```

II. 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 preprocessor and parser for the entire SystemVerilog 2017. The grammar files have been tested on a wide range of Open Source cores and SystemVerilog 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. 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 2 minutes on a single thread to precompile the UVM package and about 300ms to retrieve it from disk.
The preprocessor/parser supports 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 creates placeholder for Classes, 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) Elaborator

The Elaborator supports all flavors of parameter passing, including defparams. It does perform generate statement evaluations and hierarchy tree expansion. We have tested it on designs like BlackParrot and Earlgrey which make a good use of these features.

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.
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 things like “end statement” that does not have an explicit object in the UHDM model)
- Red text is not present in UHDM but present in the AST, in the case below the else branch of the if-else generate statement is not active hence not populated in the UHDM model.

```
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.

Overall Coverage: 97.3%

Cov: 31.4%	**../src/lowrisc_prim_diff_decode_0/rtl/prim_diff_decode.sv**
Cov: 36.1%	**../src/lowrisc_ip_aes_0.6/rtl/aes_sbox.sv**
Cov: 42.1%	**../src/lowrisc_ibex_ibex_core_0.1/rtl/ibex_alu.sv**
Cov: 45.8%	**../src/lowrisc_prim_all_0.1/rtl/prim_present.sv**

III. SURELOG-UHDM-VERILATOR

At the time of this writing we are actively working on simulating the Earlgrey core design using the Surelog-UHDM-Verilator flow. The Verilog model converges and we are making it working module by module.

IV. SURELOG-UHDM-YOSYS

At the time of this writing we are actively wokring to Synthesize the Earlgrey 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.

V. SV-TESTS COVERAGE

Surelog is running part of the SV-TEST [12] regression framework and enjoys the top rank in terms of numbers of passing tests among all parsers. It is also the slowest due to the use of the C++ Antlr runtime and the full compilation and serialization it does perform.
Future work includes optimizing the Antlr runtime for parsing speedup and finishing the complete language support in the compiled UHDM model.

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*

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